Concurrent programming and Operating Systems Lesson 2

Dag Jönsson

LINKÖPING UNIVERSITY

General data structures

Doubly linked list

- **`lib/kernel/list.[h|c]`**
- Well documented, study the source files for example usage
- Can store any kind of data
- Do **not** reuse the list elem between different lists
- Sortable, through the use of
  `typedef bool list_less_func(const struct list_elem *a, const struct list_elem *b, void *aux);`

Synchronisation

Synchronisation

- What is it and why do we need it?
- Consider the simple expression: `++i`
- Once compiled, will do the following:
  1) Fetch `i` from memory and store it in a register;
  2) Increment the value in the register by 1;
  3) Write the register value back into memory;
  4) Return the value stored in the register;
- One expression is in reality several instructions!

Synchronisation

What can happen if two processes, `p1` and `p2`, executes `++i` at the ”same time”?

1) Fetch i from memory to a register `p1` `p2`
2) Increment the register by 1 `p1` `p2`
3) Write the register value back into memory `p1` `p2`
4) Return the value stored in the register `p1` `p2`

The result will be that `i` increased by 1, not 2.

Critical section

- A sequence of instructions, operating on shared resources that need to be executed by a given number of processes without *interference*. Also called *mutal exclusion*
- Care has to be taken when working with shared resources
- Examples: lists, arrays, network connections, shared variables (memory), files and so on

Synchronisation primivites

- Solution already exists! Pintos have implementations for the following:
  - Locks
  - Semaphores
  - Conditions (also known as monitors)
- Declared & defined in **`threads/synch.[h|c]`**
- See also Pintos Doc Appendix A.3

Locks

- Operations: **`acquire_lock()`** and **`release_lock()`**
- To release the lock, a thread need to hold it first
- Used to ensure mutual exclusion
- Note: Do not keep the lock longer then necessary

Lock example

- Simple lock solution for our earlier example
- Instead of directly calling i++, wrap it in a helper function

```
1  // Initialization somewhere
2  int i = 0;
3  struct lock lock;
4  init_lock(&lock);
5
6  int inc() {
7    lock_acquire(&lock);
8    int ret = i++;
9    lock_release(&lock);
10    return ret;
11 }
```

Semaphore

- Operations: **`sema_down()`** and **`sema_up()`**
- Unlike the lock, any thread can call **`sema_up()`**
- Works by trying to decrement an internal counter. If the result is >0, will wait until the counter is >0.
- The counter is configured at initialization

Semaphore example

- Imagine we have a thread calling the **`send()`** function, adding a new message, call it **S**.
- And we also have a second thread calling **`recv()`**, call it **R**.

```
1  struct list msgs;
2  struct semaphore sema;
3  init_sema(&sema, 0);
4
5  void send(struct msg *msg) {
6    append(&msgs, msg);
7    sema_up(&sema);
8  }
9
10  void recv() {
11    sema_down(&sema);
12    struct msg *msg = list_pop(&msgs);
13    handle_msg(msg);
14  }
```

Interrupts

- **Internal:** Caused by CPU instructions, e.g., system calls
- **External:** Caused by hardware outside the CPU, e.g., timers, keyboards, disks.
  - `intr_disable()` causes external interrupts to be postponed
  - Returns the previous state, which should be used when restoring
- See also Pintos Doc Appendix A.4
- Try to use synch primitives in the first hand, but sometimes it might be required to call `intr_disable()`

Scheduler

- Handles thread scheduling
- Threads can be preempted so that the scheduling can occur
- Preemption is based on timer interrupts
- In Pintos, the scheduler preempts a thread after 4 timer ticks.
- There are 100 ticks per second

Synchronisation II

- The primitives are implemented by disabling interrupts
- During that time, external interrupts are ignored, so no preemption can occur
- Crude, but effective

Synchronisation II

Keep the following in mind

- External interrupts are disabled only during the **`lock_acquire()`** or **`sema_down()`** call
  - Your thread may be preempted while executing its critical section!
  - However, the synch primitives will protect the critical section, if used correctly
- You can **not** use synch primitives in external interrupt handlers
- **Hint:** Disable interrupts rather than using locks when the interrupt handler is the problem

Lab 3

Lab 3: Files and functions

- `devices/timer.[h|c]`
- `void timer_init()`
- `void timer_sleep(int64_t ticks)`
- `void timer_interrupt()`

