Concurrent programming and Operating Systems

Lesson 3

Dag Jönsson

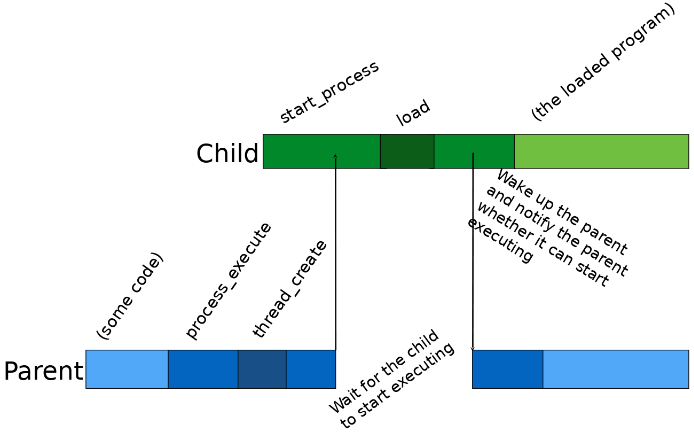
LINKÖPING UNIVERSITY

LINKÖPING
UNIVERSITY

Overview

- Implement syscall wait - handle different scenarios
- Implement input validation - check to make sure that we get valid input from the user program

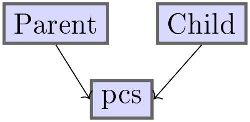
Lab 4: Refresher


start_process
load
(the loaded program)
Child
Wake up the parent and notify the parent whether it can start executing
(some code)
process_execute
thread_create
Parent
Wait for the child to start executing

wait

- `int wait(pid_t pid)` - sleep the parent until child finishes and return the child's exit status.
- Define a new structure for the shared memory

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



wait

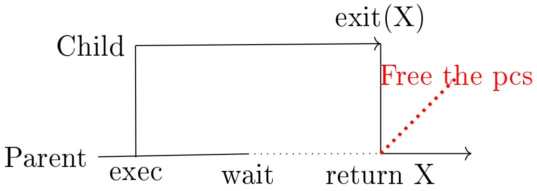
- Scenarios:
  - Parent calls wait before the child terminates
  - Parent calls wait after the child terminates
  - Parent terminates before the child, without wait
  - Parent terminates after the child without wait
- In each of these scenarios, your code must work and shared resources need to be freed when it's not needed anymore
- Remember that a process can have several children, but only one parent!

wait

- wait can only be called *once* per child.
- If anything goes wrong, -1 is expected as the return.
- Busy waiting is *NOT* allowed.
- Hint: Since the exit status has to be available even after the child terminates, store in dynamically allocated memory.

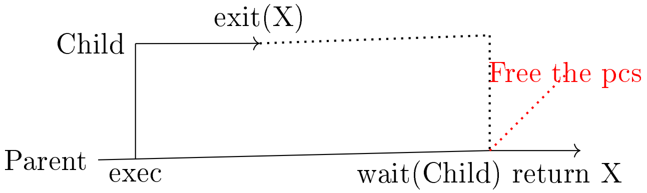
wait scenario 1

parent waits for child to exit



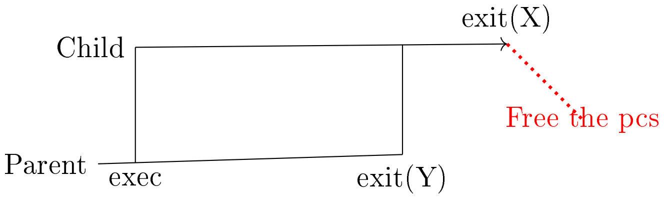
wait scenario 2

child exists before the parent, and then the parent waits



wait scenario 3

parent never waits for the child and exits



Input validation

- Argument paranoia: nothing the user processes does should crash Pintos
- Example: `read(STDIN_FILENO, 0xc0000000, 512);`
- *All* pointers from the user processes to the kernel must be validated!
- If a pointer is not valid, the caller should be terminated with exit status -1

Input validation

- A valid pointer from a user process comply with the following:
    - Below `PHYS_SPACE` in virtual memory (not in kernel memory)
    - Associated with a page in the page table for the calling process (`pagedir_get_page()`)
- `pagedir_get_page()` is an expensive operation, so it's not effecient to call it for every address. It's possible to only use it once per page a given buffer spans. (Why? How?)

Input validation

- Suppose a process calls
  `create((char *) PHYS_BASE - 12345, 17);`
- `filesys_create()` does not validate the string, and the string is not null. This will likely crash Pintos.
- Hint: You must check that the `char *` is a valid C-string by iterating over *every* character, and check that the pointer is valid. A valid C-string is null terminated (`’\0’`)

Input validation

- Suppose a user process calls `write(1, malloc(1), 1000);`
- Hint: We must check that every possible pointer is valid. In this case that would mean checking 1000 pointers (at most; you can optimise this by computing the page boundaries, and check those).
- Hint: In contrast to strings, the size is given and we do *not* have to search for `'\0'`

Input validation

- The user process can modify its own stack pointer: `asm volatile("movl $0x0, %esp; int $0x30" :::);`
- That means that you need to validate the stack pointer as well. If you increment the stack pointer, you need to redo that check!
- In other words, you need to validate the stack pointer for every argument you extract.
- Note that to check the memory for an integer you need to treat it as a 4 byte array (Why?)

Testing

- once you have implemented a solution for lab 5, you can run tests with `make -j check` from the `userprog/` folder
- the tests will test your solutions for labs 1, 2, 4, and 5. it's fairly common to have to fix something in older labs
- if you want to run a single test, you can do the following from `userprog/`:
  ```
  make build/tests/userprog/halt.result
  ```

Testing

- `tests/userprog/halt.c` - The actual test program
- `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 program run

LINKÖPING UNIVERSITY

Overview

- Synchronise the file system in Pintos
- Reader-writers problem
- Testing your implementation

File system

- You need to implement synchronisation for accessing data in files when they are shared between multiple processes that are not already synchronised
- Use locks and/or semaphores!
- You could synchronise the filesystem by using *one* lock for everything, this however will lead to **unacceptable** performance

File system

- `threads/malloc.[h|c]` - Heap memory allocation (shared, already synchronised)
- `devices/block.[h|c]` - Low-level operations on the 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 represents an individual file. When you write/read data to/from an inode you modify the actual physical file (shared)

File system

- `filesys/file.[h|c]` - A file object contains an inode and things like seek position. Every process has its own object (not shared)
- `filesys/directory.[h|c]` - Operations on directories (parially shared)
- `filesys/filesys.[h|c]` - Operations on the file system, such as create, open, close, remove and so on (shared)

Readers-writers

Some requirements:

- Several readers are 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
- When a process is reading a file, no other process can write to that file
- When a process is writing to a file, no other process can read from that file

Readers-writers

- Reader-writers algorithms can achieve the aforementioned requirements.
- Hint: Implementing a readers-preference is fairly easy, but might lead to starving writers
- Hint: There is at most 1 inode per physical file

Research lab example

- Imagine a research lab, where either research or drop-in visits may happen
- Only one of the activites can be active at a time
- Outside of the lab there are a sign that indicates if the room is occupied or not, and a counter of the number of visitors in the room
- Design a simple protocol to enter the room based on the signs. It should be visible at a glance if the room is available for either activity

Hints

- Some questions you can ask yourself to help you understand what needs to be done. What can happen, in the worst case, if two processes try 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 interpreted as the first operation is interrupted by the second

Dag Jönsson
dag.jonsson@liu.se

www.liu.se

LINKÖPING
UNIVERSITY