Operating System project 2 : Program

SUSTech Operating System Project 2

Group member

11610310 Lu Ning

11610309 Gong ZeLin

Task 1: ARGUMENT PASSING

Data Structures

- list of pointer `int* argc`

  This pointer list stores the address of the arguments. Used in method `setup_stack`.

process.c

- Modify the `process_create` : Use `strtok_r` to extract the filename in the input string. Use the real name to create
- Modify the `setup_stack` : Initialize a stack and push the stack according to the **80x86 convention**. Here is an example: `/bin/ls -l foo bar`

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

Algorithms

Analysis

Process of creating a new process:

- First call the method `process_execute`

The key part of this task is to push the argument according to the **80x86 convention**. We implement this process in the `setup_stack` method.

```
/* in setup_stack */

// copy the command stirng (file_name)
char * copy = malloc(strlen(file_name)+1);
strlcpy (copy, file_name, strlen(file_name)+1);

// use the copied string to get argc
for (token = strtok_r (copy, " ", &save_ptr); token != NULL;
     token = strtok_r (NULL, " ", &save_ptr))
    argc++;

// initialize pointer list: int* argv
int *argv = calloc(argc,sizeof(int));

// push the argv value
for (token = strtok_r (file_name, " ", &save_ptr),i=0; token != NULL;
     token = strtok_r (NULL, " ", &save_ptr), i++)
{
    *esp -= strlen(token) + 1;
    memcpy(*esp,token,strlen(token) + 1);
    argv[i]=*esp;
}

// word align, world size is 4, every loop increase the stack pointer by 1 byte
while((uint32_t)*esp % 4 != 0)
{
    *esp-=sizeof(char);
    char x = 0;
    memcpy(*esp,&x,sizeof(char));
}

// push null pointer
int zero = 0;
*esp -= sizeof(int);
memcpy(*esp,&zero,sizeof(int));

// push argv[] pointer
for(i=argc-1;i>=0;i--)
{
```

```
    *esp-=sizeof(uint32_t);
    memcpy(*esp,&argv[i],sizeof(uint32_t));
}

// push the poniter which points to argv
uint32_t ptr = *esp;
*esp-=sizeof(uint32_t);
memcpy(*esp,&ptr,sizeof(uint32_t));

// push argc
*esp-=sizeof(int);
memcpy(*esp,&argc,sizeof(int));

// push return address which is 0
*esp-=sizeof(int);
memcpy(*esp,&zero,sizeof(int));

// do free
free(copy);
free(argv);
```

Synchronization

Make the creation of stack atomically in case of safety problem.

```
  /* Create the stack atomically */
  old_level = intr_disable ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  intr_set_level (old_level);
```

Rational

The algorithm is simple, and easy to debug. We just follow the description and push the arguments. After finishing using the char list, we free the space manually to save memory.

Task 2: Process Control System Call

Data Structure

There are significant changes in `thread.c` and the `struct thread`, also create a new struct `child_data`

```
// changes in struct thread

 /* exit status, every thread's initial exit status is INIT_EXIT_STATUS except for the kernel thread*/
int exit_status;

/* after child  load, pass sucess to parent->child_load_success */
bool child_load_success;

/* lock of load, when the child is loading and executing, lock its parent */
struct semaphore load_sema;

/* list of this process' children, list elem is struct child_data*/
struct list children;

/* parent of this process */
struct thread* parent;

/* the child this process is waiting, in pintos a process only wait one child at one time */
struct child_data * waiting_child;
```

```
// struct child_data

/* this struct is mainly used for wait. Some information is stored in this struct in case that the child exited before the parent wait for it.
This also implies a relation between processes. */

struct child_data {
      int tid;                        // tid of the child process
      struct list_elem child_elem;    // list elem for struct thread->children
      int exit_status;   /*store its exit status to pass it to its parent */

      /*store wheather the child process is exited, if exited the wait system call don't UP the wait_sema */
      bool is_exited;
     /* lock the parent when it is waiting for this child process */
      struct semaphore wait_sema;
};
```

```
16
```

The corresponding modifies are in the <thread.c>

The other big changes are in the `syscall.c`, where the system call handler and system call functions are implemented. And function `is_mapped_user_vaddr` check whether the pointer address is in the user space, which is used in system calls to check the bad address.

Algorithms

User Process

In `init_thread` , initialize `load_sema`, `children` list and set `exit_status` to **INIT_EXIT_STATUS** which is a predefined special int . If a thread is created but not load successfully, its `exit_status` is **INIT_EXIT_STATUS**, then the exit process will detect it and change the `exit_status` to -1.

In `thread_create`, except for creating a `struct thread`, we also need to create a `staruct child_data` for this process and its `parent` is the current thread. `tid` is equal to the thread `tid`. `is_exited` is false. Initialize `wait_sema` with resource 0. Then push the `child_elem` to the `children` list of its parent (current thread) .

In `process_execute` , this is the start point of a user process. After the new thread is being created, we need to lock its parent until the result of loading (in `child_load_success`) is got in method `start_process` . The reason to do so is that we need to get the right result after the load operation. If there is no semaphore, we don't know when to check the load result. And this implementation will prevent that the child is load quicker than its parent.

