

TDDDB68/TDDDD47

Lesson 2

Felipe Boeira

Based on previous slides from the course

Contents

- Synchronisation
- Interrupts
- Scheduler
- Busy Waiting
- Lab details
 - Lab 2
 - Lab 3
 - Lab 4

Synchronisation

- Multiple processes running concurrently
- What is synchronisation and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does about the following:
 1. fetch `i` from memory and store it in a register;
 2. increment the register by 1;
 3. store the value in the register in memory;
 4. if of interest, return the value in the register.
- Even an innocent looking line like `++i` consists of several instructions!

Synchronisation

- What can happen if two processes, p1 and p2 executes ++i at the same time?
- Both processes execute the sequence instructions, but they might be interleaved in any order. For example, the result of the following *should be* $i=+2$

	1. Fetch i from memory to a register	1. --	
	2. --	2. Fetch i from memory to a register	
	3. Increment the register by 1	3. Increment the register by 1	
p1	4. store the value in the register in memory;	4. --	p2
	5. --	5. store the value in the register in memory;	
	6. if of interest, return the value in the register.	6. if of interest, return the value in the register.	

- However, the result is $i=+1$

Synchronisation

Critical section

- A sequence of instructions, operating on shared resources, that should be executed by a *single* process without *interference*. Also known as *mutual exclusion*
- Concurrent accesses to shared resources can lead to unexpected behaviours. In Lab 0, what could happen if two processes called `prepend` to the same list, at the same time?
- Typical examples are data structures (e.g. lists), network connections, static and global variables, hard drives, files, and so on

Synchronisation

Some tools to solve the aforementioned problem are:

- Locks/Mutexes
- Semaphores
- Monitors (not needed in the labs)
- Conditions (not needed in the labs)

Synchronisation

Locks/Mutex (Mutual Exclusion)

- Two operations: acquire lock, and release lock
- The *same process* that acquires the lock must release it. For a *mutex*, another process may release it
- Ensures that at most one process executes the code enclosed between acquire and release lock
- In the previous example, p1 would have to finish executing `++i` before p2 could start
- Overzealous use of locks leads to poor utilisation of concurrency, i.e. try to not lock more than necessary
- Acquiring and releasing locks are relatively expensive, but you do not have to think about this during the labs

Synchronisation

Locks in Pintos:

- `#include "threads/synch.h"` // You need to include this
- `struct lock l;` // Defines and declares l as a lock
- `lock_init(&l);` // Initialises the lock l
- `lock_acquire(&l);` // Acquires the lock l
- `lock_release(&l);` // Releases the lock l

Synchronisation

Semaphores:

- A generalisation of locks
- Instead of `lock_acquire` and `lock_release`, we write `sema_down` and `sema_up`
- The process doing `sema_down` does not need to be the one doing `sema_up`
- You can set the number of processes that are allowed to execute the critical section concurrently
- For locks, this number is set to 1. You can set it to 2 or more, or even 0.
- When the number is 0, then the process doing `sema_down` immediately stops executing and waits for a `sema_up` (which another process can do). If another process `sema_up` before the first `sema_down`, then the process continues executing

Synchronisation

- Semaphores in Pintos:
- `#include "threads/synch.h"`
- `struct semaphore s;`
- `sema_init(&s, X);` // X is the initial value of how many processes are allowed to execute the section concurrently
- `sema_down(&s);`
- `sema_up(&s);`

Interrupts

- There are two types of interrupts in Pintos:
- Internal interrupts, which are caused by CPU instructions such as system calls and invalid memory accesses. The function `intr_disable()` does not disable internal interrupts
- External interrupts, which are caused by hardware devices such as the system timer, the keyboard, disks, and so on. The function `intr_disable()` postpones external interrupts

Scheduler

- The scheduler handles process scheduling. In short, it is the scheduler's job to decide when every process gets to execute
- In operating systems, processes get preempted so that another process can execute for a while
- When to preempt is based on timer interrupts, which are external interrupts
- In Pintos, the scheduler preempts the running process every 4th timer tick, and there are 100 ticks per second

Synchronisation Again

- The implementation of the synchronisation primitives in Pintos is based on disabling interrupts
- When external interrupts are disabled, the scheduler cannot preempt it and thus no other process can execute the critical section concurrently (Pintos do not support several cores or CPUs)
- This is a very crude way of doing synchronisation, but gets the job done in Pintos

Synchronisation Again

Beware of the following:

