Concurrent programming and Operating Systems

Lesson 2

Dag Jönsson

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Doubly linked list

- Declared and defined in `lib/kernel/list.[h|c]`
- Store any kind of data, data retrieval via macro
- Well documented, if you want to try it out without Pintos, just copy the files and use them as normal
- Do make an effort to understand how to use the list
- Remember: Do **not** reuse the elem structure between different lists

Hashtable

- Declared and defined in `lib/kernel/hash.[h|c]`
- Documented in the Pintos documentation, A.8
- Not necessary to use

Synchronisation

- What is it and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does the following:
  1. Fetch i from memory and store it in a register;

Synchronisation

- What is it and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does the following:
  1. Fetch i from memory and store it in a register;
  2. Increment the register by 1;

Synchronisation

- What is it and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does 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;

Synchronisation

- What is it and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does 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

Synchronisation

- What is it and why is it needed?
- Consider the following, simple, expression: `++i`
- The expression, when compiled, does 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?

1. Fetch i from memory to a register `p1`

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`

Synchronisation

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

1. Fetch i from memory to a register
2. Increment the register by 1 `p1 p2`

Synchronisation

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

1. Fetch i from memory to a register
2. Increment the register by 1 `p2`
3. Store the value in the register in memory `p1`

Synchronisation

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

1. Fetch i from memory to a register
2. Increment the register by 1
3. Store the value in the register in memory `p2`
4. If of interest, return the value in the register `p1`

Synchronisation

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

1. Fetch i from memory to 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 `p2`

Synchronisation

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

1. Fetch i from memory to 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

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

Critical section

- A sequence of instructions, operating on shared resources, that should be executed by a given number of processes without *interference*. Also known as *mutual exclusion*
- Concurrent accesses to shared resources can lead to unexpected behaviours.
- Typical examples are data structures (e.g. lists), network connections, shared variables, hard drives, files, and so on

Synchronisation primitives

To help us solve these problems, Pintos implements the following primitives

- Locks
- Semaphores
- Conditions (also known as monitors)

Well documented in `threads/synch.[h|c]` and also in Appendix A.3 in the Pintos documentation.

Locks

- Two operations: `acquire_lock` and `release_lock`
- The *same* process that acquires the lock must also release it.
- Ensures that at most one process executes a critical section enclosed by the acquire and release of the lock.
- As an example, in the `i++` example earlier, p1 would have to finish before p2 could start executing.
- Overzealous use of locks leads to poor utilisation of concurrency, so do not lock more than absolutetly necessary.

Lock example

```c
int shared = 0;
struct lock lock;
init_lock(&lock);

int func() {
  lock_acquire(&lock);
  int ret = shared++;
  lock_release(&lock);
  return ret;
}
```

Lock example

```c
int shared = 0;
struct lock lock;
init_lock(&lock);

int func() {
  lock_acquire(&lock);
  int ret = shared++;
  lock_release(&lock);
  return ret;
}
```

- A:5
- B:?

Lock example

```c
int shared = 0;
struct lock lock;
init_lock(&lock);

int func() {
  lock_acquire(&lock);
  int ret = shared++;
  lock_release(&lock);
  return ret;
}
```

- A:5
- B:6

Lock example

```c
1  int shared = 0;
2  struct lock lock;
3  init_lock(&lock);
4
5  int func() {
6    lock_acquire(&lock);
7    int ret = shared++;
8    lock_release(&lock);
9    return ret;
10 }
```

- A:6
- B:7:lock

Lock example

```
1  int shared = 0;
2  struct lock lock;
3  init_lock(&lock);
4
5  int func() {
6    lock_acquire(&lock);
7    int ret = shared++;
8    lock_release(&lock);
9    return ret;
10 }
```

- A:6:waiting
- B:9:released

Lock example

```c
1  int shared = 0;
2  struct lock lock;
3  init_lock(&lock);
4
5  int func() {
6    lock_acquire(&lock);
7    int ret = shared++;
8    lock_release(&lock);
9    return ret;
10 }
```

- `A:7:lock`
- `B:9`

Lock example

```
1  int shared = 0;
2  struct lock lock;
3  init_lock(&lock);
4
5  int func() {
6    lock_acquire(&lock);
7    int ret = shared++;
8    lock_release(&lock);
9    return ret;
10 }
```

- `A:7:lock`
- `B:9`

Semaphores

- A generalisation of locks
- `sema_down` and `sema_up`
- Not necessarily the same process doing `sema_down` doing the `sema_up`
- You can set the number of processes that are allowed to execute the critical section concurrently
- For locks, this is set to 1. With semaphores you can set it to any other non-negative value.
- When the number is set to 0, any process calling `sema_down` will immediately wait until some process calls `sema_up`.

Semaphore example

```c
struct list msgs;
struct semaphore sema;
init_sema(&sema, 0);

void send(struct msg *msg) {
  safe_append(&msgs, msg);
  sema_up(&sema);
}
void recv() {
  sema_down(&sema);
  struct msg *msg =
    safe_pop(&msgs);
  handle_msg(msg)
}
```

Semaphore example

```
struct list msgs;
struct semaphore sema;
init_sema(&sema, 0);

void send(struct msg *msg) {
  safe_append(&msgs, msg);
  sema_up(&sema);
}
void recv() {
  sema_down(&sema);
  struct msg *msg =
    safe_pop(&msgs);
  handle_msg(msg)
}
```

- W:10:waiting
- S:?

