TDDB68
Lesson 3

Felipe Boeira

Contents

- Overview of lab 4
  - Program arguments
- Overview of lab 5
  - Wait System Call
  - Termination of ill-behaving user processes
  - Testing your implementation
- Overview of lab 6
  - File system
  - The readers-writers problem
  - Additional system calls
  - Testing your implementation

Lab 4: Command Line

- Handling program arguments
  - Currently, Pintos does not support arguments to programs
  - Remember *esp = PHYSBASE - 12? We needed - 12 (bytes) to compensate for the missing arguments. Remove it and add proper support
  - What is on the initial stack

Lab 4: Command Line

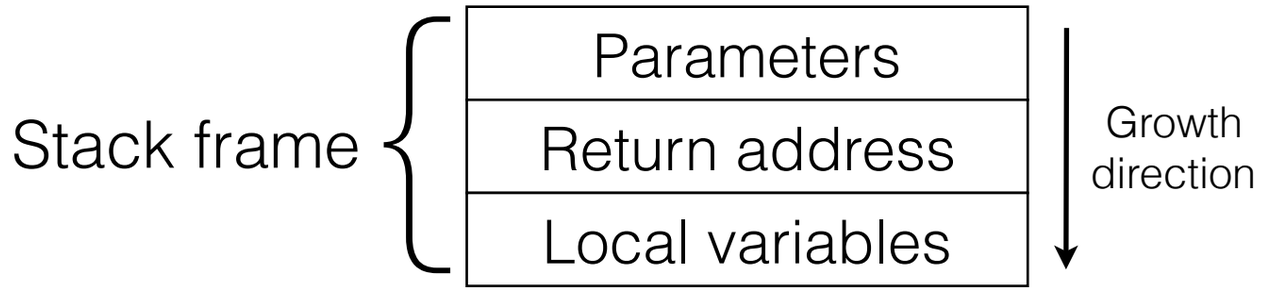
- To run a program with arguments in Pintos:
- Pintos (note the single quotes):
  `pintos --qemu --run 'insult -s 17'`
- System call:
  `exec("insult -s 17");`

Lab 4: Command Line

- Suppose we do this: `insult -s 17`
- The parameters of the main function of C programs are `int argc` and `char **argv`. In this example:
  - `argc` is `3`
  - `argv[0]` is `"insult\0"`
  - `argv[1]` is `"-s\0"`
  - `argv[2]` is `"17\0"`
  - `argv[argc]` is `NULL`
- Note that the length of the array is actually 4, but the last element is always `NULL` (required by the C standard)

Lab 4: Command Line

- Every time you do a function call, a stack frame is created:



- The function `main` is never really called, but the layout is the same

- The parameters and the return address of the stack frame are pushed onto the stack by Pintos (your code!)

Lab 4: Command Line

The string is stored on the stack

Address	Name	Data	Type
0xbfffffd	argv[2][...]	"17\0"	char[3]
0xbfffffa	argv[1][...]	"-s\0"	char[3]
0xbffffff3	argv[0][...]	"insult\0"	char[7]
...	word-align	unused	
0xbffffff0	word-align	unused	
0xbfffffec	argv[3]	NULL	char *
0xbfffffe8	argv[2]	0xbfffffd	char *
0xbfffffe4	argv[1]	0xbfffffa	char *
0xbfffffe0	argv[0]	0xbffffff3	char *
...			
0xbfffffc8	argv	0xbfffffe0	char **
0xbfffffc4	argc	3	int
0xbfffffc0	return address	unused	void (*) ()

Adjust the stack pointer so it is divisible by 4

Parameters

Local variables

The function never returns

Stack grows towards 0

Adding all of this to the stack is what Lab 4 is about!

