LINKÖPINGS UNIVERSITET

TDDE47/TDDE68: Concurrent programming and Operating Systems

Lab C: Multitasking

2026-01-15

Dag Jönsson
IDA/SaS

1 Introduction

1.1 Goal

In this lab you will start to work with concurrency and synchronization. You will take your OS from only being able to run a single user program per boot, to being able to run several different programs concurrently. Also known as multitasking.

To do this you will need to implement 3 more system calls. One of which you've already started working on.

1.2 Reading material

You probably haven't had some of the lectures yet, and that is fine. The later lectures will tackle concepts that are more relevant for the later assignment.

- Lectures 3, 4, 5, and 6
- Course literature chapters:
  - 3.2-3.4: Process scheduling to Interprocess Communication
  - 4.3 Multithreading models
  - 4.6 Threading issues
  - 6: Synchronization Tools
- Stanford Pintos:
  - Reference Guide Specifically A.3.2, A.3.3, A.5.2
- Files:
  - `threads/synch.h`
  - `userprog/process.c`
  - `threads/thread.[h,c]`
  - `lib/kernel/list.[h,c]`

1.3 Preparatory question

In the following text we will use the following definitions:

- parent = parent process
- child = a child process to some parent

Question 0: Basic concurrency

a) What is the main idea behind concurrency?
b) What is one of the main problems with concurrency?
c) What is synchronization? When is it important to use?
d) What tools do you have available in Pintos to achieve concurrency?
e) How many CPU cores are available in Pintos?
f) What threading model does Pintos utilize?

Question 1: Process creation

The following questions should be answered in the context of Pintos, and should help orient you for the assignments.

a) What functions are involved when a new process is started? What are their responsibilities?

b) How do these functions communicate between each other?
c) How many threads are involved in creating a new process?
d) During the process creation, What functions run on which thread? (parent or child thread)
e) How would a parent know if the child started correctly?
f) If the previous answer was "It doesn't" or similar: What changes would be needed to make it possible for the parent to know the status of the child startup?
g) If the new process was started correctly, what is the expected return value of `process_execute()` ?
h) Is there a difference between `pid_t` and `tid_t` ? What in that case?

Tip: Take a sneak peek at Figure 1.

Question 2: Semaphores

a) What operations can you use to manipulate a semaphore? What do they mean?
b) Is it a good idea to look at the internal values of a semaphore? Why/why not?
c) What happens when a thread calls `sema_down()` on a semaphore that currently has an internal value of 0?
d) What happens when a thread calls `sema_down()` on a semaphore that currently has an internal value of 1?
e) What happens when a thread calls `sema_up()` on a semaphore that currently has an internal value of 0?
f) Can you think of a use case for the previous answers in the context of your answers for **Question 1**?

At this stage you should be ready to create a solution for Assignment A.

Question 3: Process termination

This and below prep question can be saved until you start working on Assignment B.

a) What functions are involved in terminating a process?
b) What is the exit status used for? When should it be –1? When can it be any other value?
c) What process is interested in another process' exit status? What process should be able to read the exit status of another process?
d) When will the exit status be available?
e) Where should the exit status stored? Why?

Tip: You can take a peek at Figure 2 if you get stuck, it might help.

Question 4: Process waiting

a) What do we mean when we say "The parent is waiting on the child"? What is it waiting for?
b) What processes can a process wait on? Any? A specific set?
c) Is a parent process required to wait on all of its children?
d) Is there a function already defined that "waits" on a process? Are there any issues with it? If so what?

e) Is the above function called anywhere?

Question 5: Locks

a) What operations can you use to manipulate a lock?
b) Is it a good idea to look at the internal values of a lock? Why/why not?
c) What happens when a thread calls `lock_acquire()` on a lock that is already held by another thread?
d) What happens when a thread calls `lock_release()` on a lock that is already held by another thread?
e) What is the difference between a lock and a binary semaphore? Why do we even have locks?

Question 6: Lists

In Pintos we have some already defined data structures, in this question we are interested in the linked list declared and defined in `lib/kernel/list.[h,c]`. The files contains a lot of comments that explains pretty much everything you need to know. You are not expected to understand it completely, but knowing how you can use, and read about, it is important.

a) What operations does the list implement?
b) What struct is actually "stored" in the list? What struct is responsible for the *links* between the nodes?
c) How is it possible for the list to store *any* data type?
d) How do you get the stored data type back out of the list? Give an example.
e) How can you iterate through a list? Are there any pitfalls when doing so that you should be aware of? (**Hint:** What happens if we are removing elements while iterating?)