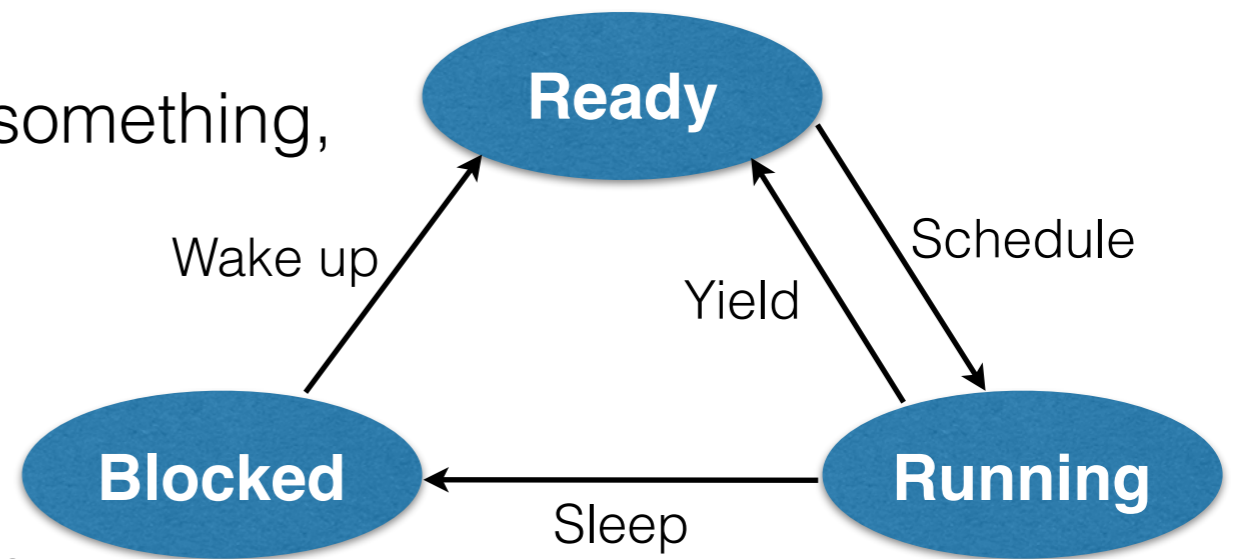
- Interrupts are NOT disabled until you release the lock (or the semaphore), only during the function calls (i.e. acquire and release). In other words, the critical section can still get preempted
- You can NOT use locks in the interrupt handler, read the Pintos documentation A.4.3 for more details on why
- Hint: You need to disable interrupts rather than using locks when the interrupt handler can cause race conditions (but use locks otherwise)

Busy Waiting

- Sometimes processes has wait for something, for example, to acquire a lock or semaphore, or simply wait a fixed number of ticks
- The assignment of Lab 2 is to improve the implementation of sleep
- The current implementation is this:

```
int64_t start = timer_ticks ();  
while (timer_elapsed (start) < ticks) {  
    thread_yield (); //Wasting CPU-cycles!  
}
```

- The process goes from running to ready to running over an over: there is little waiting going on
- This is unacceptable. Implement sleep in a much more efficient way!



Lab 2: Files and Functions

- devices/timer.[h|c]
 - **void** timer_init()
 - Remove busy-waiting from this function:
void timer_sleep(**int64_t** ticks)
 - **int64_t** timer_ticks()
 - **int64_t** timer_elapsed(**int64_t** then)

Lab 2: Hints

- Semaphores do not suffer from busy waiting, it only checks if there are any waiting processes when a `sema_up` occurs
- Recall that semaphores can be upped by *anyone*, and that if you set the initial value to 0 then the calling process stops executing immediately
- You need a list of sleeping processes, take a look at the list implementation in Pintos
- If the list is sorted, then it is possible to very quickly check if there are any processes that needs to be woken up

Lab 2: Testing

- Run the tests
 - alarm-single
 - alarm-multiple
 - alarm-simultaneous
 - alarm-zero
 - alarm-negative
- `make SIMULATOR=--qemu check`
- An individual test is run by a command like this:
`pintos --qemu -- run alarm-simultaneous`
- The tests will pass before you make any modifications because of the busy waiting implementation! However, the implementation is very inefficient!
- Remove any `printf`, the checks compare the output with solution sheets/text files: if there are any differences, then the check fails