Lab 4: Command Line

- Step 1: Change `*esp = PHYSBASE - 12;` to `*esp = PHYSBASE;`
- Step 2: Put the words on the stack, word alignment, the argv array, argc and the return address
- Hint: `start_process` calls `load`, which calls `setup_stack`.
- Hint: Take a look at `setup_stack(void **esp)`. Note that the argument is *a pointer to the stack pointer*! In other words, you must dereference twice to write to the stack, and dereference once to change the stack pointer itself!
- Hint: `setup_stack` is called by `load`, and after the call there is debug code for printing the stack. To print the stack, uncomment `/*#define STACK_DEBUG*/`

Lab 4: Command Line

- Hint: You get a string, such as `"insult -s 17\0"`, and you need to split it up in smaller parts.
  Use (in `lib/string.[c|h]`):
  `char *` `strtok_r(`**`char *`**`s, `**`const char *`**`delimiters, `**`char **`**`save_ptr)`

  - The following loop tokenises the string `s = "insult -s 17\0"`.

    ```
    for (token = strtok_r (s, " ", &save_ptr); token != NULL;
          token = strtok_r (NULL, " ", &save_ptr)) { ... }
    ```

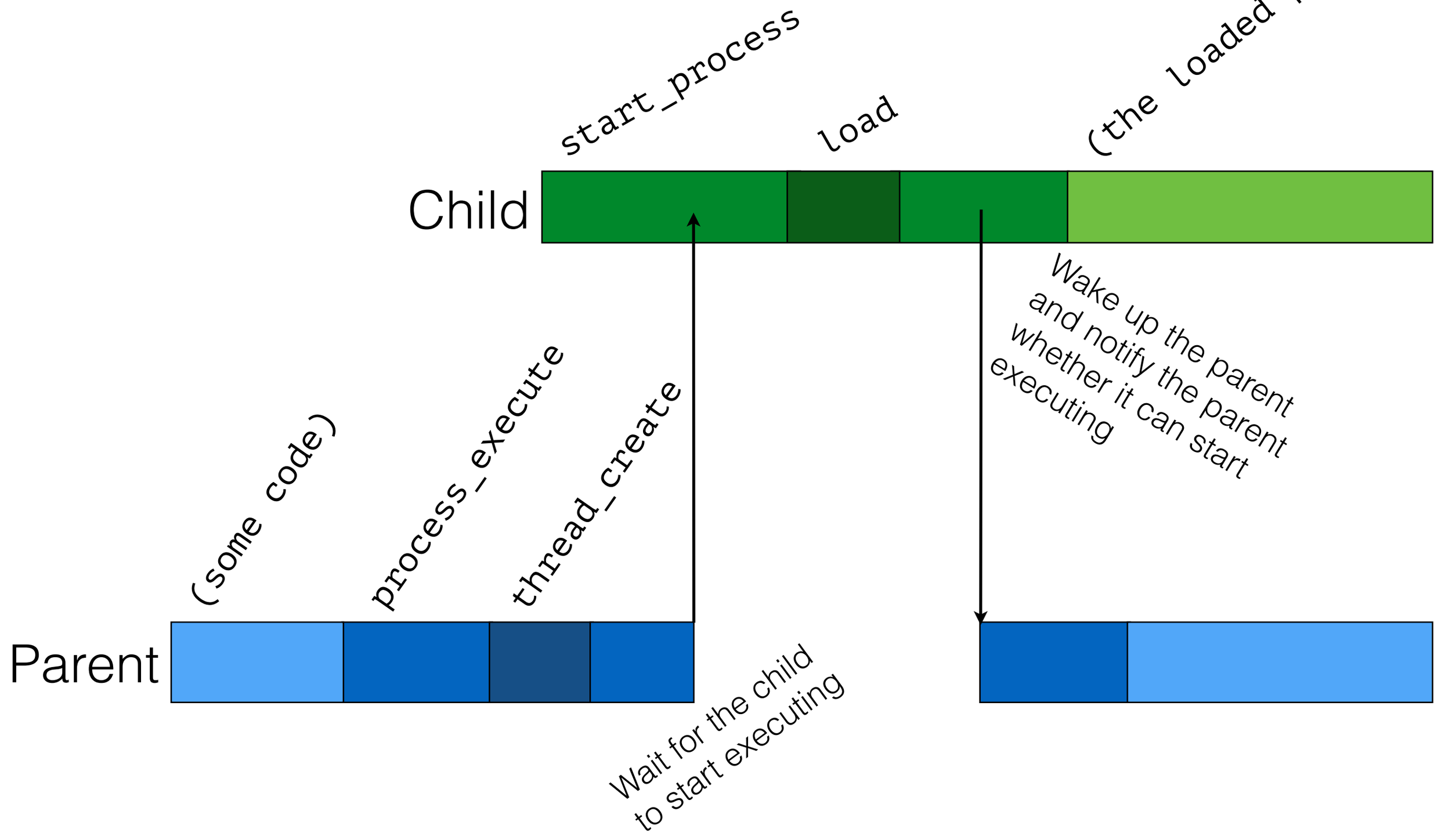
    Every iteration you get the next word, i.e. `"insult\0"`, `"-s\0"` and `"17\0"`. However, it does so destructively!

