Project 1

Thierry Sans

Overview

```
devices/timer.c          |   42 +++++-
threads/fixed-point.h |  120 ++++++++++++++++++
threads/synch.c          |   88 ++++++++++++-
threads/thread.c         |  196 ++++++++++++++++++++++++++----
threads/thread.h         |   23 +++
5 files changed, 440 insertions(+), 29 deletions(-)
```

- ➡ Most changes in threads and devices
- ➡ Look in lib/kernel for useful data structures: list, hash, bitmap

Pintos thread implementation

➡ Pintos implements user processes on top of its own threads

- Per-thread state in thread control block structure

```
struct thread {
    ...
    uint8_t *stack; /* Saved stack pointer. */
    ...
};
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```

- C declaration for asm thread-switch function

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

- Thread initialization function to create new stack

```
void thread_create (const char *name, thread_func *function, void *aux);
```

Desirable timeline

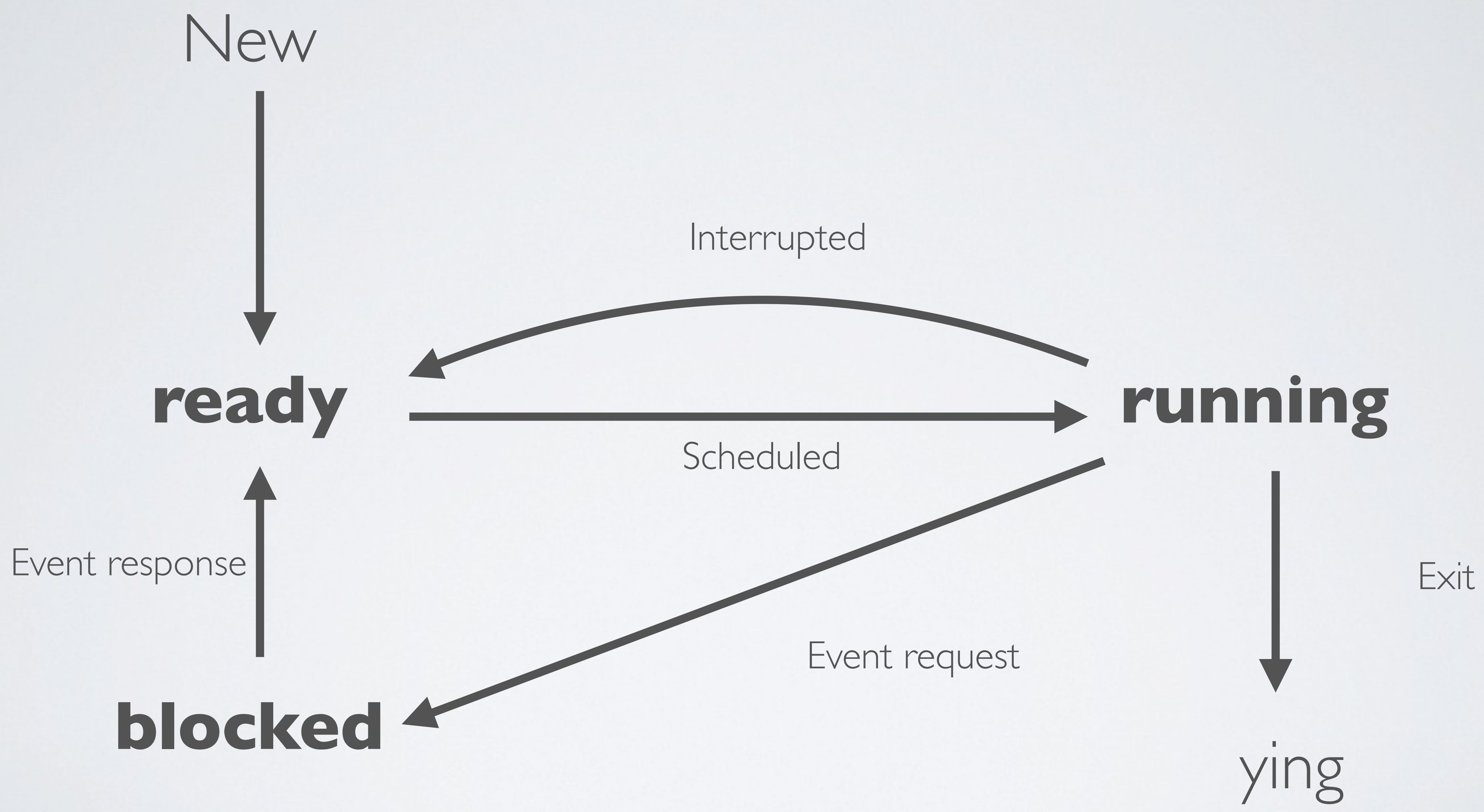
- Week 1 : From polling to interrupts
- Week 2 : Priority Scheduling
- Week 3 : Multilevel Feedback Queue Scheduler

From polling to interrupts

Alarm Clock

- ➡ Reimplement `void timer_sleep(int64_t ticks)` in `devices/timer.c`
- ◉ Existing implantation uses "busy waiting" (hardware polling)
- ✓ Suspends execution of the calling thread until time has advanced by at least ticks timer ticks (interrupt)

Thread States



Synchronization

Two techniques to serialize access to shared resource

1. **Disabling interrupts**

   turns off thread preemption; only one thread can run

   - Undesirable unless absolutely necessary

2. **Synchronization primitives** (`threads/synch.h`)
   - Semaphores
   - Locks
   - Condition variables

Priority Scheduling

Priority Scheduling

- ➡ Replace round-robin with priority-based scheduler
  - always run a thread with higher priority
  - yield immediately when higher priority thread is ready
- ◉ Priority inversion problem
- ◉ Starvation problem

Priority Inversion Problem

➡ A low priority threads (L) holds a resource needed by a higher priority thread (H)

- H is blocked because L has locked the resource
- L is blocked because a medium-priority thread (M) is running

✓ Fixed by priority donation

Priority donation

- ➡ A higher priority thread "donates" its priority to the lower priority thread it is blocked on
  - H "donates" its high-priority to L
  - When releases the resources lock, L returns to low priority
  - H runs immediately (since lower priority-threads should yield to higher-priority threads)

Multiple Priority Donation

If multiple threads needs a resource, the priority of the thread holding the resource is the max of all priorities

➡ Effective priority is the max of donated priorities

Chained Priority Donation

➡ Donated priorities propagate through a chain of dependencies

- H donates to M
- M donates priority to L

Priority Scheduling