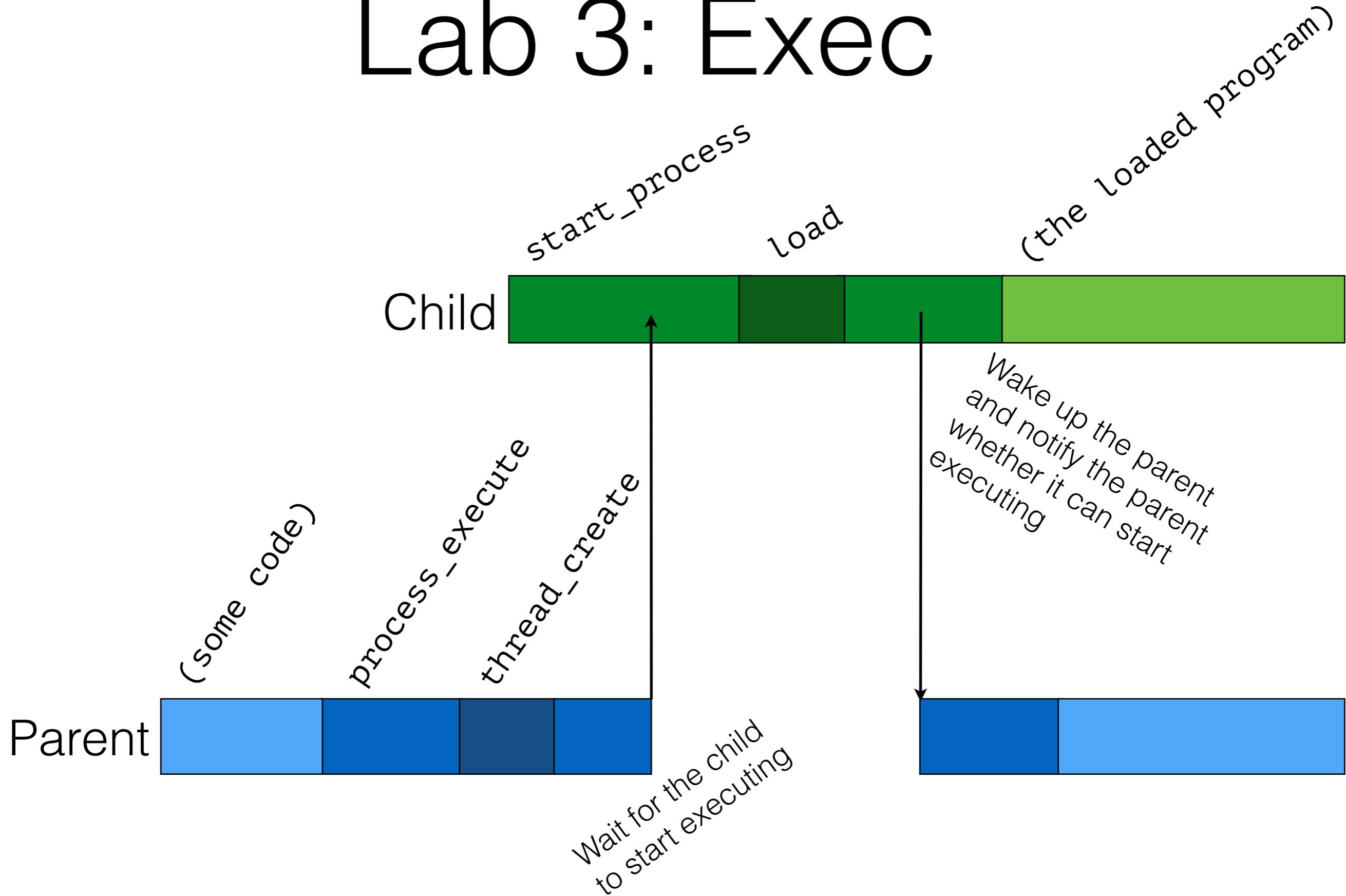
Lab 3: Overview

- Execution of several user processes
 - Exec: starts up a child process that executes the given file
 - Exit: terminates the process, frees allocated resources and saves the exit status somewhere
 - Alive counting to decide who should free dynamically allocated memory