  - This is what the memory looks after the loop: `s = "insult\0-s\017\0"`

  - You need to save a pointer to every word

- Hint: Read Pintos documentation 3.5.1, it presents another stack print!

Previously… in TDDB68



Lab 5: Wait System Call

- `int wait (pid_t pid)`
- Scenarios:
  - Parent calls wait before the child terminates
  - Parent calls wait after the child terminates
  - Parent terminates before the child without calling wait
  - Parent terminates after the child without calling wait
- In each of these scenarios, your code must work and shared resources must be released by whoever terminates last
- Remember that a process can have several children!

Lab 5: Wait System Call

- The exit status is only available until the first wait call by the parent. If the parent waits twice on the same child, then the second wait returns -1

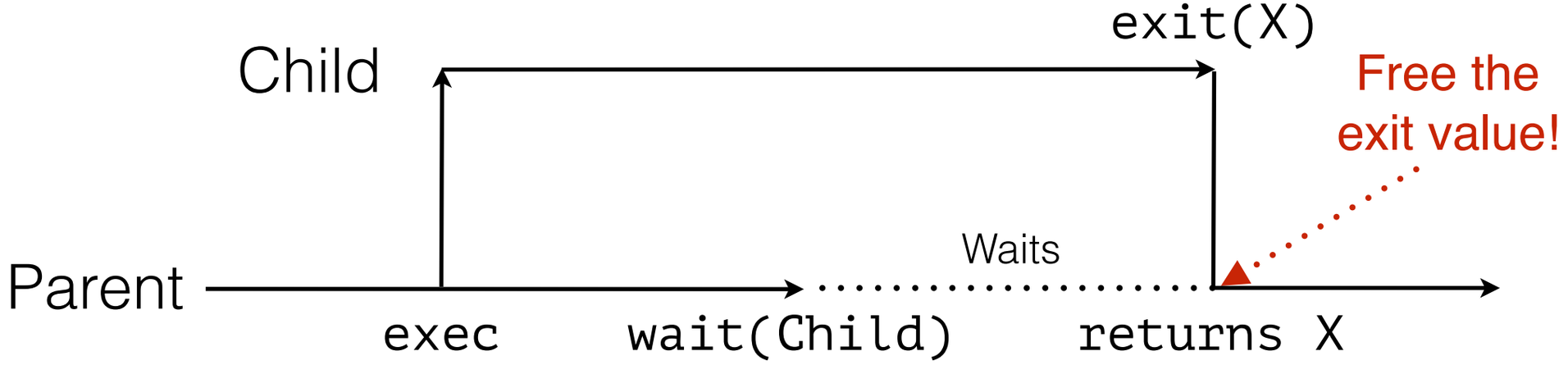
- Hint: Since the exit status has to be available even after the child terminates, then the exit status can be stored in dynamically allocated memory

- Hint: A common approach is to have a struct of data for every parent-child pair. The struct contains, amongst other things, the exit status

- As we noted earlier, this struct must be freed by whoever terminates last, i.e. either the parent or the child

