Project 2: User Programs

Preliminaries

> Fill in your name and email address.

Ivory E. S 2000012957@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.

I refered to xv6-code to implement a function array to run different system call by syscall number.

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.

```
struct argument_bearer{
    char buf[ARGSIZE];
    uint32_t argc;
    char *argv[MAX_USERARG_CNT];
    struct thread *parent;
};
```

A new structure is defined to pass argument. We save the raw data in buf array, argument number in argc and the pointer to each argument string in argv.

The parent is passed to implement syscall "wait" and "exec".

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?

Place the arg string in stack and update corresponding argv[] to a user address

Then write argv array with new address in stack

Cut the command at ARGSIZE - 1 if it was too long. The inserted content length is less than a page, so the stack won't overflow because of argument passing.

RATIONALE

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

It implements the iteration of all the argument separated with a specified delimiter by invoking strtok_r with save_ptr from last function call.

> 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.

This phrasing work doesn't need any privileged operation. It is better to do it in user-mod to avoid some risky arguments insertion to kernel.

And this task transfer to lower level alleviates the kernel load and reduce kernel code length for complicated string processing.

In recent research, the role of kernel becomes insignificant and many drivers can bypass. Perhaps the drivers can realize self-government one day.

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
{
    ...
#ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir; /**< Page directory. */
   int exit_status; /*save exit status for WAIT function to read */
   struct semaphore wait_child_load; /* wait child load finish in EXEC function*/
   struct semaphore end_process; /* wait child end in WAIT function or
    in EXEC function changing child parent pointer to NULL before child thread
exits */
   struct semaphore parent_sema; /* changing parent pointer in child thread when
exits */
   struct list child_list; /* link all the child thread in a list*/
   struct list file_list; /* link all the file pointer in a list */
   struct list_elem as_child; /* be linked as an element in child_list */
   struct thread *parent; /* record parent, if parent is NULL, it can delete
itself.
   Or it must be recycled when its parent exits*/
   struct file *load_file; /* the file loaded. It denies to be written before the
process ends */
#endif
    ...
};
```

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

```
typedef struct {
    int fd; /* file descriptor for process */
    struct file *f; /* file pointer in file system */
    struct list_elem elem; /* element in file_list per process */
}file_descriptor;
```

File descriptors are unique just within a single process.

ALGORITHMS

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

By using the **static int get_user (const uint8_t *uaddr)** and **static bool put_user (uint8_t *udst, uint8_t byte)** offered by lab document.

In page_fault handler. If the page fault caused by kernel(not user), we change the return value(eax) to -1 and program counter to the place saved in eax.

```
/*
  get 4 bytes a word from user address uaddr
*/
static bool
get_word(uint8_t *uaddr, uint32_t *arg) {
  uint32_t res = 0;
  int tmp = 0;
  int i = 0;
  for (i = 3 ; i >= 0 ; --i) {
    tmp = get_user(uaddr + i);
    if (tmp == -1) return false;
    res = res << 8 | (tmp & 0xff);
  }
  *arg = res;
  return true;
}

/*
  get the ord th argument
  the start address of the argument is user_stack + ord * 4
*/
static bool
get_arg(int ord, uint32_t *arg) {
  if (!get_word(user_stack + ord * 4, arg)) return false;
  return true;
}

/*
  get content from user address for exact n bytes
  return false due to invalid memory visit
*/
```

```
static bool
getnbuf(char *uaddr, char *buf, size_t n) {
  size_t i;
  int tmp;
  for(i = 0 ; i < n ; ++i) {
    if ((tmp = get_user((uint8_t *)uaddr)) == -1) {
      return false;
    }
    *buf = (tmp & 0xff);
    ++buf;
    ++uaddr;
  }

  return true;
}
/*
  put content to user address for exact n bytes
  return false due to invalid memory visit
*/
static bool
putnbuf(char *uaddr, char *buf, size_t n) {
  size_t i;
  for(i = 0 ; i < n ; ++i) {
    if(!put_user((uint8_t *)uaddr, *buf)) return false;
    uaddr++;
    buf++;
  }
  return true;
}

