DAW Design & Implementation

Using Threads

Parallelism, in general

- Data parallelism
- SIMD: single instruction, multiple data
- “lots of data, all being processed in the same way”
- Task parallelism
- “more than one thing to do”
- Threads are related to task parallelism

Old Programs

- Read input from user
- Read data from disk
- Compute something
- Print or store the answer
- Exit

Less Old Programs

```
while (!done)  {
    read input from user
    read data from disk
    compute something
}
print or store the answer
exit
```

Event-Driven Programming

- The application displays a graphical interface throughout most of its existence
- The application takes input from a mouse or similar object(s) in addition to the keyboard
- The application has no predetermined computational purpose – it does what you ask it do by interacting with it as it runs

The Main Event Loop

```
display user interface
while (!done) {
    event = get_next_mouse_or_keyboard_event();
    do_something_with (event);
}
exit
```

Main Event Loop Issues

- How do we handle other kinds of I/O?
- USB, Bluetooth, MIDI, OSC, network

The Extended Main Event Loop

```
display user interface
while (!done) {
    event = get_next_mse_or_kbd_or_file_event();
    do_something_with (event);
}
exit
```

More Main Event Loop Issues

- When does drawing happen?
- What about non-event-driven things?
- e.g data analysis, lots of file I/O, waiting on slow network connections

The Really Extended Main Event Loop

```
display user interface
while (!done) {
    event = get_next_mouse_or_keyboard_event (timeout);
    if (event) {
        do_something_with (event);
    } else {
        do_application_specific_idle_stuff ();
        redraw_screen ();
    }
}
exit
```

Two more problems

- What happens if “app-specific idle” stuff takes too long?
- What happens if app has stuff do ALL the time, not just when there are no events?

Such as ...

- Audio I/O
- Spectral (FFT) analysis
- Peak data computation

Other Kinds of Input/Output

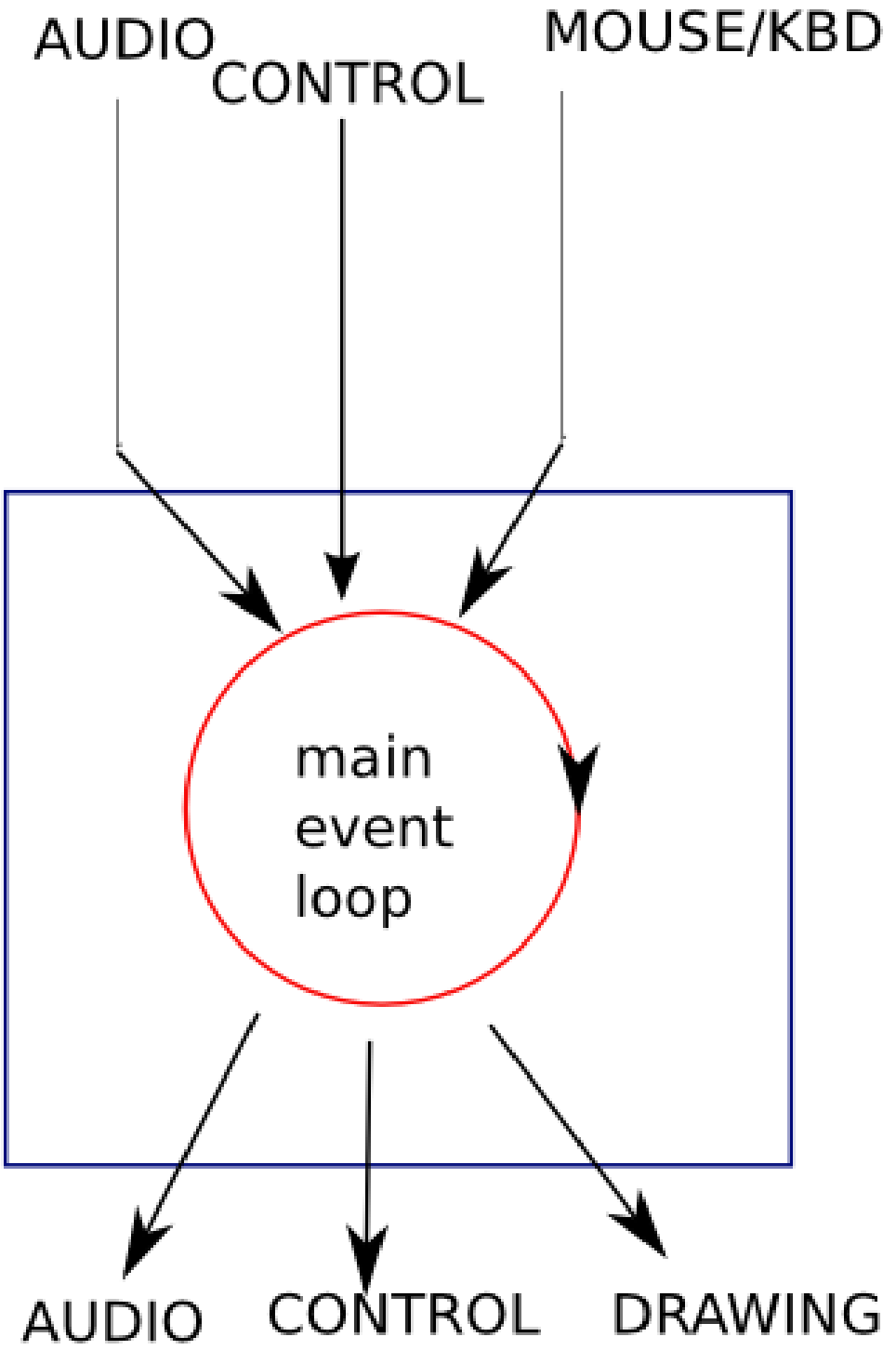
- We could use the main event loop to handle (e.g.) MIDI or OSC
- But the handling the data is delayed by GUI event handling and redrawing
  - Could be bad

