Project User Programs Design

Group 107

Name	Autograder Login	Email
Yuanxiu Wang	Student364	iamwangyuanxiu@berkeley.edu
Qirui Ruan	Student153	ruanqirui@berkeley.edu
Marvin Hsin	Student243	bdfb1997@berkeley.edu
Tianxiang Liu	student156	ltx@berkeley.edu

Argument Passing

Data Structures and Functions

```
pid_t process_execute(const char* file_name);
static void start_process(void* file_name_);
bool load(const char* file_name, void (**eip)(void), void** esp);
static bool setup_stack(void** esp);
```

Algorithms

There are mainly 2 steps at this stage. Firstly, we will read **argv** and **argc** from the file passed into **load** which is called by **start_process** in the thread created by **process_execute**. Push **argv** and **argc** arguments onto the stack inside **start_process** before calling **load**. All the file reading, instruction pointer setup will be done in **load**. Subsequently, **argv** and **argc** have to be pushed onto the stack set up by allocating stack pointer in **setup_stack**.

Additionally, it would also be good to change the naming of the thread created inside **process_execute** to the file name instead of including the arguments.

Synchronization

There is no synchronization problem at this stage.

Rationale

Basically, what we try to do here is simply extracting **argc** and **argv** from the file. And, we push them onto the stack as well as setting up **esp** and **eip** so that we can use them afterwards.

Process Control Syscalls

Data Structures and Functions

```
//Syscalls
static int practice (int i) //show the syscall interface
static void halt (void) //terminate Pintos
static void exit (int status) //terminate the cur user prog
static pid_t exec (const char *cmd_line) //run the passed-in e
xecutables
static int wait (pid_t pid) //wait for a child process pid(til
l it terminates)


//Existing modification
struct thread{
  struct wait_status own_ws; /* Own wait status */
  struct list_elem children; /* List element for children's wa
it status */
}
static void start_process (void *file_name_) // return TID_ERR
OR if load fail, sema_up(); to support synch load & thread_cre
ate
static void syscall_handler(struct intr_frame* f UNUSED) // ge
t arguments
```

```
pid_t process_execute (const char *file_name)// return TID_ERR
OR if load fail, sema_down(); to support synch load & thread_c
reate
int process_wait(pid_t child_pid UNUSED) //wait for the proces
s with pid to die
bool load(const char* file_name, void (**eip)(void), void** es
p) // modify to support sychronization

//New Addition
struct wait_status{
  pid_t pid;                  /* Process identifier. */
  int exit_code;              /* Own exit code. */
  int ref                     /* Reference count = 2, w/ lk */
  struct lock lk;             /* Lock */
  struct semaphore sema;      /* Semaphore = 1 */
  struct list_elem elem;      /* List element. */
}
struct synch_load {
  char *filename;             /* File name */
  struct semaphore sema;      /* Semaphore = 1 */
  bool loaded;                /* if loaded or not */
};
// array of syscall functions, find by indecies: syscall_i sys
calls[NSC];
typedef static void (*syscall_t)(uint32_t *eax, uint32_t *args
UNUSED)
static int NSC //index, number of syscalls
bool arg_valid(uint32_t argc uint32_t *argv) // check if args
are valid
```

Algorithms

1. Read syscalls' arguments with changes constructed in Argument Passing

2. Check if arguments are valid by calling **arg_valid**

```
bool arg_valid(uint32_t argc uint32_t *argv){ //Implementation
  if argc ==NULL or pagedir_get_page(argc, &argc) == NULL:
    return false;
  for each arg in argv: // bounded by argc
    if arg == NULL: //check if invalid memory access
        return false;
    if is_user_vaddr(&arg) == NULL or pagedir_get_page(arg, &a
rg) == NULL:
    //check if illegal pointers or invalid pointers
        return false;
  return true;
```

3. Call syscalls in **syscall_handler**

```
static void syscall_handler (struct intr_frame *f UNUSED){ //M
odification
  //note that uint32_t* args = ((uint32_t*)f->esp);
  if args == NULL or is_user_vaddr(&args) == NULL:
    f->eax = -1; // return the user program using *eax
    return;
  args++; //move the pointer to the actual arguments of syscal
l
  syscalls[args[0]]((uint32_t *) &(f->eax), args);
  //record the syscall w/ index and call the syscall
  return;
}
```

For New Syscalls:

