Project User Programs Design

Group 43

Name	Autograder Login	Email
Arul Loomba	student483	arulloomba@berkeley.edu
Michael Hom	student324	michaelhom@berkeley.edu
Dmytro Krukovskyi	student387	dkruk@berkeley.edu
David Zechowy	student496	davidzechowy@berkeley.edu

Argument Passing

Data Structures and Functions

```C/C++
int argc;
char* argv[];
/* Hold the values taken from the filename argument. Will be created in the
load function and be used after we create stack*/
```

Algorithms

```C/C++
*((int *)(*esp) + 1) = 1 --> used to set 1 byte above esp to value 1
(we can apply it to setting bytes above esp to argc and argv)
```

- Parse the string argument (`const char* file_name`) which is in the form of `program-name arg1 argc2` based on the spaces using strtok() and strtok_r() and assign argc to number of arguments, and argv to the words.
- Modify address pointers above stack pointer to argc and argv values
- Increment esp in a way that maintains 16-byte alignment, insert argc and argv above it

Synchronization

- Currently n/a

Rationale

- Start-process → load
- During the load function, we will create argc and argv. We also confirm that our stack is setup using !setup_stack(esp). If the stack is properly set up, using the new base/esp, we can push argc and argv above the base pointer, because only at that point have we padded. Then we can push argc, argv, and our return address onto the stack, ensuring that arguments are properly passed and programs don't' crash when trying to access argv[0], etc.
- According to the Pintos source code we can only push argc and argv onto the stack once we have padded it appropriately for 16 bytes divisible; thus once we get argc and argv, we should only push them onto the stack once its been appropriately initialized in load.

Process Control Syscalls

Data Structures and Functions

```C/C++
//for wait function

// might need data structure to hold if the child process exited normally or
terminated by the kernel even when the process is freed
struct child_list_elem
  {
    int childPid;
       int status;
    struct list_elem elem;
};
//because it needs to be flexible to add or remove children

struct process {
  /* Owned by process.c. */
  uint32_t* pagedir;          /* Page directory. */
  char process_name[16];      /* Name of the main thread */
  struct thread* main_thread; /* Pointer to main thread */
  struct list child_list;
};
//very simple to implement initialization functions of new pintos list
void init_child(child_list* c_list);
child_list* find_child(child_list* c_list, int Pid);
child_list* add_child(child_list* c_list, int Pid, int status);
int remove_word(child_list* c_list, int Pid);
```

Algorithms

```C/C++
//Checkpoint 1
/*practice
use is_user_vaddr() to check if user pointer is valid
We will update the integer argument, which is passed in the args[1], by adding
1 and placing the value into the eax to be returned.
```

```
Edge case: add one to 2^31 in a 32 bit system which is max of the integer
limit. - exception_init - a call to page fault to address issue - #OF Overflow
Exception" in userprog/exception.c


Add all syscalls to switch statement in syscall_handler in userprog/syscall.c
*//Checkpoint 2
halt
// call shutdown_power_off
exec
// takes cmd_line of user- verifies it is doesn't call any kernel stuff
// combination of fork and exec of UNIX
// needs to create a new process to run the cmd_line
// Requires syncing the functions from parent and child processes
wait
// needs to update check child_list to remove children that are called once,
// ensure children are not adopted and only wait for direct children
// update the fork function to accommodate for addition of new children by
updating the new data structure
// call sema-down for the parent process to wait and sema-up to have the child
wake up the parent process
//Final
fork
// should be simpler to implement the fork after creating exec and wait
function as both exec and wait contains portions of what fork achieves
// also need to map a virtual address to physical memory as stated in spec
```

Synchronization

List all resources that are shared across threads and processes. For each resource, explain how it is accessed (e.g. from an interrupt context) and describe your strategy to ensure it is shared and modified safely (i.e. no race conditions, deadlocks).

- All functions need locks to ensure the security when working in kernel mode
- When the process structure is edited, locks can ensure that there will be no race conditions or deadlocks.
- Need synchronization for exec to “check if child process has completed its executable before returning” from spec
- Wait needs synchronization because you need to put parent to sleep requiring locks to ensure the transfer between parent and child threads; we can use a semaphore to put the parent to sleep and wait for the child to exit
    - Semaphore needs to be available to both the parent and the direct child
- When forking, lock the child_list to prevent race conditions

Rationale

Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

```
For all pointers - safety read + write memory
Check for invalid, null, and illegal pointers - from spec

//Checkpoint 1
practice
    -   Simple to implement since we are building off of what we have for exit
    -   Very little coding required
    -   time/space complexity very small
    -   Easy to extend design to accommodate
//Checkpoint 2
halt
    -   Simple implementation as we just need to call and check the user inputs for
        calling the shutdown_power_off function
    -   Very little coding required
    -   time/space complexity very small
    -   Easy to extend design to accommodate
exec
    -   Moderate amount of coding and the design is easy to understand
    -   The time and space complexity is not too great, and it should be easy to extend
        the design to add more features.
wait
    -   Creating a pintos list to update the status of children even after they have
        exited allows for the flexibility to add and remove many child processes
    -   We should store the child_list in each instance of the parent process to allow
        the parent to call wait wherever they want
    -   Moderate amount of coding and the design is easy to understand
    -   The time and space complexity is not too great, and it should be easy to extend
        the design to add more features.
//Final
Fork
    -   From spec- build off of the implementation of exec and wait
    -   Given that the exec and wait functions are implemented, the following
        implementations should be simpler
    -   Time/space complexity is small
    -   Tougher to extend the design if it has to rely on the implementation of exec
        and wait
```