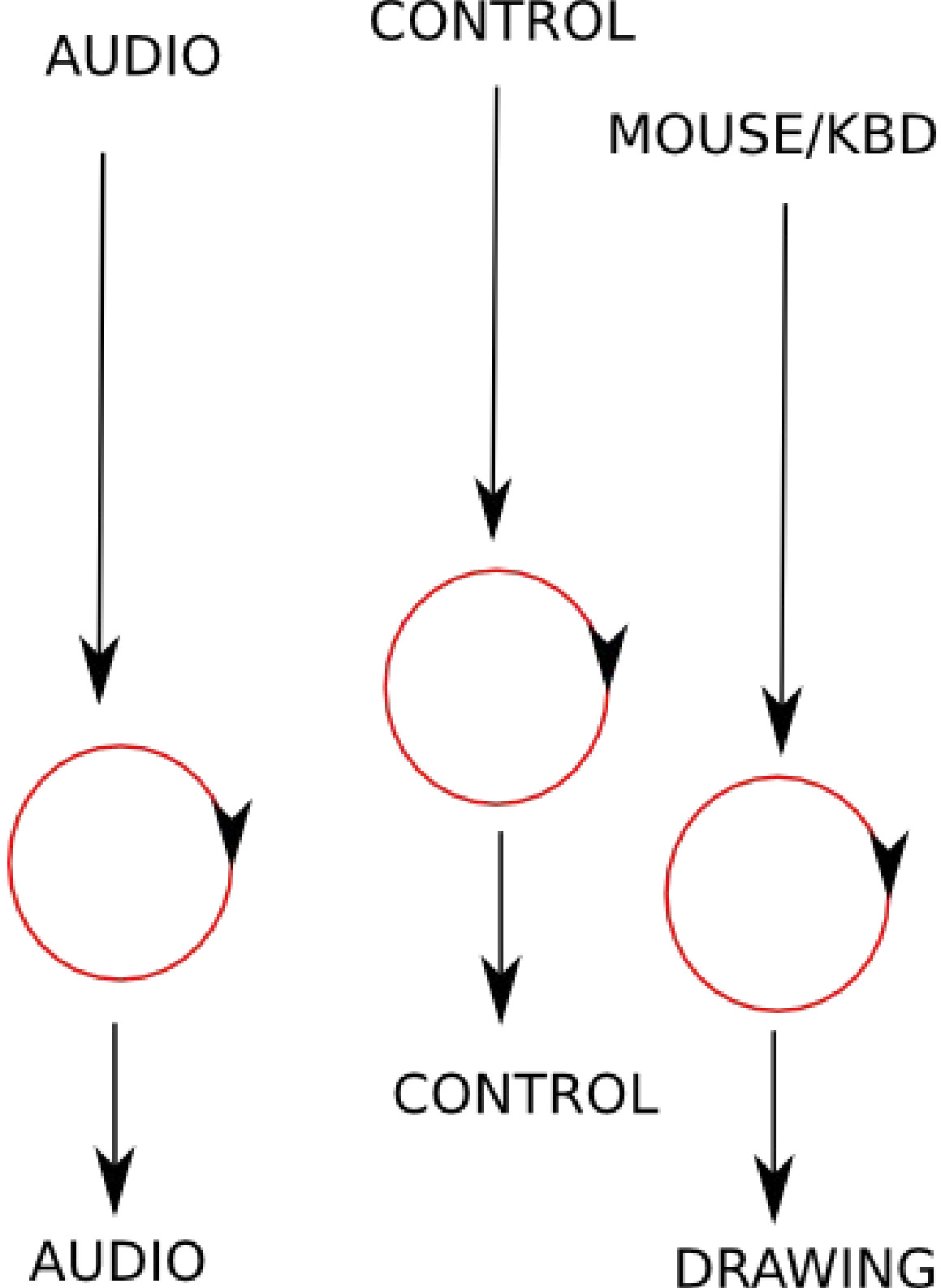
Hacks for app-specific idle code

```
my_slow_and_expensive_idle_function ()
{
        while (!done) {
                do_some_expensive_stuff ();
                tell_GUI_to_process_events ();
        }
}
```

