Lecture 15, Feb 13, 2024

Threading

- *Concurrency*: switching between two or more tasks (interrupting the tasks to context switch)
  - The goal is to make progress on multiple things
- *Parallelism*: running two or more things independently at the same time
  - The goal is to run as fast as possible
- *Threads* are like processes, but the memory is shared
  - Registers, program counter, and stack are still independent
  - Address space is shared, so if one thread modifies memory, all other threads see it
  - To get memory specific to a thread, we need to specify *thread-local storage* (TLS)
  - One process can have multiple threads
- Due to fast context switching, threads can execute concurrently even with just a single CPU
- Threads are lighter than processes and faster to run because:
  - Code/data/heap is shared
  - Cheap creation (no need to copy resources like page tables)
  - Cheap context switching (no need to flush caches like the TLB)
- Threads live within an executing process (unlike processes which can execute independently); when a process dies, all its threads will die with it
  - When a thread dies, only its stack is removed from the process
  - So once the main thread dies, all other threads immediately stop executing
    - There is no such thing as orphan threads, but there can be zombie threads since their resources don't get released until you wait
  - There is no parent-child relationship for threads

```
int pthread_create(pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void*),
                   void* arg);
```

- To create a thread, use the `<pthread.h>` library
  - Arguments:
    - `thread`: output handle to a thread struct that will be populated
    - `attr`: thread attributes
    - `start_routine`: a function pointer to start execution at
    - `arg`: additional argument to pass to `start_routine`
  - Returns 0 on success or error number otherwise
  - Unlike `fork()`, the new thread starts executing at a specified different location instead of the current location
- `int pthread_join(pthread_t thread, void **retval);` will wait for the thread to terminate before returning
  - This is the equivalent of `wait()` for threads
  - The pointer `**retval` is set to the location of the `void*` returned by the thread function
  - Note calling this more than once on a thread leads to undefined behaviour!
- `void pthread_exit(void *retval);` will terminate a thread early with the specified return value
  - This is called implicitly when the function of a thread returns
- *Joinable* threads are the default kind, which wait for a thread to call `pthread_join()` before releasing its resources
  - We can *detach* threads via `int pthread_detach(pthread_t thread);`
    - These threads are non-joinable, so calling join on them is undefined behaviour
    - Calling detach on a detached thread is undefined behaviour
  - Calling `pthread_exit(NULL);` in the main thread will keep the process alive until all other threads have exited
    - This is useful for detached threads since we can't join them

- Thread attributes such as the stack size can be set explicitly using `pthread_attr` functions