File Operation Syscalls

Data Structures and Functions

```C/C++
deciding between two different structs for the file descriptor table
/* Defined at the process.h file */
#define MAX_FDS = 1000;
/* Pintos List Implementation of File Description Table*/
/* struct file_descriptor {
       int fid;
       struct FILE* file;
       struct list_elem elem;
}; */

or

/* List of files where each index represents the file descriptor number. We
will be modifying the process struct, so we can access the list of files from
the PCB */
struct file *ofile[MAX_FDS];

/* Implement Locks (global lock for entire file operation syscalls). Implement
above syscall handler*/
lock_t* lock;
lock_init(lock);

/* Functions to Implement */
bool create (const char *file, unsigned initial_size)
bool remove (const char *file)
int open (const char *file)
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)
int tell(int fd)
void close (int fd)
```

Algorithms

- Create
    - Acquire global lock → Check if file is null, otherwise return false → call filesys_create wrapper function from filesys.h with appropriate arguments filled in → Release lock → Return true if success, false if fail
- Remove
    - Acquire global lock → Check if file is null, return false → call filesys_remove wrapper function from filesys.h with appropriate arguments filled in → Release lock → Return true if success, false if fail
- Open
    - Acquire global lock → Check if file is null, return false
    - Call file sys_open
        - If null return -1
    - Allocate file descriptor struct and set fd, starting at 2; Insert into process's file list; Release lock and return fd
- Filesize
    - Acquire global lock
    - Find file with fd argument
        - If no such fd, return -1
    - Call file_length(file), release lock, and return file size
- Read
    - Acquire global lock → If fd == 0, use input_getc() to read → Validate fd and buffer (return -1 if null, not fd, size too small, etc.) → Call file_read(), Release lock → Return bytes read, or -1 on error
- Write
    - Acquire global lock → If fd == 1 (stdout), use putbuf(buffer, size) → Validate fd and buffer (return -1 if null, not fd, size too small, etc.) → Call file_write(file, buffer, size) → Release lock → Return bytes written, or -1 on error
- Seek
    - Acquire global lock → Find file corresponding to fd (if it exists), Call file_seek(file, position) → Release lock
- Tell
    - Acquire global lock → Validate fd or return -1 (find the file) → Call file_tell(file) → Release lock → return current position
- Close
    - Acquire global lock → Find file corresponding to FD if it exists → Call file_close(file) → Remove fd from list and release memory → Release lock

Synchronization

- Pintos file system is not thread safe so we have to acquire and release a global lock for each critical section, which in this case is each file operation syscall
- At the beginning and end of each syscall we acquire and release the locks, ensuring that there are no race conditions, and ensures that our file operations work exactly how we want them to
- Concurrency analysis
    - Multiple threads can read and write from the files safely, given they share the same file descriptors

Rationale

- Pretty simple way to make the syscalls, serving as a wrapper class that provides basic functionality and reuses pintos file operations that have already been created
- Avoids race conditions, so its safe
- Uses a lock and struct for the file descriptor, so we can access each file safely
- Deciding on what type of list to access the file description table
    - Pintos List with the fid and FILE *
    - FILE * Array with a fixed size

Concept check

1. Take a look at the Project User Programs test suite in src/tests/userprog. Some of the test cases will intentionally provide invalid pointers as syscall arguments, in order to test whether your implementation safely handles the reading and writing of user process memory. Identify a test case that uses an invalid stack pointer (%esp) when making a syscall. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).
    - Sc-bad-sp.c
        - Invokes a system call with the stack pointer (%esp) set to a bad address. The process must be terminated with -1 exit code
        - Sets ESP to a value 64 MB below current instruction, which is way way below our valid user memory, and resides far outside it. Since this is an invalid stack pointer, we should exit with a code -1, otherwise we fail → the kernel cannot access this address, so the moment it tries it will result in a exit(-1)
        - Line 16, we call asm volatile and set the %esp to an incorrect address.
        - Line 17, we fail if we get anything besides exit(-1)
        - As said in the description of the test, this test is intended to cause an exit(-1)
2. Please identify a test case that uses a valid stack pointer when making a syscall, but the stack pointer is too close to a page boundary leading to some of the syscall arguments being located in invalid memory. Provide the name of the test and explain how the test works. Your explanation should be very specific (i.e. use line numbers and the actual names of variables when explaining the test case).
    - Sc-bad-arg.c
        - Passes $0xbffffffc as the ESP
        - Sticks system call sys_exit at very top of stack, and invokes the syscall with the ESP set to the address given above. Because the argument to the system call would be above the user address space, we should exit with -1 exit code
    - In line 12 of the test we call asm volatile and movl $0xbffffffc to %%esp
    - In line 13 we ensure that the test fails if we didn't return with exit -1

3. Identify one part of the project requirements which is not fully tested by the existing test suite. Provide the name of the test and explain how the test works. Explain what kind of test needs to be added to the test suite, in order to provide coverage for that part of the project.
    - We do not effectively test all the file system sycalls. While we do test open, read, and write, we do not test close, filesize, seek, or tell
        - To test all of these, you would need quite a number of tests that close a normal file, an invalid file, one that doesn't exist, and more
        - You would also have to ensure in the tests that the functions return the right value, for example, filesize should return the correct file size or -1 if the FD is not valid or does not lead to a valid file
    - We can either create one test that tests all parts of these syscalls, or separate tests for close, filesize, seek, and tell, that each check the behavior and results of the syscalls, and ensure that all edgecases pass when using these syscalls.