- Requires a partitionable task
- Ugly, toolkit dependent code in non-GUI functions

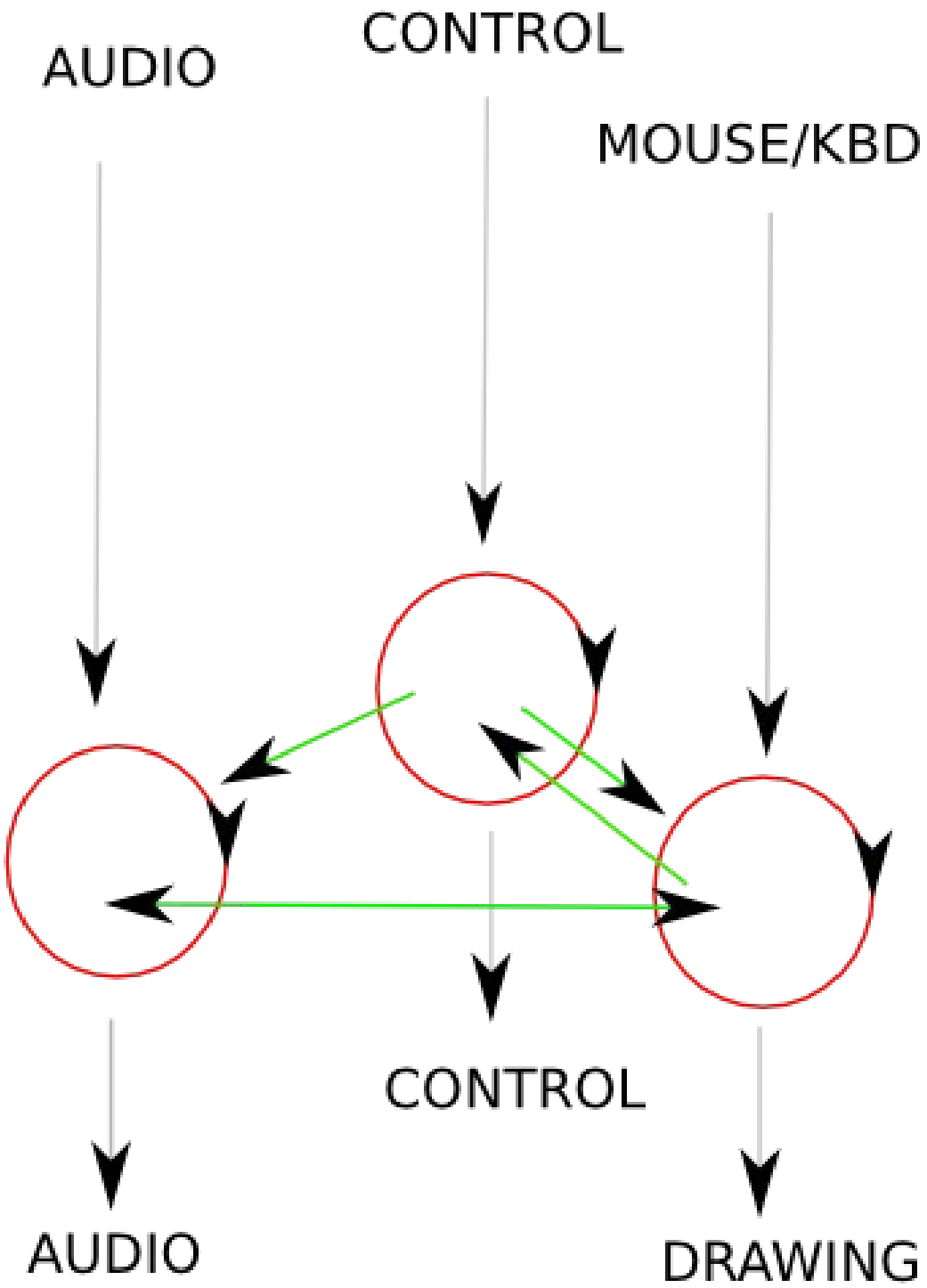
Time for Threads




AUDIO
CONTROL
MOUSE/KBD
CONTROL
AUDIO
DRAWING

Yes but ...

- What happens if threads need to talk to each other?


AUDIO
CONTROL
MOUSE/KBD
CONTROL
AUDIO
DRAWING

Communication Between “tasks”

- GUI thread wants to start playback
- MIDI/OSC thread wants to change a plugin parameter
- Audio thread has detected an xrun

Without threads

```
while (!done) {
        event = get_next_event ();
        do_something_with (event);
}
```

- Just add audio, control etc. to “events”
- Voila!
- No bounded response times
- control/analysis/audio/GUI all get in each others way

Thread-to-Thread Communication

Let's do this some other time
Can use thread API primitives, or custom
Lock-free/realtime issues

Threads & More Threads

- We used to have 1 thread in our app, now we have N
- Different tasks within the program don't interfere with each other – good
- But there is still only a finite number of processors (NCPU)
- If (NCPU < N) ... problems?
- Yes ... and no.

Just a few details

- Ordinary code:

```
this_function (an_argument);
that_function (another_argument);
the_other_function ();
```

- Adding a thread call “that_function()”

```
this_function (an_argument);
thread_create (that_function, another_argument);
the_other_function ();
```

- Simple. Too simple ?

Consequences

- The old order of execution of the functions was known: this, that, the_other
- The new order is known only as: this (that and/or the_other)
- `that_function()` and `the_other_function()` are executing “at the same time” or .. maybe `that_function()` first, and `the_other_function()` second, or maybe vice versa, or ....
- For fully independent task parallelism, no problem
- If `that_function()` and `the_other_function()` have any shared data or other dependencies, we have issues
- But not till next week...

