Concurrent programming and Operating Systems Lesson 3

Dag Jönsson

LINKÖPING UNIVERSITY

Lab 5

Overview

- Implement process waiting
  - Make it available as a syscall
  - The kernel will also need to use the implementation (see `threads/init.c`)
- Implement pointer validation
  - Need to validate any pointer that the user gives
- Validate your solution
  - Test suite containing 62 tests

What is ’wait’?

- A way for a parent process to wait for a child to finish execution
- The child’s exit status is the return
- Note that the kernel is waiting on the first process

wait

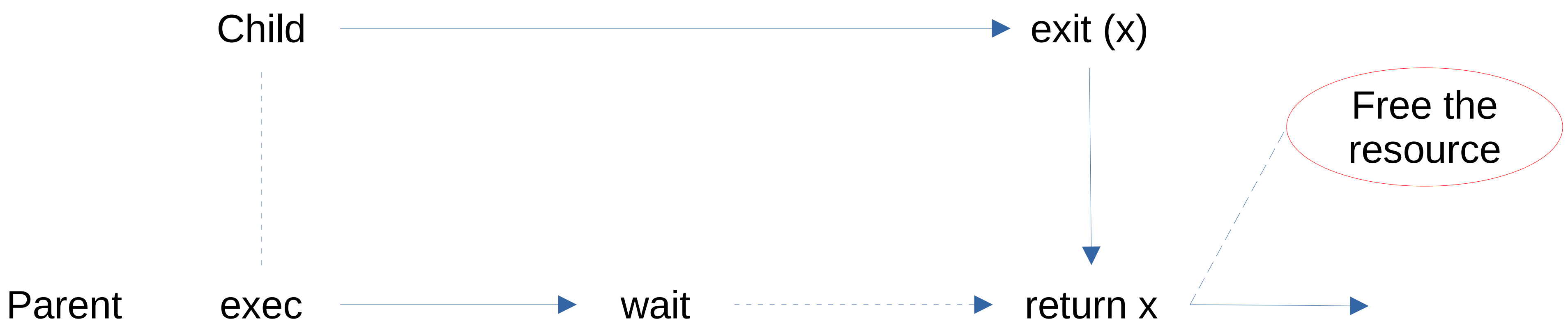
- Scenarios
  - Parent calls wait before the child terminates
  - Parent calls wait after the child terminates
  - Parent terminates before the child, without wait
  - Parent terminates after the child, without wait
- Your solution need to be able to handle all of the above
- Shared resources should be freed as soon as they are not needed anymore
-

wait

- A process can have several child processes, but only one parent
- `wait` can only be called once per child
- If anything goes wrong, -1 is expected as return
- Busy waiting is obviously not allowed