Lab 3: Exec

- **pid_t** exec (**const char** *cmd_line)
- The first word of the string `cmd_line` is a file name, the rest are the arguments to the program. Spawn a new *child* process that loads the file and executes it. If the child process could load and start executing the file then return the process ID (PID) of the child, return -1 otherwise
- The current implementation of `exec` does not wait to see if the child could load the file/program
- Make `exec` wait for the new process to finish loading before returning the PID
- Hint: The “current” implementation is `process_execute`
- Hint: You can assume that TID and PID are the same thing in Pintos
- Hint: A process can have several children at the same time

Lab 3: Exec



Lab 3: Exec

- The following functions and lines of code are of interest.
- `tid_t process_execute (const char *file_name)`
- `tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy); // Inside process_execute`
- `tid_t thread_create (const char *name, int priority, thread_func *function, void *aux)`
- `static void start_process (void *fname)`
- `start_process` is the function that the new thread starts executing, and `aux` is the argument to said function

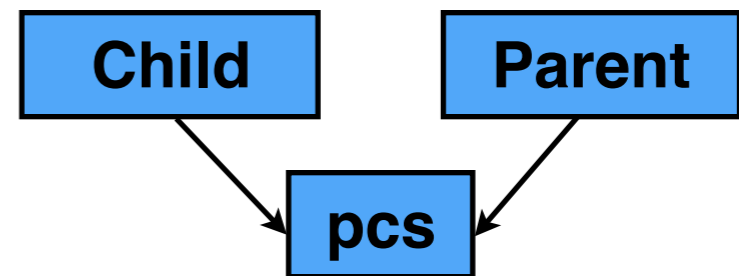
Lab 3: Exec

- Hint: The only place where `start_process` is "called" is in `process_execute`. Hence, you can extend the parameter of `start_process` (`fname`) and the argument to `thread_create` (`fn_copy`) to *whatever you want* (e.g. a pointer to a struct)
- Hint: The initial thread "main" does not have any parent, and is not created by `thread_create`, but it is initialised by `init_thread`

Lab 3: Exec

- How to track parent-children and 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 3: Exit

- **void** exit (**int** status)
- The exit status has to be made available to its parent process. This will be used in lab 5 when implementing wait().
- The process has to release all its allocated resources, such as free dynamically allocated memory, close opened files, close network connections, and so on. Release whatever resource *you* allocate in *your solution*
- If the process crashes for any reason, then the exit status should be -1. Furthermore, all of the allocated resources must be released
- Hint: Release resources in `thread_exit()` or `process_exit()`
- Hint: At exit, do: (make check will fail otherwise)
`printf("%s: exit(%d)\n", thread-name, thread-exit-value)`

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

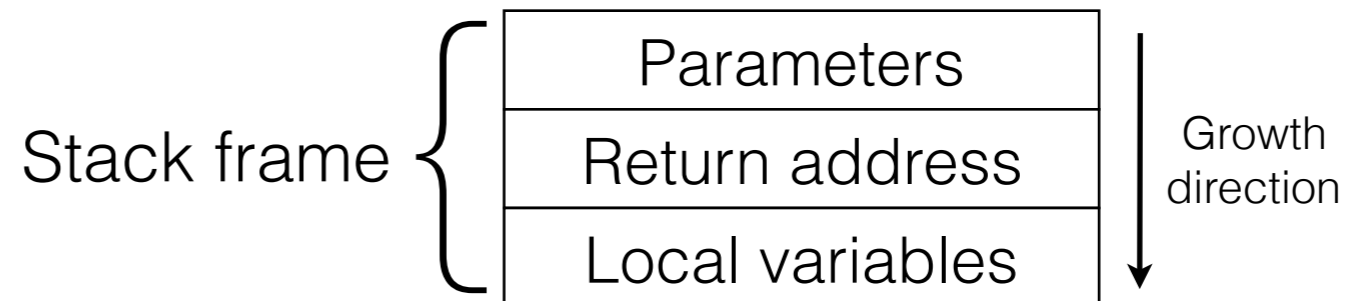
- To run a program with arguments in Pintos:
- Pintos (note the single quotes):
`pintos --qemu --run 'binary -s 17'`
- System call:
`exec("binary -s 17");`

Lab 4: Command Line

- Suppose we do this: `binary -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 `"binary\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]
0xbfffff3	argv[0][...]	"binary\0"	char[7]
...	word-align	unused	
0xbfffff0	word-align	unused	
0xbffffec	argv[3]	NULL	char *
0xbffffe8	argv[2]	0xbfffffd	char *
0xbffffe4	argv[1]	0xbfffffa	char *
0xbffffe0	argv[0]	0xbfffff3	char *
...			
0xbffffc8	argv	0xbffffe0	char **
0xbffffc4	argc	3	int
0xbffffc0	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

Pushing these variables to the stack is what argument passing 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 "binary -s 17\0", and you need to split it up in smaller parts.