```
void thread_set_priority (int new_priority)
```

- Set the current thread's priority to new_priority
- Yield if the thread no longer has the highest priority
- If thread has donated priority, it still operates at the donated priority

```
int thread_get _priority ()
```

- Returns the current thread's priority
- With priority donation returns the higher (donated) priority

Multilevel Feedback Queue Scheduler

Principles

➡ Multilevel feedback queue scheduler tries to be fair with CPU time

- No priority donation
- Give highest priority to thread that has used the least CPU time recently
- Prioritizes interactive and I/O-bound threads
- De-prioritizes CPU-bound threads

✓ Details in section 2.2.4 and Appendix B

Priority

```
priority = PRI_MAX - recent_cpu/4 - nice*2
```

✓ Details in Appendix B.1

Nice

`nice` allows threads to declare how generous they want be with there own CPU time

➡ Integer value between `-20` and `20`

- `nice > 0`
  lower effective priority, gives away CPU time
- `nice < 0`
  higher effective priority, takes away CPU time from other threads

✓ Details in Appendix B.1

recent_cpu

`recent_cpu` : CPU time a thread has “recently” received

- Exponentially waited moving average
- Incremented every clock tick when a thread is running
- Recomputed for all threads every second:

```
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice
```

✓ Details in Appendix B.3

load_avg

`load_avg` : average number of ready threads in the last minute

- Single value system wide
- Initialized to zero
- Recomputed every second

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads
```

✓ Details in Appendix B.4

Implementation

Add `-mlfqs` kernel option to allow the scheduling algorithm to be configured at startup time

➡ add to `parse_options()`

No priority donation

- `thread_set_priority()` should do nothing
- `thread_get_priority()` returns priority calculated by scheduler

✓ Details in section 2.2.4 and Appendix B

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*