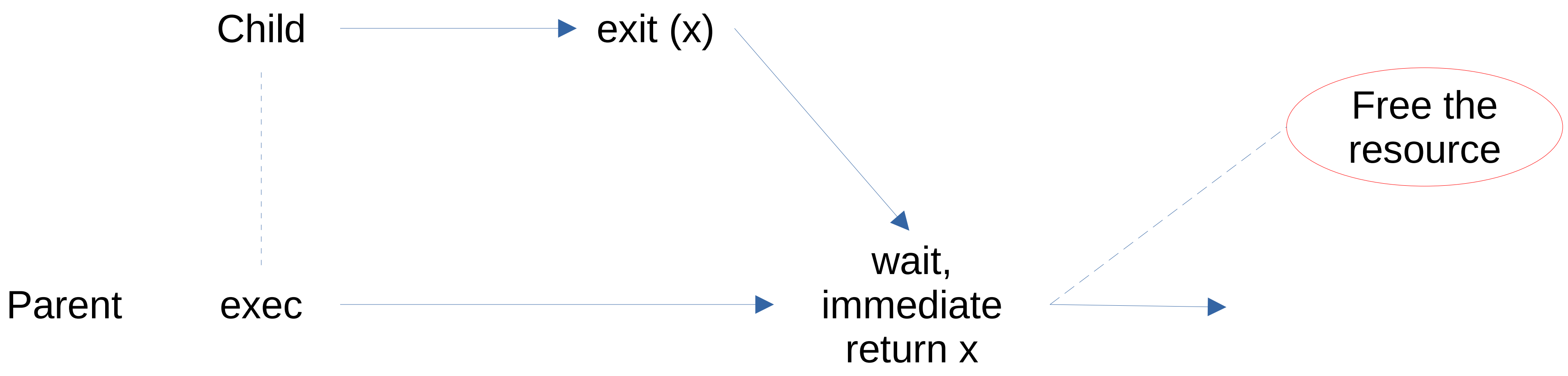
Scenario 1

- Parent waits for child to exit



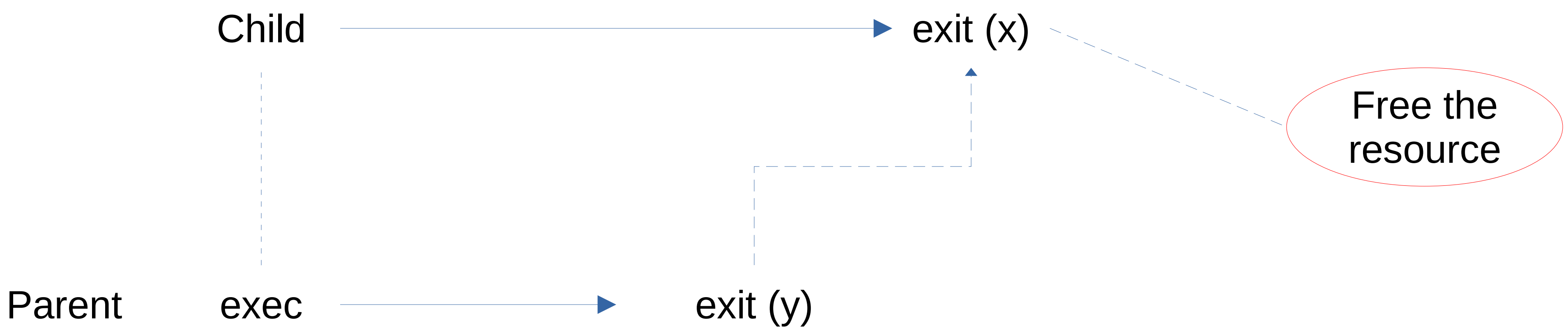
Scenario 2

- Parent waits after the child terminates



Scenario 3

- Parent never waits

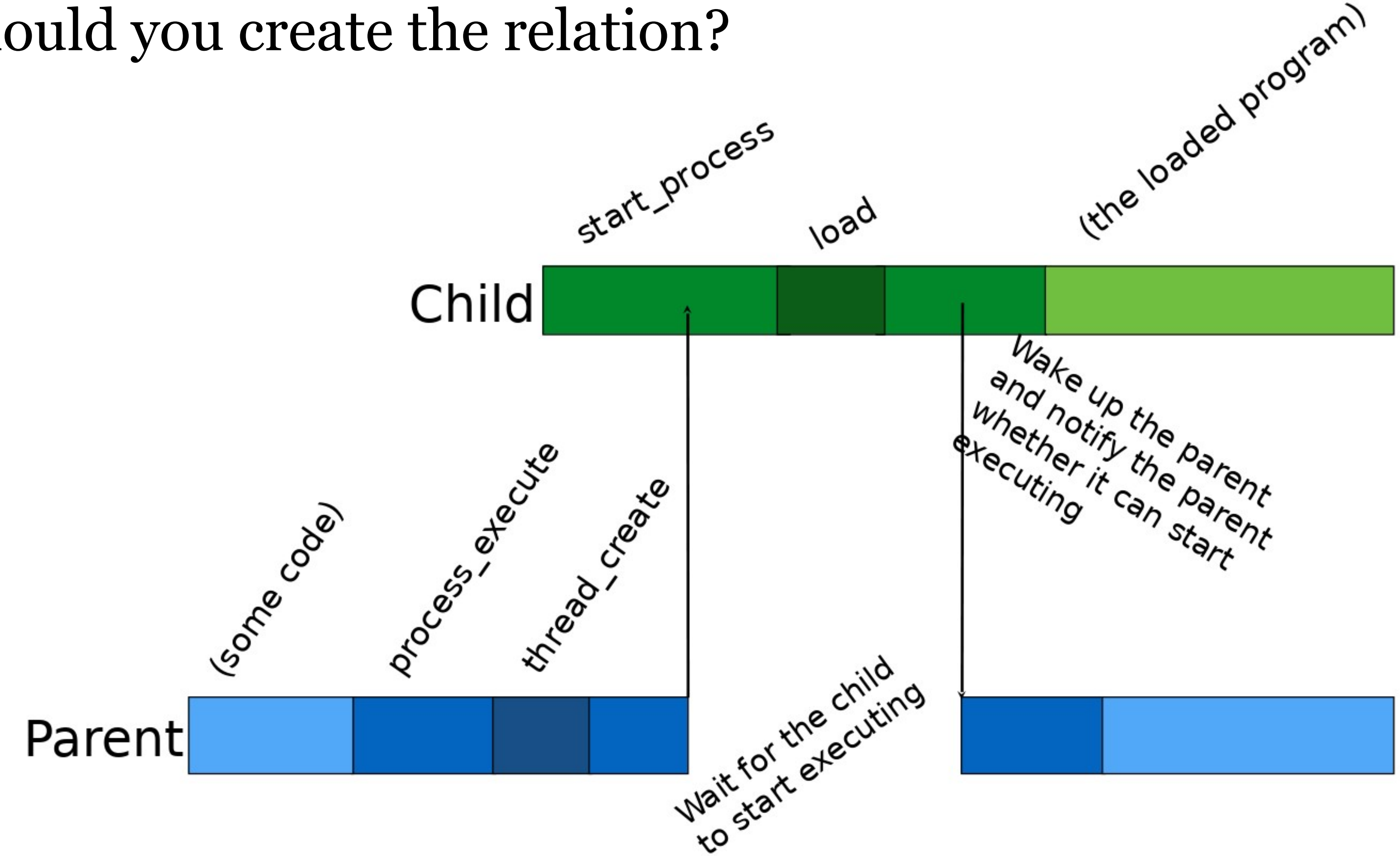


Wait resource

- So what is the resource?
- A shared struct, holding all the necessary data to facility wait
- Think of the resource as the relation between the processes
- Consider that a process can have several children, but only one parent
- Note: Do **not** make the `thread` structs the shared structure. Consider their lifetimes

Lab 4: Refresher

- When should you create the relation?



Pointer validation

- A valid pointer has the following properties
  - Below `PHYS_BASE`
  - Associated with a page in the current process' page table
- `pagedir_get_page()` can be used to verify the last point
  - This function is very expensive however. To pass later tests you will need to use it in a smart way
  - Consider what you know about the page once you've verified one address in it

Pointer validation

- Suppose a program makes the following syscall:
- `create((char *) PHYS_BASE – 12345, 17);`
- `filesys_create()` does not validate the string, and it's not `NULL`. Will likely crash Pintos (the kernel)
- The answer then is to validate the C-string before passing it along
  - Remember that a C-string is terminated with a `‘\0’` character
  - If it's not valid, just terminate the calling thread

Pointer validation

- Suppose a program makes the following syscall:
  - `write(1, malloc(1), 1000);`
  - Need to validate at most 1000 addresses.
    - Can be optimized, think about how the pages work
  - The answer then is to validate any given buffer before passing it along
    - In contrast to strings, we are given the length and do not have to search for `‘\0’`
    - If it’s not valid, just terminate the calling thread

Pointer validation

- Even the stack pointer needs to be validated!
- `asm volatile(”movl $0x0, %esp; int $0x30” :::);`
- <u>Hint:</u> It might be easier to treat the esp as a buffer, and validate as much as you need.
  - Remember that an int and pointers take up 4 bytes.

Test suite

- `tests/userprog/halt.c` – The actual test