If you want to experiment with the Pintos list, you can find some prepared files in `pintos/standalone/plist`. Slightly easier to test your theories when you don't have to worry about the rest of Pintos.

2 Assignment A: `exec`

Start with studying Figure 1 if you haven't already, it should help you understand what the goal is, and what the current problem is. You can also use the figure to check your answers in Section 1.3.

In short, the problem can be described as the following: `process_execute()` currently doesn't return a value we can trust. Your assignment is to fix that.

You will also need to wire up the `exec` system call.

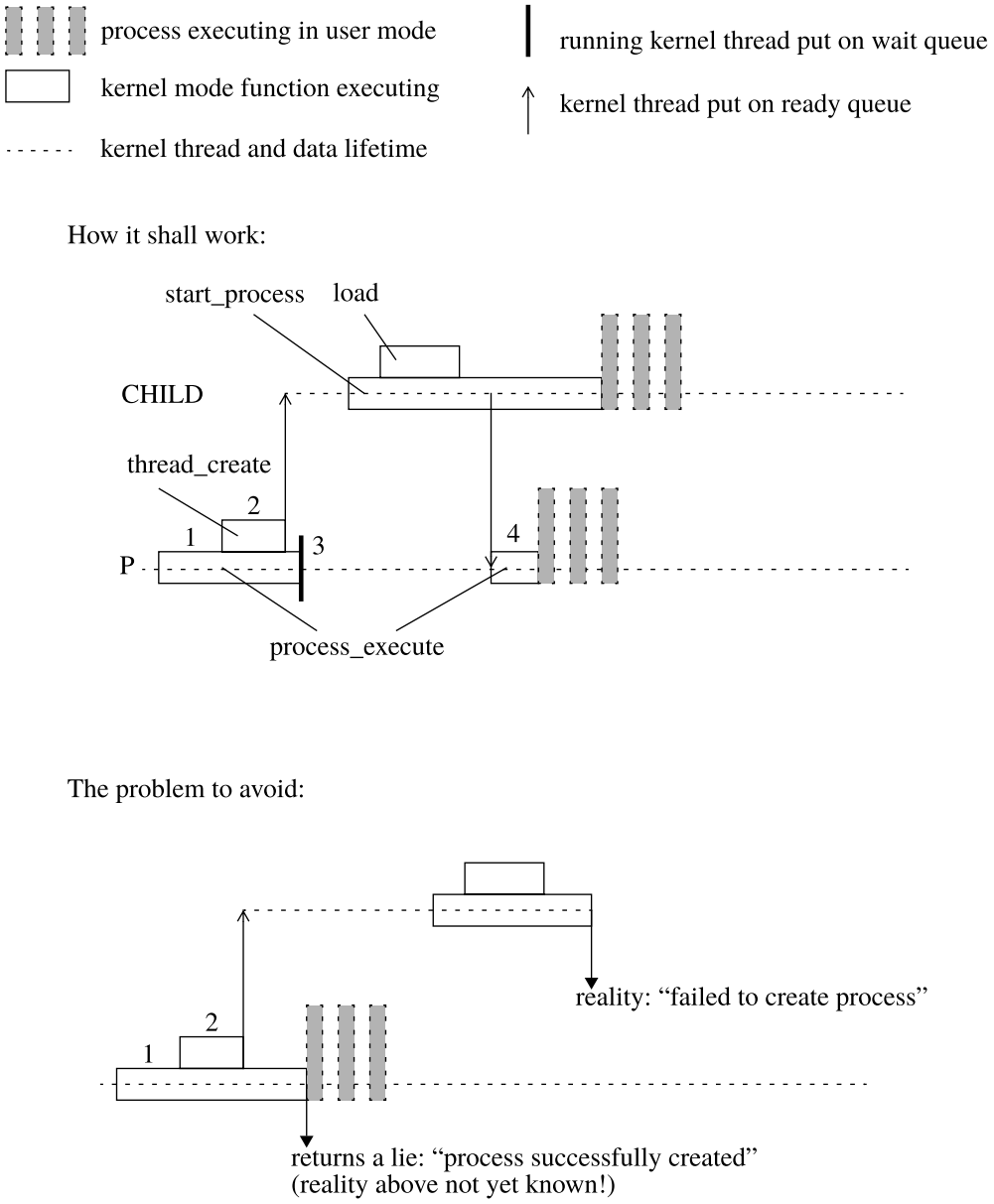
- `pid_t exec (char const *cmd_line)`

  Runs the executable given by `cmd_line`, any arguments will be put onto the new process' stack. If Pintos is unable to execute the `cmd_line` for any reason, return −1. Otherwise, return the (process) id given to the new process.

