

Threads & Context Switching

Note: Must commit to the **mp2** branch in your provided GitHub repository by assignment deadline.

Please read the entire document before you begin.

Introduction

The earliest computers were mainframes that lacked any definition of a kernel. Each job had a certain period of time on the machine and contained program and data on punched paper cards or magnetic tape. The machine would run until the program finished or crashed.

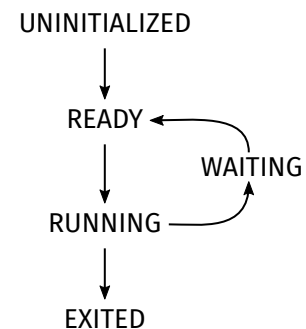
As machines grew more powerful, the time to run programs diminished and the time to manually switch to another program became a bigger source of delay than the runtime of programs. This motivated engineers to automate the process of switching from one process to another, which gave birth to multi-threaded systems.

The Thread Abstraction

The concept of a thread can be summarized as the smallest unit or object that can be scheduled to run on a CPU core. Threads contain all relevant data and code to run a series of instructions, on a given CPU core. In modern kernels like the Linux kernel, there is a specific mechanism running in the kernel called the *scheduler* that manages the queue of threads that are ready-to-run, waiting, blocking on I/O, sleeping, etc.

Our thread abstraction for this checkpoint deals with co-operative kernel threads. Co-operative threads voluntarily give up control when they are waiting for any arbitrary reason, e.g. waiting on device I/O, timer interrupt, user input, etc, and transfer control to another thread that's ready to run. At any given time, a thread is in one of five possible states:

- **UNINITIALIZED:** For a very brief period during its creation, a thread is in the UNINITIALIZED state, before transitioning to the READY state.
- **READY:** A thread in the READY state is ready to run, but is not currently running. All threads in the READY state must be on the ready-to-run list *ready_list*. The ready-to-run list must contain *only* threads in the READY state.
- **RUNNING:** A thread that is currently being executed by the CPU is in the RUNNING state. A thread that is RUNNING must not be on the ready-to-run list *ready_list*.
- **WAITING:** A thread waiting on a condition variable is in the WAITING state. When the condition is signalled, the thread transitions to the READY state. A thread can only enter the WAITING state by calling `condition_wait` function.
- **EXITED:** When a thread exits, it enters the EXITED state. Such a thread is not removed from the thread table until its parent calls `thread_join` or `thread_join_all`.



The frequent context switches between threads create the illusion of each thread executing on its own CPU (with a common memory).

Thread API

Each thread is represented by the struct below:

```
struct thread {
    struct thread_context context; // data required to run/resume a thread
    enum thread_state state; // current state of the thread
    int id; // thread id
    const char * name; // thread name for debugging purposes
    void * stack_base; // pointer to base of thread stack
    size_t stack_size; // size of thread stack
    struct thread * parent; // pointer to parent thread, that spawned said thread
    struct thread * list_next; // pointer to next thread in linked list
    struct condition * wait_cond; // pointer to condition variable thread is waiting on
    struct condition child_exit; // pointer condition variable for thread to signal parent on
    ↪ exit
};
```

The process will contain 4 threads in total:

Main Thread

This thread only spawns two other threads, the Star Trek thread and the Rule 30 thread. Afterwards, the thread will call `thread_join` and wait for the Star Trek thread to terminate. When the Star Trek thread terminates, it will signal the main thread through a condition variable and the main thread will exit. This will shutdown the entire program, including the Rule 30 thread.

Idle Thread

The idle thread exists to soak up any unused CPU cycles and to ensure that there is always a thread that is ready to run. It is not necessary to have an idle thread, but having one simplifies the scheduler. The idle thread is very simple: if there are no threads that are ready to run, it uses the RISC-V wait-for-interrupt instruction `wfi` to wait for an interrupt. On real hardware, this would stop the CPU clock to conserve power. On QEMU, this stops executing instructions until an interrupt occurs.

☞ *Check your understanding:* Why is it correct to stop and wait for an interrupt when there are no READY threads?

Star Trek

The Star Trek game is the same as in Checkpoint 1. However, now you can tell it which UART to use by passing the UART number as the `comno` parameter (see the new `main` function.)

Rule 30

Rule 30¹ is a ASCII screensaver that consists of cells that are either in an off or on state. The state of every pixel in a line depends on its neighbors on the previous line. The thread will periodically be woken up by the timer using a condition variable and draw the next iteration of cells based on the previous iteration.

Users of the thread API do not manipulate or refer to the thread structure shown above directly. Instead, they refer to threads by a thread identifier (TID). The thread API contains the following functions:

- `void thread_init(void)`
Globally initializes threading
- `int thread_spawn(const char * name, void (*start)(void *), void * arg)`
Creates and starts a new thread
- `int thread_join(int tid)`
Makes current thread wait for its child thread with id `tid` to terminate

¹https://en.wikipedia.org/wiki/Rule_30

- `int thread_join_any(void)`
Makes the current thread wait for any of its children to terminate
- `void thread_yield(void)`
Yields the CPU to another thread

For further details please read `thread.h`.

