CSCI 350
Pintos Intro

Mark Redekopp

Resources

- Pintos Resources
  - https://web.stanford.edu/class/cs140/projects/pintos/pintos.html#SEC_Top
    - Skip Stanford related setup in section 1.1 and 1.1.1
  - http://bits.usc.edu/cs350/assignments/Pintos_Guide_2016_11_13.pdf
    - Keep this handy!!!

Emulated OS

- Could run on an actual x86 machine
  - But painful to debug for students
- Runs on at least two emulators:
  - bochs (project 1)
  - qemu (project 2-4?)

Startup

- src/threads/start.s
- src/threads/init.c
  - Main entry point for Pintos

Project 1 Area

- All your project 1 code is in:
  - src/threads
  - src/devices/timer.*

Tour of thread.h and thread.c

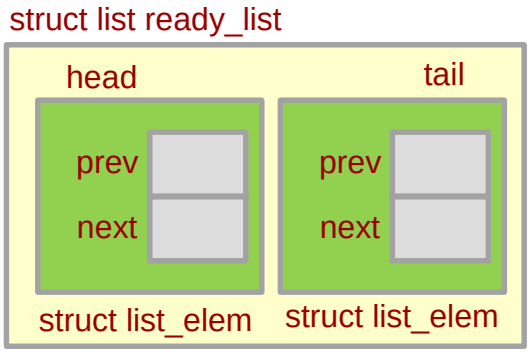
- thread_init()
  - Initialize the threading system & turns main() into a thread
- thread_start()
  - Start the threading system
  - Creates an "idle" thread to run if no other threads ready which executed idle()
- thread_create()
  - Create a thread's data structure
  - Uses init_thread() helper function
- schedule(), next_thread_to_run(), switch.S
  - Select next "ready" thread and perform a context switch

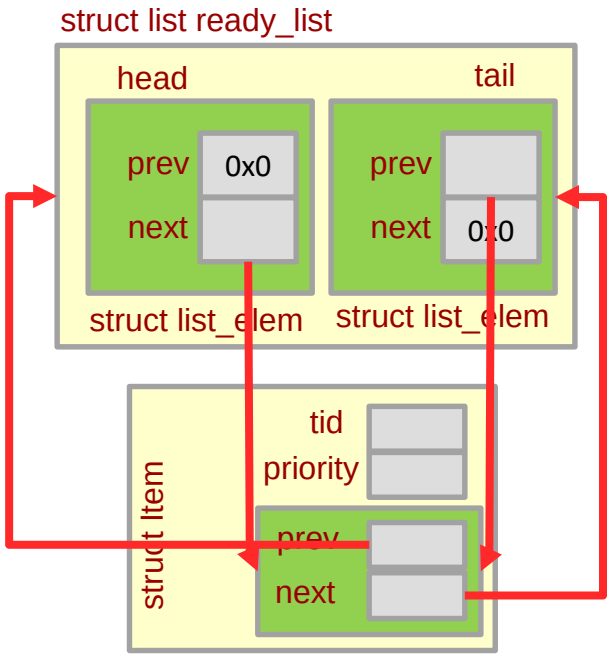
PRACTICALS

Lists

- Look in lib/kernel/list.h(.c)
- struct list
- struct list_elem

```
struct list ready_list;
struct Item {
  int tid, priority;
  struct list_elem elem;
  // could contain other list_elems if
  //  it is desirable to be a member of many lists
};
struct Item first;
void init()
{
  list_init(&ready_list); // construct empty list
  list_push_back(&ready_list, &first.elem);
}
```