Lab 3: Hints & Testing

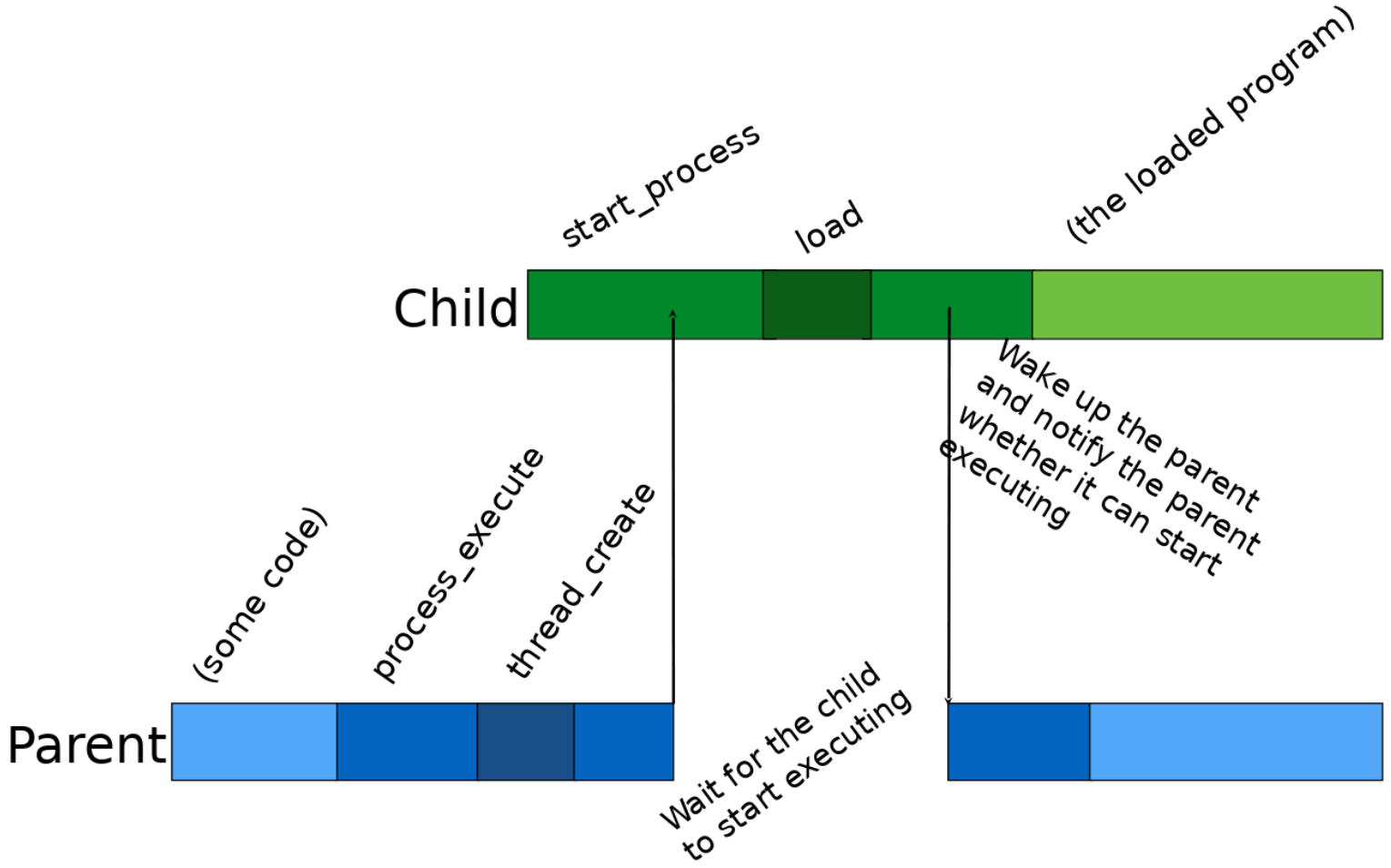
- The scheduler will only schedule threads that have the status = READY
- You can control this status with the **`thread_block()`** and **`thread_unblock()`** functions
- When done: run **`make -j check`** in **`threads/`**
- The tests will pass as is, but the task is to remove the busy-wait in **`timer_sleep()`**

Lab 4

Lab 4: Overview

- Implement one more syscall: **`pid_t exec(const *cmd_line)`**
- Spawn a new process based on the given cmd_line
- Current implementation does not wait to check if the new process spawned correctly or not
- The ”parent” need to allow the **`start_process()`** function to finish before continuing

Lab 4: exec flow



Lab 4: Code of interest

- Closer study of the following functions are neccessary

```
tid_t process_execute(const char *cmd_line) {
  ...
  tid = thread_create(cmd_line, PRI_DEFAULT, start_process, cl_copy);
  ...
}

tid_t thread_create(const char* name, int priority, thread_func *function, void *aux);

static void start_process(void *cmd_line_);
```

Lab 4: Hints

- **`start_process()`** is only ”called” in one place, **`process_execute()`**. Because of this, you can change what the **`void *cmd_line`** is actually pointing to.
- Assume that PID and TID are the same
- Don’t keep track of the relationship between the ”parent” and ”child” yet

Dag Jönsson
dag.jonsson@liu.se

LINKÖPING UNIVERSITY