1. **practice**

```
static int practice (int i){
  load uint32_t *argv
  if !arg_valid(1, &argv):
```

```
    exit;
  return argv[0]+1;
```

2. **halt**

```
static void halt (void){
  shutdown_power_off();
}
```

3. **exit**

```
static void exit (int status){
  load uint32_t *argv
  if !arg_valid(1, &argv):
    return -1;
  int status = argv[0];
  struct thread* t = thread_current();
  //checks if it has a parent waiting; if so, return the statu
s
  t->own_ws->exit_code = status;
  printf("%s: exit(%d)\n", t->pcb->name, t->own_ws->exit_cod
e);
  thread_exit();
  process_exit();
  return 0;
}
```

4. **exec**

```
static pid_t exec (const char *cmd_line){
  load uint32_t *argv
  if !arg_valid(1, &argv):
    return -1;
  char *file_name = cmd_line;
  pid_t excecuted = process_execute(file_name);
  if excecuted == TID_ERROR:
    return -1;
```

```
  else:
    return excecuted;
}
```

5. **wait**

```
static int wait (pid_t pid){
  load uint32_t *argv, uint32_t *eax
  if !arg_valid(1, &argv):
    return -1;
  pid_t child_pid = (pid_t) argv[0];
  int child_status = process_wait(child_pid);// the child's ex
it status
  return child_status;
}
int process_wait(pid_t child_pid UNUSED){
//Waits for process with PID child_pid to die and returns its
exit status.
  if exp_killed()://if it was terminated by the kernel
    return -1;
  if child_pid == NULL://if child_pid is invalid
    return -1;
  if done_wait(child_pid)://if process_wait() has already been
successfully called
    return -1;
  struct thread* t = thread_current();
  for child in t->children:
    if child->own_ws->pid == child_pid://find the child
      sema_down(child->own_ws->pid->sema);//wait for child to
exit
      return child->own_ws->exit_code;
    else://if it was not a child of the calling process
      return -1;
```

```
}
```

Synchronization

We modify functions and structures to support synchronization with semaphores to control parents and children. When the parent waits, we call **sema_up** in the case that one of its child has died, and **sema_down** otherwise. We use **struct synch_load** to record and synchronize the actions done in **process_exec**; we call **sema_down** when the parent thread waits for the child to complete, and **sema_up** when synch_load→loaded is true; return correct value to syscall_handler.

Rationale

The main alternative is the put all new syscalls into one main functions and call the syscalls accordingly. The advantage of this alternative is the decrease of the repeated codes/calls to finish certain actions, e.g getting the arguments and verifying if they are valid. However, the drawback of this alternative is huge: it could easily catch an error or mess up the synchronization throughout the process.

File Operation Syscalls

Data Structures and Functions

```
static void syscall_handler(struct intr_frame* f UNUSED);

bool create (const char *file, unsigned initial_size);
bool remove (const char *file);

int open (const char *file);
struct fnode{
  struct list_elem elem;
  int fd;
  struct file data;
};
```

```
int filesize (int fd);
int read (int fd, void *buffer, unsigned size);
int write (int fd, const void *buffer, unsigned size);
void seek (int fd, unsigned position);
unsigned tell(int fd);
void close (int fd);
```

Algorithms

The modifications to the **syscall_handler** check the current status of the program just when entering syscall by using if statements and constants inside **syscall-nr.h.** Then, it loads the arguments accordingly right above **esp** to call the appropriate functions.

create calls the function **filesys_create** in **filesys.c** to create a file with the given size, and returns true if **filesys_create** returns true, and false otherwise.

remove calls the function **filesys_remove** in filesys.c to remove the file with the given name. Return true if **filesys_remove** returns true, and false otherwise.

open involves 2 parts.

First, **open** calls the function **filesys_open** in **filesys.c** to open the file with the given name. The **filesys_open** function normally returns the file struct pointer to the open file if the action succeeds. If the pointer is NULL, that means the file failed to open, which means the **open** function should return -1.
Because it is necessary to save the list of file structs being opened, a list of file structs should be saved by creating a simple struct called **fnode**.
By creating this list struct and numbering off the file structs using a list, it would be possible to fit a growing number of file structs inside them.
If the struct is not initialized, initialize the list and number the first file struct starting with fd = 3, and increase the number each time a new file struct is pushed onto the list.
After pushing the file struct onto the list, return the fd in the node struct of the file.