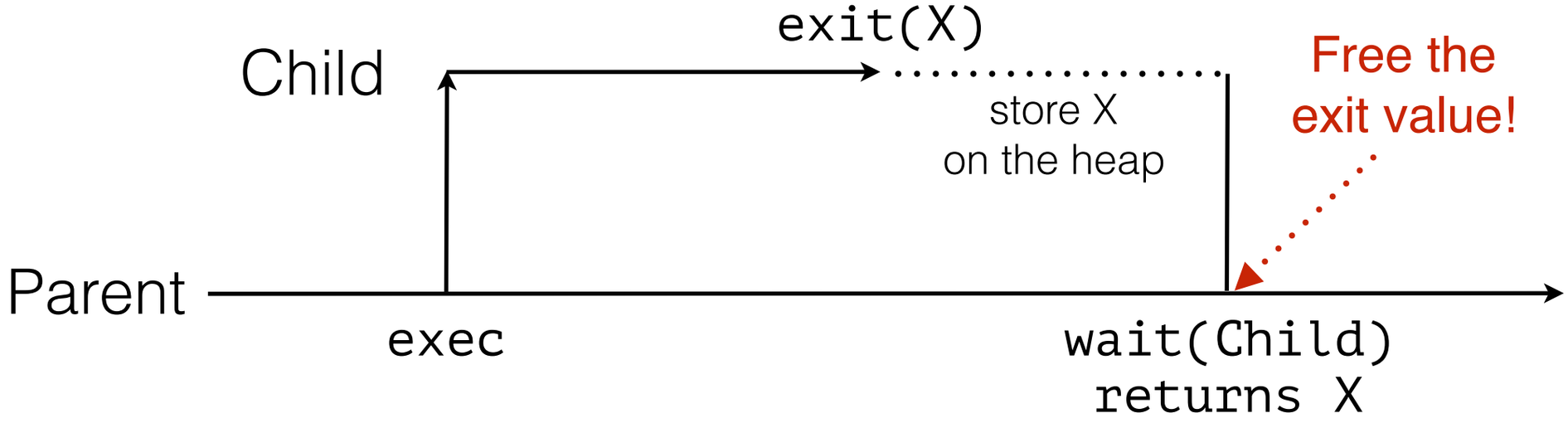
Lab 5: Wait System Call

- Scenario: parent waits for child to exit
- Important! Busy waiting is NOT allowed!