2.1 Testing

In `examples` you have two files, `exec-simple.c` and `exec-loop.c`. Just like previous examples you have a comment at the top that explains how to run them, and what to expect if they are run successfully.

Figure 1 — Process creation



3 Assignment B: `wait` , `exit`

Similar to what you did in Assignment A, you should start by studying Figure 2. In this figure you can see some different interactions between a parent and child process. You can also see an extra block between the time lines of the child and parent threads, with some dotted lines going back and forth. This block is to signify the minimum "process information" lifetime. You should already have some experience with the "process information", since you've fixed a bug with it in the first lab.

3.1 Details

In this assignment you will need to wire up the `wait` system call.

- `int wait (pid_t pid)`

  Return the exit status of the child process with the id `pid` . If the child hasn't finished executing yet, will wait until it has. If the child has already terminated, return immediately without waiting. If the child terminated abnormally, the exit status to return should be −1. If the parent has already waited on that child once, return −1. If `pid` refers to a process that is **not** a direct child to the calling process, return −1.

You will also need to revisit `exit` .

- `void exit (int status)`

  Terminates the current user program, passing the exit status to the kernel. A parent should be able to fetch the exit status by calling `wait` .

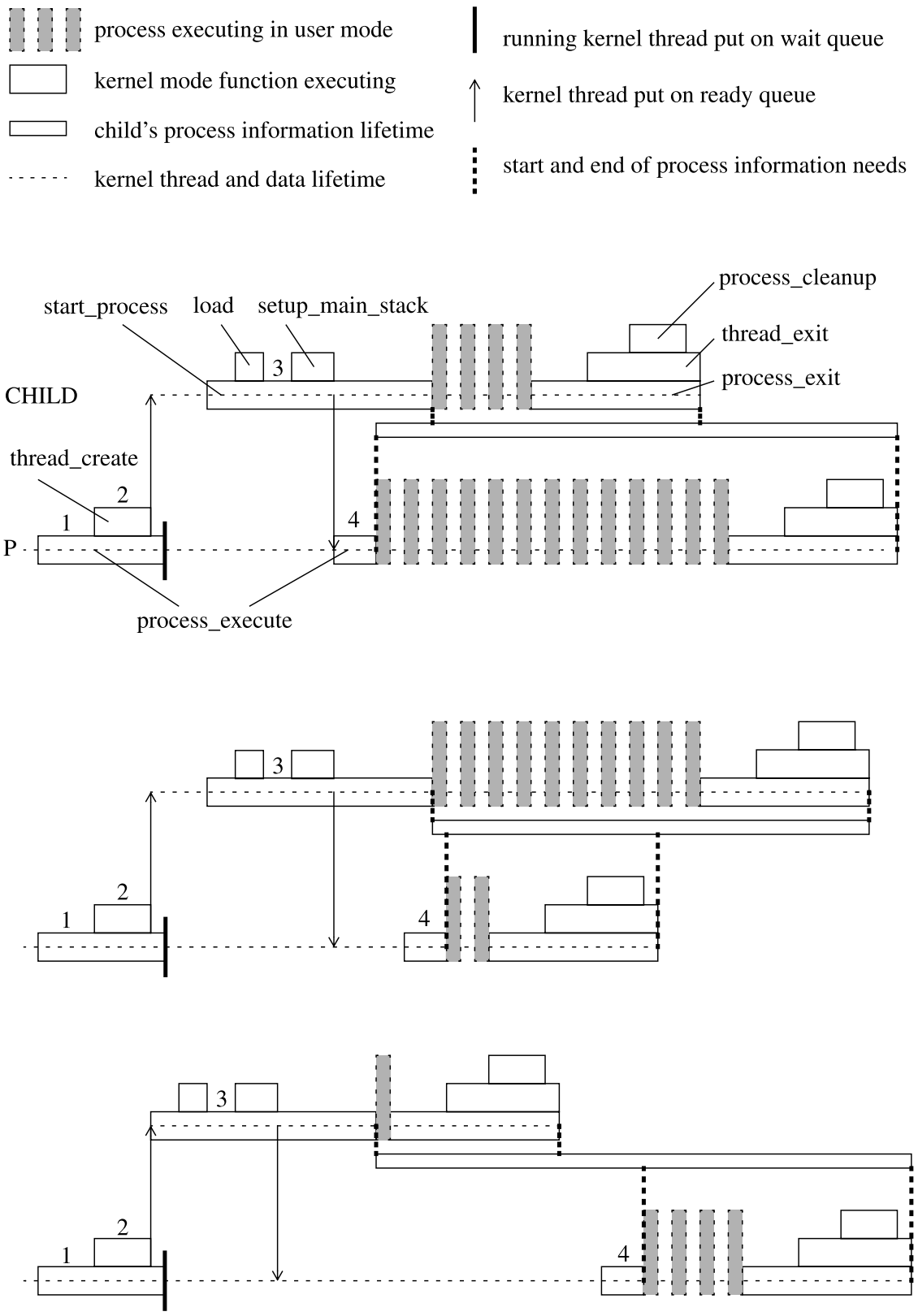
In this lab you will need to work a bit with the "process information" struct. The details are up to you to figure out, but consider the following:

