PROJECT 2: USERPROG DESIGN DOCUMENT

Andrew Adel <andrewadel2013@gmail.com>..... 17
Hamza Hassan <hamzahassan835@gmail.com>..... 26
Islam Yousry <islamyousry16@gmail.com> 14
Mahmoud Manfy <mahmoud.manfy159@gmail.com>..... 57

ARGUMENT PASSING

DATA STRUCTURES

---

thread.h

```
struct thread
{
   struct list child_processe_list;
   struct thread* parent_thread;
   bool is_child_creation_success;
   int child_status;
   struct file* executable_file;
   struct semaphore wait_child_sema;
   struct semaphore parent_child_sync_sema;
   struct list_elem child_elem;
};
```

Child_process_list: list of all child processes.
Parent_thread: pointer to the parent process.
Is_child_creation_success: check if the child process created successfully or not.
Child_status: when the parent is waiting on its child process, child_status is updated with the child's exit code.
Executable_file: the current file that the process executes.
Wait_child_sema: the parent uses it to block itself until the child process finishes its execution.
Parent_child_sync_sema: the parent uses it to block itself until the child process is loaded.
Child_elem: struct for the list of the process’s children.

ALGORITHMS

Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order?

- When the *setup_stack* method returns true in *load* function inside *process.c* file, we then call our *split* function which takes the entire file_name as an argument and starts to parse the file_name and fill the stack as required.
  split(char file_name , void** esp);*
- During tokenizing the *file_name* with *strtok_r()*, we push each argument in the stack(so they are pushed from left to right)
- When we push the argument *i*, we store at arg_add[i] the starting address in the stack at which we store this argument(so arg_add[i] contains an integer representing the address in the stack at which the argument is stored)
- After pushing the argument, we add alignments if needed and then push 4 bytes of zero.
- Next, we had to push the addresses of the arguments from right to left. This was an easy task thanks to the arg_add array we filled previously, so we iterate over the array from *right to left* and push each address in the stack.
- We then push a pointer to arg_add[0]. That's it, a pointer to the pointer to the address of the first argument in the stack !!
- Lastly, we push the number of arguments(args) and a NULL pointer as a return address.
- Needless to mention that we free the memory reserved by the arg_add at the end

How do you avoid overflowing the stack page?

- Honestly, the only check we make when we start pushing the arguments in the stack is if the number of the arguments is less than a certain threshold(we defined it as 30 in our implementation). However, if the user enters the arguments in such a way that each argument is very long, this may lead to troubles.

RATIONALE

Why does Pintos implement strtok_r() but not strtok()?

- Strok_r() is an updated version of strok() which is safer.
- The reason for that is that strok() uses a global variable to keep track of the string position. So using strok() in multiple strings simultaneously may lead to race conditions!

In Pintos, the kernel separates commands into an executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

- Allowing the shell to separate the commands will reduce a lot of overhead because the shell may validate if the command and the arguments are valid before passing it to the kernel.

- Allowing the shell to separate the commands will provide a layer of abstraction to the code because the shell is a user program, so the validation and separation are made in the user side rather than the kernel's.

SYSTEM CALLS

DATA STRUCTURES

In thread.h

```
struct open_file{
   int fd;
   struct file* ptr;
   struct list_elem elem;
};

struct thread
{
   struct list open_file_list;
   int fd_last;
};
```

In syscall.c

```
static struct lock files_sys_lock;
```

Struct open_file:
fd: file descriptor of the file that is held by the struct.
ptr: pointer to the file.
elem: to put the struct into a list.

Open_file_list: is a list of opened files
fd_last: is incremented when each element is added to open_file_list.

Describe how file descriptors are associated with open files.

- When an *open* syscall is called, we create a struct open_file and increment fd_last for the process with one and assign this value to fd which is inside the struct and push the whole struct in the open_file_list.

Are file descriptors unique within the entire OS or just within a single process?

- File descriptors unique just within a single process.

ALGORITHMS

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

- Firstly, we pop fd, buffer and size from the stack.
  Secondly, we check if the fd is 0 (read from stdin) or 1 (write to stdout)
  Thirdly, in case that fd has another value, we search for the file in the list of the process that we need to read from or write to it.
  Lastly, if the file is found inside the list, we read from or write inside the file else return -1.

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?

- At least 1 and at most 2.

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

- When the parent calls *wait(pid_t)*, we first check that there is a child with the pid_t given as an argument.
  Otherwise, we return -1.
- Each parent has a list of child processes that he is waiting on. This enables us to check the previous point with a single iteration
- Assuming the given pid_t is correct, the parent removes that child from the list and wakes it up.
- The parent will be blocked until the child finishes its execution.

- The parent has a variable called `child_status` which the child can update.
- Before the child terminates, it wakes up the blocked parent and updates the *child_status* variable (there is a connection between the parent and the child so the child can update the parent's field and updates it)

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.

- We tried as much as we could to put error-handling code in separate functions so that we avoid the code getting messy.
- We try to make validation code as early as possible to keep the code simple.

  Example:
  - We always check that the stack pointers are valid before storing them in variables and casting them.
  - We always check that pointers variables aren't null before using them.

SYNCHRONIZATION

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"?

- By is_child_creation_success flag, this flag is held by the parent process and when the load is terminated, the child process will update that flag.

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?**

- When p calls wait(c), we make sema_up(c->parent_child_sync_sema) and make sema_down(p->wait_child_sema). So the child process will start to execute and when it exits, it makes sema_up(p->wait_child_sema).

- **After C exits?**

- This scenario won't happen because when the parent spawns the child, the child will be blocked until the parent is waiting for that child or the parent is terminated.

- **How do you ensure that all resources are freed in each case?**

- When the process is terminated, it will iterate on child_processe_list to wake up all blocked children and iterate on open_file_list to close all opened files and it will free the memory of the struct holding the file.

- **How about when P terminates without waiting, before C exits?**

- When the parent terminates, he will wake up all its children and the children will execute and terminates also without any problem.

- **After C exits?**

- This scenario won't happen because when the parent spawns the child, the child will be blocked until the parent is waiting for that child or the parent is terminated.

- **Are there any special cases?**

- No, there aren't.

RATIONALE

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

- We didn't deal with user memory directly, we use functions provided by PintOs to interact with user memory from the kernel.

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

- **Advantages:**
  - We chose to use a simple implementation which didn't include a lot of corner cases so we weren't worried about issues such as race conditions.
- **Disadvantages:**
  - Only one process could read from a file at a time
  - If one process reads or writes in a file, no other files could be read or writte

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

- We didn't change it actually.