Use (in `lib/string.[c|h]`):

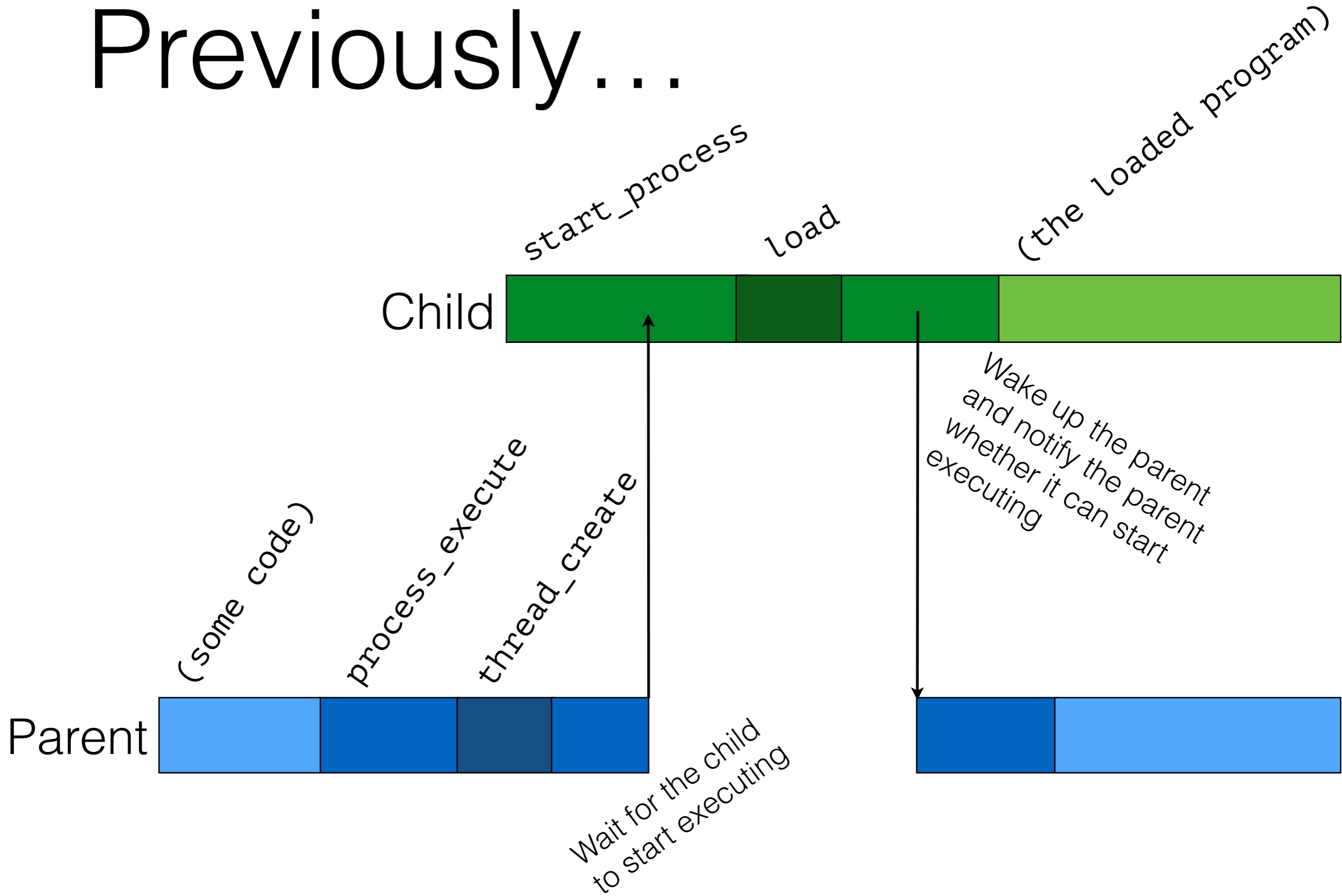
```
char * strtok_r(char *s, const char *delimiters, char **save_ptr)
```

- The following loop tokenises the string `s = "binary -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. "binary\0", "-s\0" and "17\0". However, it does so destructively!
- This is what the memory looks after the loop: `s = "binary\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...



Questions?