Condition Variable API

A condition variable is a synchronization primitive that (in our system) supports two operations:

- `condition_wait`: Suspends the calling thread until another thread calls `condition_broadcast`. Such a waiting thread is in the **WAITING** state. It is *not* safe to call this from an interrupt context.
- `condition_broadcast`: Signals (wakes up) all threads waiting on the condition. Signalling a thread puts it in the **READY** state. This function may be called from an interrupt context.

Condition variables often also support a *signal* operation, which wakes up *one* thread waiting on a condition. Our system does not use or implement this operation.

A condition variable is represented by the following C struct:

```
struct condition {
    const char * name; // name for debugging purposes
    struct thread_list wait_list; // list of threads waiting on condition
};
```

The condition variable has 3 functions that operate on the above struct

- `void condition_init(struct condition * cond, const char * name)`
Initializes a condition variable for use
- `void condition_wait(struct condition * cond)`
Causes the calling thread to sleep until awoken by a `condition_broadcast`
- `void condition_broadcast(struct condition * cond)`
Causes all sleeping threads on condition to wake up

For further details please read `thread.h`.

MP2 CP2 Assignment

The following files should be unchanged from the first checkpoint.

- `console.h`
- `console.c`
- `string.h`
- `string.c`
- `trap.c`
- `trapasm.s`

The following files have been updated from their CP1 versions and are in the release repository. You need to pull and merge these files into your `kern/` directory.

- `csr.h`
- `intr.h`

- `halt.c`
- `start.s`
- `trap.h`
- `serial.h`

The following files are new files that you don't need to edit. They are also in the release repository and need to be merged. They should not cause any merge conflicts. Read though these to understand what they are doing and what functionality the corresponding modules provide.

- `heap.h`
- `ezheap.c`
- `timer.h`
- `thread.h`

You will need to complete or modify the following files:

- `intr.c`
- `timer.c`
- `serial.c`
- `thread.c`
- `thrasm.s`
- `serial.c`

In each file, you will add your implementation where you find comments labeled `// FIXME`. The functions that you need to implement are also listed in the sections below.

Note : Please make sure that the files that you edit follow the function signatures in their respective `.h` files

Thread Function Implementations

You will to implement the following thread functions:

- `int thread_join(int tid)`

Waits for a specific child thread to exit

This function in `thread.c` waits for a specified child thread to exit. Argument `tid` is the TID of the child to wait for. If the calling thread is the parent of the specified child thread, `thread_join` waits for the child to exit and then returns the child TID. If there is no thread with the specified TID or the calling thread is not the parent of the specified thread, `thread_join` should return `-1`. There are two cases you must cover:

1. *Child already exited.* If the child has already exited, `thread_join` does not need to wait. It should return immediately.
2. *Child still running.* If the child is still running, the parent should wait on the condition variable `child_exit` in its own `struct thread`. Note that an existing child signals this condition in `thread_exit`.

In either case, the parent should release the resources used by the child thread by calling `recycle_thread`. See the function `thread_join_any` for inspiration. (The `thread_join_any` function waits for *any* of a thread's children to exit.)

- `void condition_broadcast(struct condition * cond)`

Wakes up all threads waiting on the condition variable

This function `thread.c` wakes up all threads waiting for a condition to be signalled. Waking up a thread entails:

- Changing its state from `WAITING` to `READY`,
- Placing it on the ready-to-run list, and
- removing it from the list of threads waiting on the condition.

After broadcasting the condition, the set of threads waiting on the condition should be empty.

- `void suspend_self(void)`
Suspends the execution of the current thread and switches to another ready thread
 This function in `thread.c` suspends the current thread, removing it from execution and switching to the next ready-to-run thread in the `ready_list`. The `suspend_self` function implements a simple round-robin scheduler. The thread being suspended, if it is still runnable, is inserted at the tail of `ready_list`, and the next thread to run is taken from the head of the list. Note that `suspend_thread` is called from `thread_yield`, `condition_wait`, and `thread_exit`. The calling thread may be `RUNNING`, `WAITING`, or `EXITED`. Only in the first case should your implementation of `suspend_self` place the current thread back on the ready-to-run list (and update its state accordingly).
 ☞ *Check your understanding:* Can the threads to be suspended and resumed ever be the same thread?
- `void _thread_setup(struct thread * thr, void * sp, void (s*start)(void * arg), void * arg)`
Setup the thread context
 This function in `threads.m.s` should initialize the `thread_context` structure, which is the first member of `struct thread`, but should not start executing the thread. The initial stack pointer and the entry function for the new thread are given by the `sp` and `start` arguments. The `_thread_setup` function should arrange for the thread to start execution in `start` with `arg` as the first argument when it is scheduled (it is passed to `_thread_switch` as the argument). Furthermore, `_thread_setup` should arrange it so that returning from `start` is equivalent to calling `thread_exit`. *Hint:* Examine carefully the two places where this function is called to understand how it is used.