System call parameter get

Use `esp` to get the parameter in order. The in stack order is defined by the system. Here is an example:

```
1  // this is the system call function for user calling
2  int
3  write (int fd, const void *buffer, unsigned size)
4  {
5    return syscall3 (SYS_WRITE, fd, buffer, size);
6  }
7
8  /* stack push the parameters. Push order:
9  [arg2] [arg1] [arg0] [syscall_number] -> stack top
10 */
11 #define syscall3(NUMBER, ARG0, ARG1, ARG2)                    \
12         ({                                                    \
13           int retval;                                         \
14           asm volatile                                        \
```

```
            ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "    \
             "pushl %[number]; int $0x30; addl $16, %%esp"      \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER),                         \
                 [arg0] "r" (ARG0),                             \
                 [arg1] "r" (ARG1),                             \
                 [arg2] "r" (ARG2)                              \
               : "memory");                                     \
          retval;                                               \
        })
```

System call parameter check

We do a general check for the pointer :

- check the pointer address is null or not
- check the page_ptr of the process address is null or not.
- check the pointer address is in user space

Special check:

- check the parameter address is valid
- check some string's address

Other exceptions are processed in the `page_fault` handler.

When an invalid pointer is find, the process is forced to exit with exit_status = -1;

System call

- syscall_halt

  This method is easy, just call the `shutdown_power_off` provided by "devices"

- syscall_practice

  Need to add `SYS_PRACTICE` in the "usr/lib/syscall.h".

  The process is just fetch the parameter from the stack (`*((f->esp) + 1)`), add it by 1, and put ti into `f -> eax`

- syscall_exec

  call the method `process_execute` and the put return value into `f -> eax`

- syscall_exit

  In the basic of the raw method `thread_exit()`, we add some operation and generate new method `exit_p( exit_status )` . Basically these operation are related to the correct **wait**.

  - First store the `exit_status` in the `child_data` struct and set `is_exited` to be true. This operation solve the problem that the parent may start waiting after its child exited.
  - Another operation is to check whether the exiting process is waited by its parent. If so, UP the `wait_sema` to tell the parent he is exiting.
  - At the end, call `thread_exit`

- **syscall_wait**

  This part is the most complex part in this task.

  There are two cases :

  - Parent starts waiting before child exits
  - Parent starts waiting after child exits

  For the fist case, we need to hang up the parent and listen for the exit of child. We use the `wait_sema` to implement it.

  For the second case, we use the `child_data` to store the essential information before some child is exit. Parent can just scan the list `children` and find its waiting child's `child_data` to get the information.

Synchronization

Basically use semaphore to solve the load synchronization problem, which ensure the execute order.

Use semaphore to make wait correctly happens. Use atomically operation to the `exit_p` process.

Rational

We keep a children list for a process, so we can find the children more quickly than scanning the all process list. And use semaphores to solve the synchronization problem.

TASK 3 : File system

Data Structures

struct thread

- list of opened file `struct list opened_files` Keep an record for all files opened by a thread.

threads.h

- lock for file system `lock filesys_lock` Keep the Synchronization problem of all file operation.

syscall.h

- struct to keep file information `struct occupy_file''' Keep the file pointer, list element of` struct list opened_files ``` and file descriptor for thread.

Algorithms

1. Using a list in `struct thread` to record all files which is opened by a thread, implement a `search\_occupy\_file` method to search a specific file by its file descriptor.In this way each thread could only operating on files opened by itself, which implement a relatively independent filesystem for each thread.

2. Using a global file system lock to implement Synchronization for all file operations in `filesys.c`, by requiring a lock before every filesys function call and release the lock after the function return.
3. As we add new members in `struct thread`, we must pay attention on the initialization and destory of this new members, `struct list opened\_files` for every thread should be initiated in `init_thread` when the thread is newly created. Meanwhile, `lock filesys\_lock` also need an initialization but in function `thread\_init` at the very begining.
4. When a thread is killed or terminated, all the files opened by it must be closed and the `struct occupy_file''' need be destoried, this is put in` process_exit``` function and neer the default page directory destory part for more code readability.
5. Some of the file system call are using a file name to find the file pointer(like `create, open, remove`), which should consider NULL filename pointer or empty filename. Other file system which using a file descriptor(fd) to find opened files, should consider ilegal fd(NULL `struct occupy\_file`) and NULL file pointer in `struct occupy\_file`.

Synchronization

1. Using a global file system lock to implement Synchronization for all file operations, Without modifacation in `filesys.c`, we implement this Synchronization totally in `syscall.c`. Requiring the `filesys_lock` is restricted before any call for functions in `filesys.c`, and one must release `filesys\_lock` after the function return.
2. code example:

```
lock\_acquire(&filesys\_lock);
res = file\_length(file->file_ptr);
lock\_release(&filesys\_lock);
```

Rational

Using a list to handle all opened files for a thread is easy to implement. As a build-in struct, `struct list` has a lot default funtions such as `list\_push\_back()` or `list\_pop\_front()`, It can save lots of work.

Questions

1. 卢宁(11610310) did all of Task 1 & Task 2. 宫泽林(11610309) did all Task 3. Our pintos is far from perfect and have only passed 72/80 tests. There're still some problem in `rox-simple, rox-child, rox-multichild` tests, which means we still have some work to do in executable file write denying.
2. Safety problem: We use atomic operation to ensure the safety when creating a stack. But we haven't design some test for safety problem. The file name may be leak by other process.
3. The name conventions are consistent with pintos project, we also use some provided functions in pintos.
4. Some systemcall may be complicated because many things should be considered. We add comments on most of our codes so it's easy to read. And the report explain the main design idea of our project.
5. Some implementations may be difficult to read. But we added enough comments for it.
6. Yes, some simple/obvious declaration and operation we didn't add the comment.
7. Some system call may be duplicated.
8. No.

9. No.

Reference

We look the following doc or source code to get our design idea:

https://github.com/liziwl/operating_system_project2

Pintos doc form Xidian University of Electronic Technology