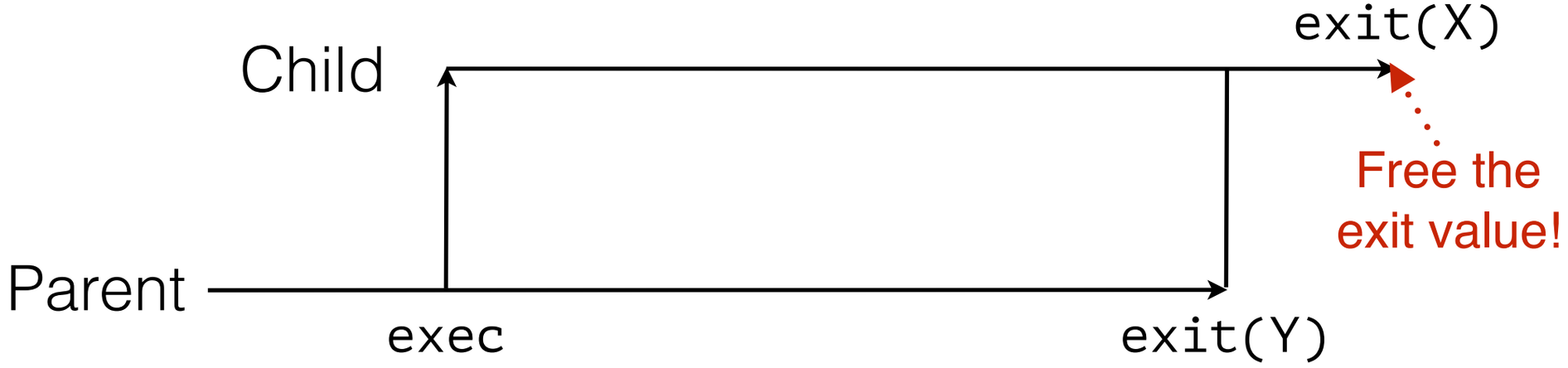
Lab 5: Wait System Call

- Scenario: child exits before the parent, and then the parent calls wait



Lab 5: Wait System Call

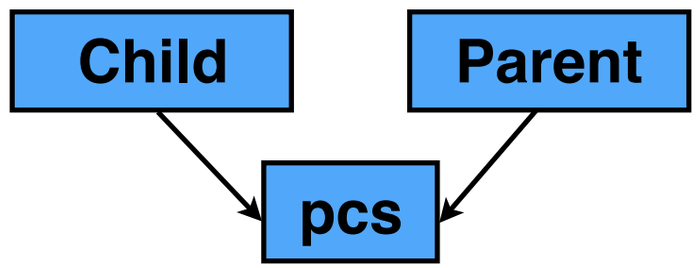
- Scenario: parent never waits for the child and exits before it



Lab 5: Wait System Call

- How to detect who is the last to exit?
- Hint: Keep a counter, together with the exit status, with the initial value 2. When the parent and child exit, decrement it by 1. The value 0 represents that both the parent and the child has exited, and whoever decrements it to 0 should free the memory
- Hint: Protect the counter with synchronisation! Pintos might leak memory otherwise!
-

```
struct parent_child {
  int exit_status;
  int alive_count;
  /* ... whatever else you need ... */
};
```



Lab 5: Input Validation

- Argument paranoia: nothing the user processes does should be able to crash Pintos
- For example: `create((char*) 1, 1);` is an invalid call since address 1 is in kernel space. Furthermore, it is not a string!
- Another example: `read(STDIN_FILENO, 0xc0000000, 666);` writes data to kernel space (will likely overwrite something important)
- Hence, ALL pointers from the user processes to the kernel must be checked! Including the stack pointer, strings, and buffers

Lab 5: Input Validation

How to validate a pointer:

- A valid pointer from a user process is:
  - below `PHYS_SPACE` in virtual memory (not kernel memory)
  - associated with a page in the page table for the process (use `pagedir_get_page`)
- If some pointer is not valid, then terminate the process! The exit status is then -1
- If you were to dereference an invalid pointer that is below `PHYS_SPACE`, then you get an error. It is possible to take care of the problem when the error occurs, but it is more challenging (although, it is faster: read the documentation if you want to attempt this)
- Hint: Read Pintos documentation 3.1.5

Lab 5: Input Validation

- Suppose a process calls `create((char *) PHYS_BASE - 12345, 17);`

- The function `filesys_create` does not check the string, and the string is not `NULL`. The call will likely crash Pintos!

- Hint: You must check that the `char *` is a string by iterating over *every character* of it, and check that the pointer to it is a valid pointer

- Hint: In other words, for strings, you must check every character until you find a '\0'. Remember to first check the pointer, then read it! (You might get an error otherwise)

Lab 5: Input Validation

- Suppose a process calls `write(1, malloc(1), 1000000);` Obviously, the arguments are invalid since the size of the buffer is 1, but the process claims it is much bigger!

- If we were to read from the buffer, then we would get an error. It is not sufficient to only test the pointer, we must also check its size!

- Hint: We must check that every possible pointer to the buffer is valid. In other words, we must test `10000000` pointers (at most, if you want to optimise it then you can compute where the page boundaries are, and check those)

- Hint: In contrast to strings, the size of the buffer is given and we do NOT have to search for a '\0'

Lab 5: Input Validation

- The user process can modify its own stack pointer:
  `asm volatile (”movl $0x0, %esp; int $0x30” ::: );`

- So you have to check the stack pointer if you want to read what it points to. If you increment the stack pointer, then you have to redo the check!