Operating System Details

- You don't need to know this to write an application
- You do need to know this to use the OS effectively
- You also need an OS to provide the correct mechanisms

OS Scheduling

- The operating system is dealing with many threads all the time
- There are N applications, each with at least 1 thread, running on NCPUs
- One of the central jobs of the OS is to decide which threads get to run on which processors and when
- A thread can be asleep, blocked or runnable
  - Asleep – its own choice
  - Blocked – waiting for I/O (e.g. disk, keyboard)
  - Runnable – all other states
- Only runnable threads are scheduled

OS Scheduling II

- OS kernel runs “every so often”
- System timer or other interrupts
- When done handling interrupt, decides if a different thread should run
- Considers a variety of factors in making the decision

OS Scheduling III

- Classic unix scheduling considers: thread priority, how much time the thread has run for, CPU cycles vs. disk I/O
- Very good for multi-user systems with a mix of very interactive programs and “batch processing” in the background
- Allowing more time per “slot on CPU” - better throughput for batch processing
- Allowing less time per slot – better interaction for users
- Not even close to useful for audio

Scheduling in the Real(time) World

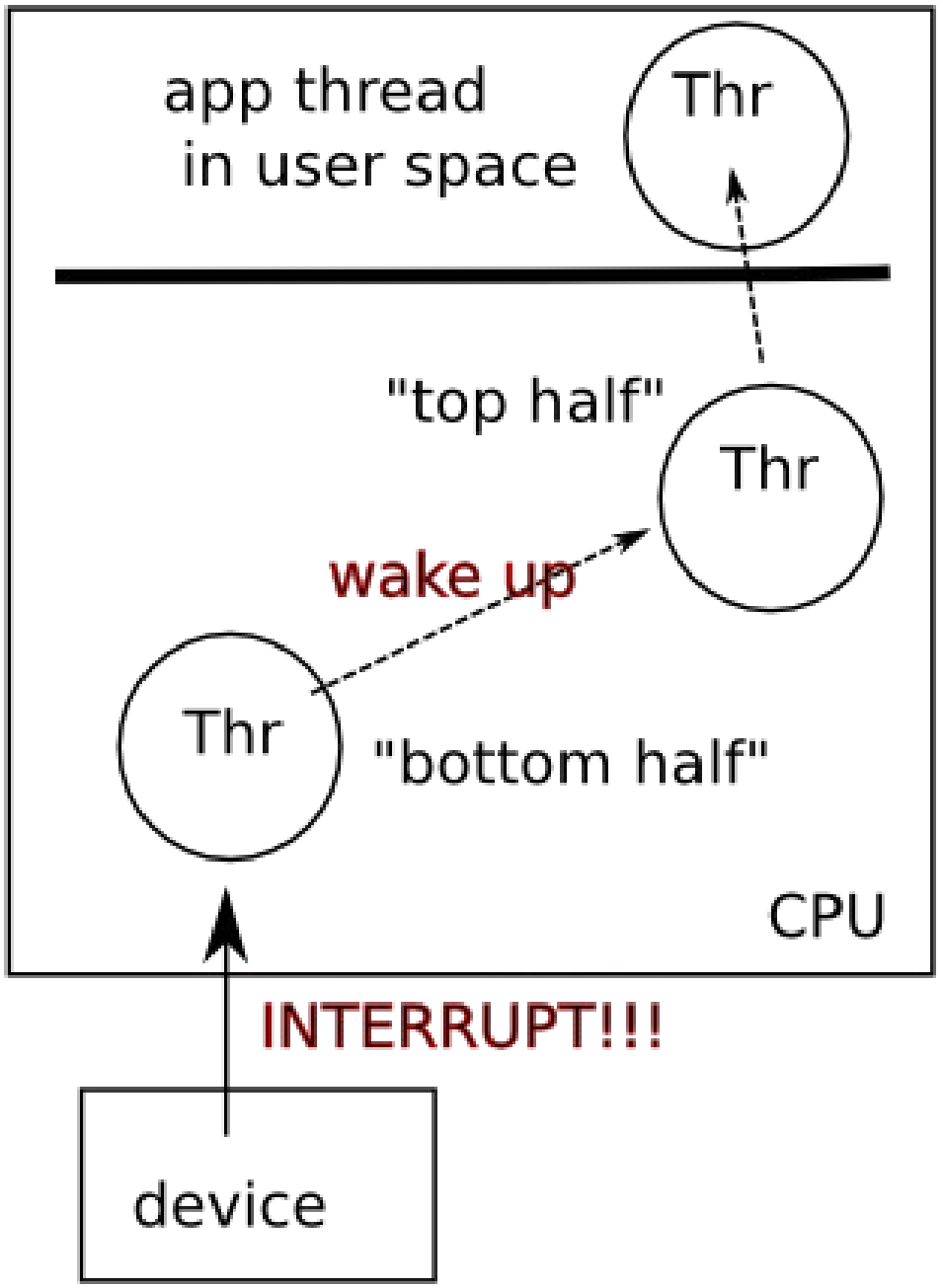
- Non-realtime threads get previously described treatment, or an alternative (fair share, etc)
- Realtime threads can choose between round-robin or first-in-first-out (FIFO) treatment
- If there are any realtime threads ready to run, then they ***must*** run instead of non-realtime threads
- RT threads are only ranked by priority
- Round-robin: still time-sharing
- FIFO: threads run it blocks or sleeps

