What do you have to do?

- **Required**
  - Adding preemption to your scheduler
    - You will heavily rely on clock interrupts in this project
    - All your minithreads code must now be made thread-safe
  - Sleeping with timeout
  - Multilevel feedback scheduling policy
    - Strict priority scheduling between levels and round-robin within a level, quanta doubling at each level
    - Feedback used to move threads between the queues

- **Optional**
  - Unreliable datagrams
A more detailed plan (1)

- Start receiving clock interrupts
  - Register interrupt handler
  - Start measuring the elapsed time

- Add preemption
  - Synchronize access to global structures
    - Your „system” code may now be interrupted at any time
    - Our method of choice: disabling interrupts
  - Switch threads in the interrupt handler
A more detailed plan (2)

- Add alarms
  - Create a structure to manage info about alarms
  - Use your software clock to measure time
  - Start firing alarms in the clock interrupt handler

- Add sleeping
  
  minithread_sleep_with_timeout()

  - Register alarms, block and unblock threads
A more detailed plan (3)

- Add multi-level feedback scheduling
  - Implement multilevel feedback queues
    - Use a regular queue as the underlying structure
    - Add a cyclic search
  - Extend your scheduler to use the new policy
    - Switch to the new data structure
    - Cycle through all the four levels (to avoid starvation)
    - Add feedback and move threads between levels
Installing the interrupt handler

- Register it during initialization

```c
typedef void (*interrupt_handler_t)(void*);
void minithread_clock_init(
    interrupt_handler_t clock_handler);
```

- How to declare it in your code

```c
void clock_handler(void* arg)
{
    ... 
}
```
The basics of interrupts (2)

- Remember that...
  - Initially interrupts are disabled, need to enable them
  - You can still receive interrupts while in the interrupt handler, so you should disable them temporarily
  - You must not spend much time inside the handler
    - Should not call system functions, print to screen etc. since they consume too much time
  - You definitely... **CANNOT BLOCK!**
The basics of interrupts (3)

- Disabling clock interrupts (what to call)

  ```c
  typedef int interrupt_level_t;
  #define ENABLED 1
  #define DISABLED 0
  interrupt_level_t set_interrupt_level(
      interrupt_level_t newlevel);
  ```

- A **strongly recommended** way to use the above

  ```c
  interrupt_level_t intlevel =
      set_interrupt_level(DISABLED);

  (here comes your code)

  set_interrupt_level(intlevel);
  ```
Interrupts and time

- Adjusting the frequency
  - Need to modify "interrupts.h"
    ```
    #define SECOND 1000000
    #define MILLISECOND 1000
    #define PERIOD (100*MILLISECOND)
    ```

- Measuring elapsed time
  - Don’t use system functions (they are way too slow)
  - Software clock: just count the interrupts
    ```
    extern long ticks;
    ```
More about interrupts (1)

- How are interrupts processed?
  - Always executed in the context of some thread... ...whichever happens to be currently running.

- What happens after the interrupt:
  - Current state is saved on the stack of the running thread
  - Handler is called
  - After the handler completes, the saved state is restored
More about interrupts (2)

- Interrupts and system calls
  - System libraries are not thread-safe...
    ...so interrupts are disabled (underneath, not by you) while the process is inside system calls.
  - What happens if e.g. a thread spends a lot of time printing to the screen?
    - Most interrupts are missed
    - Scheduler cannot promptly switch between processes
    - Software clock is drifting, alarms don’t fire on time
Adding synchronization (1)

- Why the need to synchronize
  - Clock interrupts could arrive at any time
  - Any thread might be preempted while reading or updating the structures of the scheduler itself...
  - ...so this way multiple threads could be updating the same structures at the same time!
  - Clock handler itself needs to access the same global structures (to make scheduling decisions)
Adding synchronization (2)

- **What not to use: spin locks**
  - Cannot use it in the interrupt handler
    - Any kind of active waiting would be too time consuming
    - And anyway... who’s going to enable the lock if it’s locked?

- **What to use: disabling interrupts**
  - A good, efficient method on uniprocessors
  - Critical section must be short!
  - Disabling interrupts unnecessarily will be penalized
  - Follow the recommended pattern of usage
Interrupts: Beware...