Semaphore example

```c
struct list msgs;
struct semaphore sema;
init_sema(&sema, 0);

void send(struct msg *msg) {
  safe_append(&msgs, msg);
  sema_up(&sema);
}
void recv() {
  sema_down(&sema);
  struct msg *msg =
    safe_pop(&msgs);
  handle_msg(msg)
}
```

- W:10:waiting
- S:6

Semaphore example

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

- `W:10:waiting`
- `S:7:up(sema)`

Semaphore example

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

- `W:12:executing`
- `S:6:appending`

Semaphore example

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

- `W:13:work1`
- `S:7:up(sema)`

Semaphore example

```
struct list msgs;
struct semaphore sema;
init_sema(&sema, 0);

void send(struct msg *msg) {
  safe_append(&msgs, msg);
  sema_up(&sema);
}
void recv() {
  sema_down(&sema);
  struct msg *msg =
    safe_pop(&msgs);
  handle_msg(msg)
}
```

- `W:12:work2`
- `S:?`

Interrupts

- **Internal**: Caused by CPU instructions, for example system calls, page faults and so on.
- **External**: Caused by hardware devices outside the CPU, for example timers, keyboards, disks and so on. The function `intr_disable()` postpones the handling of external interrupts, which in turn causes internal interrupts to be postponed as well.

The interrupt infrastructure is documented within Appendix A.4 in the Pintos documentation.

Scheduler

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

Synchronisation II

- The synchronisation primitives in Pintos are implemented by disabling interrupts
- When external interrupts are disabled, the scheduler cannot preempt processes, thus no other process can execute a critical section in the primitives concurrently
- This is fairly crude, but it gets the job done within Pintos

Synchronisation II

Beware the following

- External interrupts are only disabled while you are acquiring/releasing the lock or semaphore. In other words, your critical section can still be 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/semaphores otherwise)

Busy waiting

- Sometimes processes have to wait for something, for example, acquire a lock or semaphore, or simply wait for a number of ticks
- This is done by calling the `timer_sleep`, which currently will call the `thread_yield` function, which causes the caller to "give up" its timeslot on the CPU.
- This is inreadibly inefficient, and it's your task to improve this!

LINKÖPING UNIVERSITY

Lab 3: Files and functions

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

Lab 3: Hints

- The lab can be solved by using synchronisation primitives
- The cleaner solution is to call `thread_block()` and `thread_unblock()` directly. These are defined in `threads/thread.c`
- You will need a list to keep track of sleeping threads, make use of the Pintos list `lib/kernel/list.h`
- Hint: To quickly check if there any threads to wake up, keep the list sorted

Lab 3: Testing

- Run `make -j check` from the `threads/` folder
- An individual test can be run like this `pintos run alarm-single` (Grab the names from the output from `make check`).
- The tests will pass as it is, since busy-waiting is a way to solve the problem, it's just *very* inefficient.
- Remove any `printf` you've added during debugging, otherwise you will never pass the tests, since they check the output from Pintos.

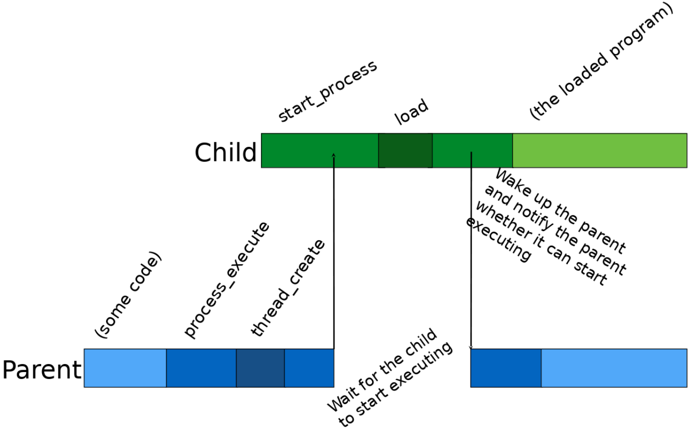
Lab 4: Overview

- Make it possible to spawn new processes from another process
- Syscall `exec` - Start up a child process that executes given file
- `pid_t exec(const char *cmd_line)`

Lab 4: exec

- Spawn a new *child* process that loads the file. If successfully loaded, return the process ID (PID) of the child, -1 otherwise.
- The current implementation does not wait to see if the child could be started. This is problematic.
- You need to make sure that the parent wait for the child to actually start executing before moving on.

Lab 4: exec flow



Lab 4: exec

The following functions and lines of code are of interest

```
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: exec hints

- The only place where `start_process` is "called" is in `process_execute`. Hence, you can change the parameter of `start_process (cmd_line_)` and the argument to `thread_create (cl_copy)` to *whatever you want* (e.g. a pointer to a struct)
- You can assume that TID and PID are the same thing within Pintos.
- Don't need to store the relationship between processes yet

Dag Jönsson
dag.jonsson@liu.se

www.liu.se

LINKÖPING UNIVERSITY