filesize searches for the file struct given its fd inside the list of files and calls **file_length** inside file.c. This returns the length of the given file struct. If the file is not found inside the list, return 0.

read searches for the file struct given its fd inside the list of files and calls **file_read** inside **file.c**, returning the number of bytes **file_read** had read. If a file is not found inside the list, return -1. If the **fd** is equal to 1, use a while loop, getc and memcpy to copy the stdin to the memory location indicated by the buffer.

write searches for the file struct given its fd inside the list of files and calls **file_write** inside file.c, returning the number of bytes for **file_write**. If a file is not found inside the list, return -1. If the **fd** is equal to 0, use a while loop and putc to write to stdout from a buffer.

seek searches for the file struct given its fd inside the list of files and calls **file_seek** inside file.c. If the file is not found, simply return.

tell searches for the file struct given its fd inside the list of files and calls **file_tell** inside **file.c**. If the file is not found, return -1.

close searches for the file struct given its fd inside the list of files and calls **file_close** inside **file.c**. Then, it removes the associated node inside the list of files using **list_remove** in **list.c**, and frees the specific node struct in the list. If the file is not found, simply return.

Synchronization

Synchronization is delt with a simple global lock around the file functions, because file operations can be atomic within multiple threads in this stage.
In other words, the detailed handling of file operations is delt in detail in later projects.
While inside any of the file operation functions, just put a global lock around the functions called by the **syscall_handler**.

Rationale

The main alternatives here is how to store and allocate space for the list of file structs.
Another alternative would be allocating arrays. This makes indexing very easy because the file structs would have their pointers stored next to each other in memory. However, a problem comes with the fact that the number of files being opened is continuously growing, and arrays don't deal very well with lists without limits.

Floating Point Operations

Data Structures and Functions

We store FPU state in each thread's interrupt and switch thread frames.

```
interrupt.h
/* Interrupt stack frame. */
struct intr_frame {
  /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
  uint8_t fpustack[108]; /* FPU state for interrupts */
  uint32_t edi;       /* Saved EDI. */
  uint32_t esi;       /* Saved ESI. */
  uint32_t ebp;       /* Saved EBP. */
  ...
}
```

```
switch.h
/* switch_thread()'s stack frame. */
struct switch_threads_frame {
  uint8_t fpustack[108]; /* FPU state for thread context switc
hing */
  uint32_t edi;        /*  0: Saved %edi. */
```

```
  uint32_t esi;         /*  4: Saved %esi. */
  uint32_t ebp;         /*  8: Saved %ebp. */
  uint32_t ebx;         /* 12: Saved %ebx. */
  void (*eip)(void);    /* 16: Return address. */
  struct thread* cur;   /* 20: switch_threads()'s CUR argument.
*/
  struct thread* next; /* 24: switch_threads()'s NEXT argumen
t. */
};
```

Algorithms

We should be able to implement these changes to assembly, in mainly start.S and switch.S.

- When the OS starts up or when a new thread is created, we use **FINIT/ FNINT** instructions to clean the registers
- During interrupts or switching threads, we can use **FSAVE/ FNSAVE** instructions to save the FPU state to stack, then use **FRSTOR** instruction to restore the data.

Synchronization

There is only one FPU, so all threads must share it. For any newly-created threads or processes, we want to be able to assume that the FPU is completely clean.
Therefore, we will save and restore Floating-point registers on the stack during these scenarios:

- OS startup
- thread/process creation
- thread/context switching
  - Interrupts
  - Syscalls

We do not need to care about the individual components, so we just save the entire FPU save state.

Rationale

Each thread has it's own 108-byte floating point unit state. We add the data structure to the stack frames to allow the threads to store its FPU states during these scenarios.

Concept check

1. In the "**sc-bad-sp.c**" line 16, we can see that the program tries to set the stack pointer to the address **$.-(64*1024*1024)**, which is 64 mb below current address. When the syscall tries to navigate syscall number later, the invalid address will force the program to exit with code -1.
2. In the "**sc-bad-sp.c**" line 12, the program uses address **0xbffffffc** which is valid as a stack pointer, but it is close to the base **0xc0000000** in pintos. Next, the syscall number **$0x30** is pushed onto the stack and make the stack pointer going above the base frame, which accesses the memory to the kernel. Therefore, it should exit with exit code -1.
3. The test suite is missing the synchronization tests. For example, If a process crashes and exit, we should properly release locks that have been allocated. Otherwise, other processes may still stay wait in process and never wake up.