- Beware of the following...
  - Unmatched enabling / disabling
    - You could be called with interrupts disabled (enabling them will compromise system safety)
    - You should never let the application code run with interrupts disabled
  - Disabling interrupts unnecessarily
    - You should not disable them outside *minithreads*
  - Disabling interrupts for too long
Alarms: Implementing

- What you need to implement:

  ```c
  int register_alarm(
      int delay, void (*func)(void*), void *arg);
  void deregister_alarm(int alarmid);
  ```

- What you need underneath...

  - Some structure to keep information about alarms
  - Code in interrupt handler that fires alarms
    - Use the global variable ticks that you’re updating on every interrupt to calculate the elapsed time
Alarms: Using

- Issues with using alarms
  - Alarms are fired in the interrupt handler, therefore...
    - Interrupts are disabled at that time
    - You cannot spend much time in your callback
    - You cannot block
    - Alarm handler is called in the context of the currently executing thread...
      ...which most of the time will be different from the thread that registered the alarm.
Sleeping with timeout (1)

- What you need to implement:
  
  ```c
  void minithread_sleep_with_timeout(int delay);
  ```

- Semantics:
  
  - Block the caller (and relinquish the CPU)
    - So you don’t put him back on the ready queue
  
  - Wake up after the timeout expires
    - Make it runnable (put back on the ready queue), but not necessarily the current thread.
Sleeping with timeout (2)

- How to implement
  - You should use the alarm functions
  - You should use the semaphores rather than explicit `minithread_start` and `minithread_stop`
    - Yields a cleaner, more modular structure
  - Avoid race conditions
    - Should register alarm / start waiting atomically
Multilevel queues

- What to implement:

```c
typedef void* multilevel_queue_t;

multilevel_queue_t multilevel_queue_new(
    int number_of_levels);

int multilevel_queue_enqueue(
    multilevel_queue_t queue,
    int level, any_t item);

int multilevel_queue_dequeue(
    multilevel_queue_t queue,
    int level, any_t *item);

int multilevel_queue_free(
    multilevel_queue_t queue);
```
A new scheduling policy

- Level 1: round robin within a level
- Level 2
- Level 3
- Level 4

Priority scheduling between levels...

...but we're not always starting at the highest level (no starvation)
Changing scheduling policy (1)

- Choosing threads for execution
  - We cycle through all four levels (moving starting point for dequeue)
  - After a given number of quanta, move to next level
  - Spend 80 / 40 / 24 / 16 quanta in levels 0 to 3, respectively
  - While at a given level, look for threads to schedule starting from a corresponding queue (keep looking in the following levels if empty)
  - Skip to next level if a queue is empty
  - While queue nonempty, keep scheduling threads from this queue in a round-robin fashion
  - Assign 1 / 2 / 4 / 8 quanta at a time at levels 0 to 3, correspondingly
Changing scheduling policy (2)

- Adding priorities to threads
  - We need to extend the TCB to keep those
  - Use a thread’s priority when enqueueing it
  - Priority determines the queue...
    - ...and the queue determines time slice and the frequency with which a thread will be scheduled
  - Initially all threads get the highest priority
  - As time goes, thread priorities decrease
    - Switch to a lower priority when a thread has outrun its time slice (detect it in interrupt handler)
Changing scheduling policy (3)

- Adding aging
  - Need to lower the thread’s priority (in TCB)
  - Do it when changing the active thread
    - Keep the current priority as it is
      - When a thread is blocking (stop and semahores)
      - When a thread is yielding
    - Lower priority (if not the lowest)
      - When a thread has outrun its quanta
  - Priorities are never raised
    - Could you think of any other reasonable policies?
Grading

- **Correctness**
  - Beware of race conditions: synchronization!
  - Correct enabling and disabling of interrupts, follow our pattern
  - Cleaning threads and structures, avoiding memory leaks

- **Efficiency**
  - Disabling interrupts: only for a short time and only when it is indeed necessary
  - Processing in the interrupt handler should be fast!
  - Idle thread should not take up non-idle time
  - Consider using semaphores if necessary rather than polling

- **Elegance**
  - Your code should have a modular structure