What about the kernel?

- Your app has a RT audio thread that is busy processing audio
- Some device (e.g. the keyboard) interrupts the CPU
- How do we stop the interrupt handling in the kernel from slowing down the audio processing thread, which has a deadline to meet?

Splitting Interrupt Handlers Apart

- Most interrupt handling involves two parts
- Reading data from the device and writing values back to notify it that we have handled the interrupt
- Doing something at the OS level with the data received (e.g. wake up a program)
- The first part is very,very, very fast
- The second part is unpredictable
- Threads!


app thread
in user space
Thr
"top half"
Thr
wake up
Thr
"bottom half"
CPU
INTERRUPT!!!
device

Overall Thread Priorities on Linux

- Bottom half interrupt handling
- Realtime application threads
- Top half interrupt threads
- Other kernel threads
- Non-RT application threads
- Note: it is never *required* to block interrupts except while inside a bottom half handler
- This ordering assumes an “RT-patched” kernel

Compare with Windows

- Interrupt Service Routines (ISRs)
- WDM Deferred Procedure Calls (DPCs) FIFO/LIFO queue, three priorities (High, Medium, Low Importance), but DPCs cannot preempt other DPCs
- Real-Time Priority Threads Timesliced, execute at Win32 priorities 16 through 31, can raise IRQL from PASSIVE (lowest) to arbitrarily high levels (i.e., block interrupts)
- Normal Priority Threads Timesliced, execute at Win32 priorities 1 through 15, can raise IRQL

Compare with OS X

- OS X is a bit more complicated
- Rather than simply using priorities, threads reserve time on the processor
- Kernel scheduler is more complex
- A bit harder to reason about
- Works well with low-medium loads

Damn Hardware ...

- Kernel scheduler is only one thing that can cause a thread to take too long to get its work done
- If a thread is running code that moves data across the PCI bus and the bus gets locked by a device ....
- Welcome to my life ...

So ...

- Bottom half interrupt handlers have the highest priority
- Our application RT threads come next
- Then everything else
- If there a runnable RT thread, and a CPU not running a higher priority RT thread, then that thread will run

For the programmer ....

- Realtime audio needs RT threads to meet deadlines regardless of kernel activity
- Any moderately complex media application needs multiple threads
- Communication (and synchronization) between threads is a major issue

Next week

Thread Synchronization & Communication in Realtime Programming
(Lock free methods, etc.)
http://tu.linuxaudiosystems.com/