- In other words, you have to check the stack pointer before reading the system call number, before reading the first argument, and so on

Testing

- When you have finished, then you can test your implementation with: `make check`
- The following tests are for Lab 1: `halt`, `exit`, `create-normal`, `create-empty`, `create-null`, `create-long`, `create-exists`, `create-bound`, `open-normal`, `open-missing`, `open-boundary`, `open-empty`, `open-twice`, `close-normal`, `close-stdin`, `close-stdout`, `close-bad-fd`, `read-boundary`, `read-zero`, `read-stdout`, `read-bad-fd`, `write-normal`, `write-boundary`, `write-zero`, `write-stdin`, `write-bad-fd`
- Most of the `exec-*` and `wait-*` tests (and Lab 1) should pass when you have finished
- Hint: You can run a single test (from `userprog/build`) with: `make tests/userprog/halt.result`

Lab 6: Overview

- Synchronising the file system in Pintos
- The readers-writers problem
- Additional system calls (seek, tell, filesize, remove)
- Testing your implementation

Lab 6: File system

- `devices/disk.[h|c]` - Low-level operations on the drive (shared and already synchronised)
- `filesys/free-map.[h|c]` - Operations on the map of free disk sectors (shared)
- `filesys/inode.[h|c]` - Operations on inodes, which represents an individual file. When you write/read data to/from an inode then you modify the actual physical file (shared)
- `filesys/filesys.[h|c]` - Operations on the file system, such as create, open, close, remove, and so on (shared)
- `filesys/directory.[h|c]` - Operations on directories (partially shared)
- `filesys/file.[h|c]` - A file object that contains an inode and things like a seek position. Every process has its own object (not shared)

Lab 6: File system

- Your assignment is to *synchronise* the files that contains data which is *shared* between multiple processes and are *not already synchronised*

- Use locks and semaphores!

Lab 6: Readers-writers

- It is easy to synchronise the file system by only using one lock everywhere, but that leads to extremely poor performance
- We have these requirements:
  - Several readers are able to read from the same file at the same time
  - Only one writer is able to a specific file at the same time
  - Several writers are able to write to *different* files at the same time
  - When a process is reading from a file, then no process can write to the file at the same time
  - When a process is writing to a file, then no process can read from the file at the same time
- Hint: There is at most 1 inode per physical file

Lab 6: Readers-writers

- The readers-writers problem is how to achieve the aforementioned requirements. There are several solutions with different properties

- Hint: Wikipedia has a few algorithms
  `https://en.wikipedia.org/wiki/Readers-writers_problem`

- Hint: The solution with readers-preference is easy to implement, but might starve writers

Lab 6: System calls

- In addition to synchronising the file system, you also have to implement these system calls:
  - `void seek (int fd, unsigned position)`:
    Sets the current seek position of the file
  - `unsigned tell (int fd)`:
    Gets the seek position of the file
  - `int filesize (int fd)`:
    Returns the file size of the file
  - `bool remove (const char *file_name)`:
    Removes the file. Hint: The removal of the file is delayed until it is not opened by any process (make sure that this counter is synchronised)
- Hint: Use built-in functions
- Hint: Check for invalid arguments!

Lab 6: Hints

- Some questions that you should ask yourself: What can happen, in the worst case, if two processes try to...
  - Create and remove the same file at the same time?
  - Create and remove the same directory at the same time?
  - Open the same file at the same time?
  - Open and close the same file at the same time?
  - And so on!
- Many of these operations should, from each other perspective, be “atomic"
- This is NOT a complete list!

Lab 6: Testing

- To test your implementation, we provide the following user programs:
  - `pfs.c`
  - `pfs_reader.c`
  - `pfs_writer.c`
- The program pfs read and write to the same file concurrently:
  - 2 writers that repeatedly fill the file with an arbitrary letter
  - 3 readers that repeatedly read the file and checks that all the letters are the same
  - If the letters differ for the readers, then the test fails
- Run `pfs` *many* times and check the result each time! Inconsistencies due to lack of synchronisation may not happen every time

Questions?