- `userprog/build/tests/userprog/halt.result` – Result only
- `userprog/build/tests/userprog/halt.errors` – Errors, faulty output
- `userprog/build/tests/userprog/halt.output` – Complete printout of the test run. Possibly most useful

Lab 6

Overview

- Synchronise the file system in Pintos
- Reader-writers problem
- Even more tests!

File system

- `thread/malloc.[h|c]` – Heap memory allocation (shared, already synchronised)
- `devices/block.[h|c]` – Low-level operations on the hard drive (shared, already synchronised)
- `filesys/free-map.[h|c]` – Operations on the map of free disk sectors (shared)
- `filesys/inode.[h|c]` – Operations on inodes, which represent an actual file on disk. Operations on the inode actually change the content on disk! (shared)

File system

- `filesys/file.[h|c]` – A file object, keeping track of where in the inode to read and write (not shared)
- `filesys/directory.[h|c]` – Operations on directories (partially shared)
- `filesys/filesys.[h|c]` – High level operations on the file system (shared)

Reader-writer problem

- Some requirements:
  - Several readers should be able to read from the same file at the same time
  - Only one writer can write to a specific file at the same time
  - Several writers are able to write to *different* files at the same time
  - While a file is being read, there should be no write operation made
  - While a file is being written, no other process can read from/write to that file at the same time

Reader-writer problem

- Reader-writer algotihms can achieve the priviously mentioned requirements
- Important that you know what kind of solution you make, and have some motiviation why potential starving isn’t a problem
- <u>Hint:</u> There is at most 1 inode per physical file

Final hints

- To help you understand the problem, consider the following questions. Try to imagine what could happen in the worst case if two or more processed tried to:
  - Create and remove the same file at the same time?
  - Read and write the same file 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!
- ”At the same time” should be interpreted as the first operation is prempted by the second

Dag Jönsson
dag.jonsson@liu.se

LINKÖPING UNIVERSITY