/*
  get a string from user address
  return -1 when visit invlaid memory
  of return the length of string
  cut the string when its length greater than n
*/
static int
getnstr(char *uaddr, char *buf, size_t n) {
  size_t i;
  int tmp;
  for(i = 0 ; i < n ; ++i) {
    if((tmp = get_user((uint8_t *)uaddr)) == -1) {
      return -1;
    }
    *buf = (tmp & 0xff);
    if (*buf == '\0') return i;
    ++buf;
    ++uaddr;
  }

  return n;
}
```

The function **get_word** is used to get four bytes a time

The **get_arg** is a wrapper of **get_word** for different argument

getnstr is generally used to read filename and refuse to process the file operation with filename longer than 14 bytes (after return)

getnbuf and **putnbuf** are used to read/write many bytes (about 70000?) from user address and then write/read them to file system

> 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?

(Is this a system call **exec** and child thread runs **start_process**?)

The least possible number is **1** when the 4096 bytes fit into one page exactly.

The greatest possible number is **2**, because the 4096 bytes may cross two pages.

For 2 bytes

The least possible number is **1** when they are in the same page and the greatest possible number is **2** when they are across pages.

I implement these copy function by the assistance of MMU(Memory Management Unit). So it may reduce the times to check page table using TLB(Translation Lookaside Buffer)?

If all the page mapping relations are cached in TLB, we do not need to inspect page table. So the possible least number is zero.

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

The wait use the semaphore **end_process** to wait child process ends. If the child process has already terminated, it is not deleted and the semaphore **end_process** is set to **1**.

The dead child process can be recycled in **process_wait**

And in termination, modify all the parent pointer of running child processes to NULL and they can deleted themselves freely.

Or the child process has already died and the parent process clears their information in memory.

> 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 false state returned by **get_arg**, **getnbuf**, **putnbuf** and **getnstr** means the invalid memory visit. Call the **exit_print** to end the thread.

In **read** and **write** system call, we write a fixed length buffer in function and write the buffer in a while loop. A flag is used to mark a invalid memory visit and break the loop immediately.

The lock is released and in the end of function. If we have visited bad memory, the thread is to be terminated.

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"?

Use a semaphore **wait_child_load** in caller thread. If the callee ends its load, it sema_up the semaphore to let caller know the success/failure status.

And the thread is linked in **child_list** and its state is written in **exit_status**. A different number is used to mark load failure to distinguish with kernel kill. So if the child thread is killed by kernel, we know that its load is successful but the its execution meets some difficulties(poor child).

If it is failed. We use semaphore **end_process** to enforce that its parent pointer is changed to NULL before it exits.

> 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?

Whether P waits C before C exits or after C exits, P can only read C's exit_status when C sets semaphore **end_process** to **1**.

When P terminated, the the parent pointers of running child processes are set to NULL. And the dead child is cleared by P.

The child process may call **process_exit** simultaneously, so a semaphore must be used to ensure one of these two conditions: its parent pointer is changed to NULL before it turns off interrupts to schedule **or** it died first and then P finds its death.

RATIONALE

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

Because the fast speed with assistance of MMU.

And the memory management only includes maintain page table and handle page fault. This way can concentrate related code.

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

The structure of **file_descriptor** is the minimal unit for a process to record a file. I can not compress its space any more.

And the operations in **process.c** are using **file_descriptor** structure. The interfaces for **syscall.c** are **fd** number or **struct file**. This builds a black box model of file processing in **process.c**.

The internal helpers are

```
/*
  helper function in process.c
  find child thread by tid
*/
static struct thread * find_child_thread(tid_t tid) {
    ...
}
/*
  helper function in process.c
  delete file_descriptor structure
*/
static void close_fd(file_descriptor *fd) {
    ...
}
/*
  helper function in process.c
  find file_descriptor structure
*/
static file_descriptor* find_fd(int fd) {
    ...
}
```

The interfaces for **syscall.c** are

```
/*
  interface for syscall.c
  For OPEN syscall to add file descriptor in process
*/
int process_add_file(struct file *f) {
  ...
}

/*
  interface for syscall.c
  For WRITE syscall to find file structure in process
*/

struct file* process_find_file(int fd) {
  ...
}
```

```
/*
  interface for syscall.c
  For CLOSE syscall to close file descriptor with specific fd
*/

bool process_close_file(int fd) {
  ...
}

/*
  interface for syscall.c
  For EXIT syscall to clear all file record and release memory resource
*/
void
process_clear_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 don't change it. But in multithread OS a same process number for thread group is needed.