Timer Function Implementations

You are required to implement or update the following functions:

- `void timer_intr_handler(void)`
Handles timer interrupts
 This function in `timer.c` should signal the `tick_10Hz` condition 10 times per second and the `tick_1Hz` condition once per second using `condition_broadcast`. The global `tick_10Hz_count` variable should count the number of times `tick_10Hz` was signaled, and `tick_1Hz_count` the number of times `tick_10Hz` was signaled. The timer on the virt device increments 10 million times per second; use the constant `MTIME_FREQ` in `timer.c` for this value.
- `void intr_handler(int code)`
 You will need to update the `intr_handler` function in `intr.c` to call the `timer_intr_handler` function when a timer interrupt occurs.

UART Function Implementations

You are required to implement the following UART functions:

- `void com_putc_async(struct uart * uart, char c):`
Writes a character to the UART asynchronously
 This function in `serial.c` writes the character `c` to the UART transmit ring buffer. If the transmit ring buffer is full, it must wait for the condition `txbuf_not_full` to be signaled. After writing the character to the buffer, it should enable the transmit interrupt.
 Hint: Think about whether this function should be executed in a critical section.
- `char com_getc_async(struct uart * uart):`
Reads a character from the UART asynchronously
 This function in `serial.c` reads a character from the UART receive buffer. If the receive ring buffer is empty, it must wait for the condition `rxbuf_not_empty` to be signaled. After reading the character from the ring buffer, it should enable the data ready interrupt.
 Hint: Think about whether this function should be executed in a critical section.

- `static void uart_isr(int irqno, void * aux):`

UART interrupt service routine

This function in `serial.c` needs to be modified from your CP1 implementation to signal the above two conditions where appropriate.

How to get things started

You can follow these steps to pull the assignment code from the release repo:

1. Switch to your MP2 branch with :

```
git switch mp2
```

2. Fetch the MP2 branch from the release repo:

```
git fetch release mp2
```

3. Merge release/mp2 into the local mp2 branch:

```
git merge --allow-unrelated-histories release/mp2
```

4. Handle any merge conflicts that occur, and conclude merge by running :

```
git add <names of files that caused merge conflicts>
git commit -m "<merge message>"
```

Since the new files that we're pulling are overwriting some lines in a way that conflicts with our local repository state, you will have to deal with merge conflicts.

Here's what a merge conflict will look like

```
struct trap_frame {
    uint64_t x[32]; // x[0] unused
<<<<<<< HEAD
    uint64_t mepc;
    uint64_t mstatus;
=====
    uint64_t mstatus;
    uint64_t mepc;
>>>>>>> 4da85a07aa26a73d8d6f346d08a2281bdf11d294
};
```

The line with the repeated "=" signs represent the split between the incoming change and the local state.

The lines above the equal sign and up to "<<<<<<< HEAD" are your local state

The lines below the equal sign and down to ">>>>>>> 4da85" are the incoming changes for this section

You have to decide which version to keep. You could choose the local version, you could choose the remote change or you could merge both manually into something equivalent.

After handling all such conflicts in every part of the file that you're working on you need to tell Git that the merge conflicts inside said file have been resolved.

You can do this by running

```
git add <conflicting filename here>
```

And when all merge conflicts are handled you can conclude the merge by running

```
git commit -m "<merge message here>"
```

Running and Testing

Your CP2 distribution included a file called `cp2-gold.elf` that demonstrates correct behavior. You can run it using the make target `run-cp2-gold`:

```
make run-cp2-gold
```

Once you have completed your implementation, you can run your code using QEMU by executing the following command:

```
make run-cp2
```

You can also run GDB by running

```
make debug-cp2
```

To view the output in a different terminal, use the screen command:

```
screen /dev/pts/N
```

where N is one of the numbers provided by QEMU on the first and second lines of output. This will ensure you can see the UART output properly. You should be able to open two separate terminal windows, one where you can play the Star Trek game, and one with the Rule 30 screen saver.

Tips

- Your RISC-V toolchain includes a command called `riscv64-unknown-elf-addr2line`, which allows you to map addresses to source file line numbers. This is very useful if your code generates an exception and you would like to know which line caused the problem. Read the manual for this command at <https://linux.die.net/man/1/addr2line>
- You can edit the Makefile to enable output of the `debug()` macro and the `trace()` macro by adding `-DDEBUG` and `-DTRACE` to `CFLAGS`, respectively. (These two flags are currently commented out in the Makefile.) See the provided sources files for examples of how these two macros are used. You may want to add these to your code to aid in debugging.
- In the provided code, you will also find an `assert` macro and `panic` function. The `assert` macro serves to document your assumptions about the state of the system at a given point in the program, especially useful when the following code assumes the asserted condition is true. If, during execution, the asserted condition is not true, the kernel will panic. This makes it easy to find bugs in your code that would otherwise be very difficult to localize.² The `panic` function is similar: It prints an error message and halts execution. Use it to catch conditions your code is not designed to handle. Take a look at `ezheap.c`, which implements a trivial memory manager. Note that it uses `panic` to report running out of memory.

Submission

You can submit as usual by pushing to your **mp2** branch. Submission on other branches will not be graded. You are also required to keep the directory structure of your branch the same, i.e. don't move files around. All files should be in the same directory as they were initially.

²Assertions helped us numerous times when debugging this MP!