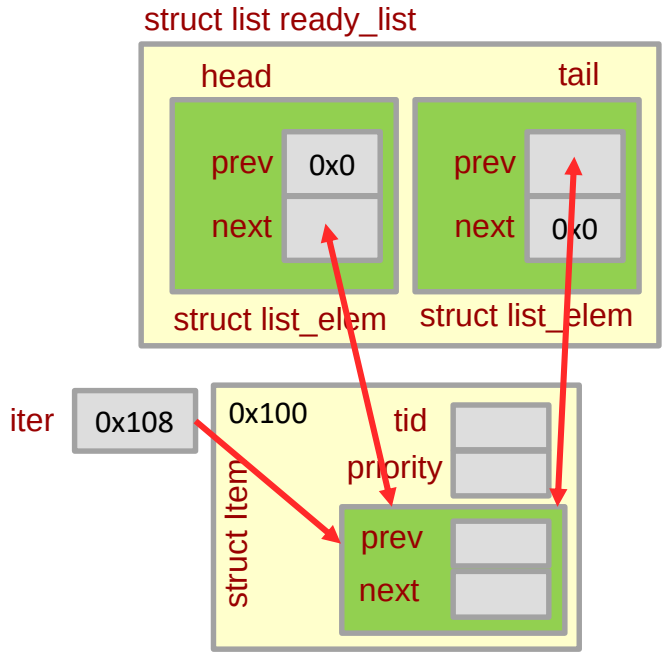




Lists

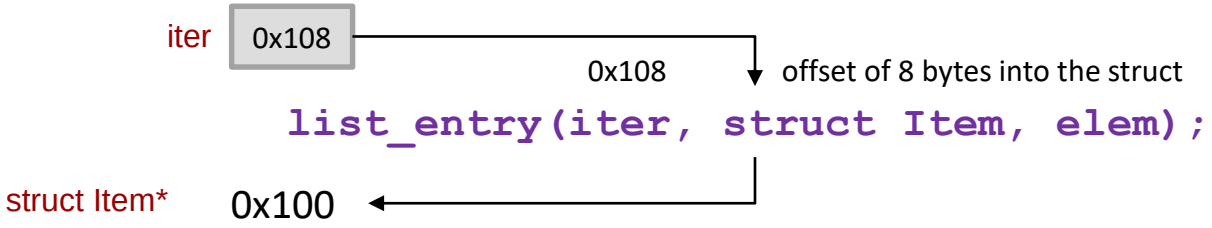
- Iterating
  - Uses struct list_elem*
- list_entry macro to get pointer to enclosing struct

```
struct list ready_list;
struct Item {
  int tid, priority;
  struct list_elem elem;
  // could contain other list_elems if
  //  it is desirable to be a member of many lists
};
struct Item first;
void init()
{
  list_init(&ready_list); // construct empty list
  list_push_back(&list.first);
}
```




```
  struct list_elem* iter = list_begin(&ready_list);
  while(iter != list_end(&ready_list))
  {
    struct Item* curr =
      list_entry(iter, struct Item, elem);
    // do something with curr Item

    iter = list_next(iter);
  }
```



Building and Running

- Navigate to src/threads
  - $ make
- Go to build directory
  - $ cd build
- To run all tests and see which pass and which fail
  - $ make check
- To run a single threads test
  - `$ pintos -v -- -q run alarm-multiple`
  - Options before the '--' are generally to configure the emulator and Pintos VM environment (e.g. virtual disk, etc.)
  - Arguments after the '--' tell Pintos what you want to do after the OS boots (e.g. run a kernel test, run a user program, etc.)
  - Replace alarm-multiple with the name of the test to run (see test names in src/tests/threads)
  - For project 1, tests are compiled into the kernel and available to run
  - For other projects, you'll run separate user applications on top of the kernel

Building and Running

- To check if the output is as expected for a single test
  - (Section 1.2.1 of Stanford Pintos site)
  - Go to the src/threads/build directory
  - $ make tests/threads/alarm-multiple.result
  - (Replace 'alarm-multiple' with the desired test name)
  - If that .result file already exists, just delete it
    - $ rm tests/threads/alarm-multiple.result

Install GDB macros

- In a terminal, git clone (or pull) the `pintos-base` repo
- Navigate to the pintos-base folder
- Copy the macros to your home folder
  - `$ cp src/utils/pintos-gdb-macros ~`
- Point the pintos-gdb script to that file
  - `$ which pintos-gdb`
    - Note the location and navigate to that folder
  - Edit pintos-gdb in a text editor (sublime, etc.)
    - Change the line the starts GDBMACROS=… and replace the … with `/home/csci350/pintos-gdb-macros`

Debugging

- Start Pintos in the emulator with the --gdb option
  - `$ pintos --gdb -v -k -T 60 --bochs -- -q run alarm-single`
- Connecting via pintos-gdb
  - In a separate terminal window, navigate to the 'build' folder in the src/threads (or whatever project you are working on)
  - Run pintos-gdb
    - `$ pintos-gdb kernel.o`
- Attach to the running Pintos instance
  - `target remote localhost:1234 (or "debugpintos" w/ macros)`
  - Set breakpoints (`break init.c:90`)
  - Resume the program (`cont`)
  - Use `s = step` and `n = next`

Debugging

- Use `print` to print a variable
- Accessing current thread TCB
  - `print $esp`
    - Note the address and change the last 3 digits to 000 in the next statement (e.g. 0xc000ee84 => 0xc000e000)
  - `print ((struct thread*)0xc000e000)->name`
- intr0e error is often due to context_switch at wrong time (i.e. disable interrupts or do your work before you yield, etc.)

Helpful Breakpoints

- thread.c : schedule()
- thread.c : next_thread_to_run()

Project 1

- Only responsible for parts 1 (thread_sleep / alarms) and 2 (priority scheduling and donation)