- Any access to shared memory *need* to be synchronized.
- As soon as the struct is not needed anymore, it should be destroyed and any dynamically allocated memory freed.
- A process should be able to create an arbitrary number of child processes.

It's up to you to figure out what modifications you will need to apply to the struct, and how to utilize it.

You should study Figure 2 carefully to make sure you've covered all the cases. You will need to extend your solution in Assignment A to support your solution in this assignment.

Figure 2 — Process flow



3.2 Testing

You have finally reached the point in your work on Pintos that you can start using the test suite that we hinted at previously. To run the test suite, simply run `make -j8 check` from `pintos/userprog`. You will have some tests failing, 12 are expected to not be passable for now. See below list for which tests you can safely ignore for now.

- `tests/userprog/sc-bad-sp`
- `tests/userprog/sc-bad-arg`
- `tests/userprog/sc-boundary-3`
- `tests/userprog/create-null`
- `tests/userprog/create-bad-ptr`
- `tests/userprog/open-null`
- `tests/userprog/open-bad-ptr`
- `tests/userprog/read-bad-ptr`
- `tests/userprog/write-bad-ptr`
- `tests/userprog/exec-bound-2`
- `tests/userprog/exec-bad-ptr`
- `tests/userprog/wait-killed`

The test suite is fairly exhaustive, so don't be surprised if you fail some tests that you've implemented the solution for in the previous lab. It does not cover absolutely everything! For instance, it doesn't catch memory leaks. And no, `valgrind` does not work in Pintos. If you want to run a basic leak check you can run the `examples/basic_syscall.c` file with the `-L` flag to Pintos (after the `--`). I.E. `pintos -v ... -a sysc -- -f -q -L run sysc`.

The memory checker isn't perfect, but if you see any of the functions you have used showing up in the resulting list, you very likely have a leak and should investigate. If you are unsure, discuss it with your assistant.

Back to the test suite, it works by comparing the output from Pintos against an expected result. To check the results of the tests, you can check in the `userprog/build/tests/userprog` directory. Here you will find files like `halt.allput`, `halt.output`, `halt.result` and a couple of others that are not of much interest. Looking through the following files should be enough:

- `.allput`: Includes everything that Pintos printed during execution. Includes the `debug()` printouts. Might be helpful to track when processes are being started and terminated, and look for any debug prints you've added yourself.
- `.output`: Same as `.allput`, but excludes the `debug()` printouts. This is the file the test suite will run its matcher on.
- `.result`: Will contain a message trying to convey what went wrong. Often it will be a diff between the expected result and the actual result. Anything starting with a `+` is something that was printed when it should *not* have been printed. Anything starting with `-` are lines that we expected to be printed, but weren't.

Sometimes there are several different expected outputs, in which case it will show the difference between all expected results.

If you want to rerun the test suite, the safest way to make sure they are all run is to do a `make clean` first. It's very common to forget to do that and then get weird results back.

If you want to run a single test, you can use the `pintos-single-test` utility. If for example you want to specifically run the `halt` test, then you would call it like: `pintos-single-test tests/userprog/halt`.

You are also free to read and study the test files themselves. Sometimes it's easier to understand what goes wrong when you can see the code causing the issue. Please note that you shouldn't modify anything in the `tests/` directory.

4 Demonstration

For the demonstration you are expected to:

- Show that your solution passes the relevant tests in the test suite.
- Be able to answer questions of similar character as the prep questions.
- Be able to explain the modifications you made to Pintos in order to make `exec`, `wait`, and `exit` to work as expected.
- Show an understanding of how semaphores and locks work.