Ch3: Thread

Threads

- An **execution unit** that is part of a **process**
    - A process can have multiple threads, which can execute at the same time
- Thread is an execution units managed by OS scheduler
- Thread is a **lightweight process**:
    - Different threads within the same process share the same memory space

Address Space

- Logical address: generated by the CPU, i.e. virtual address
- Physical address: address seen by the memory unit
- The access of data is controlled by OS. Different process's "address 0" would be different, so that different process could run at the same time. But within the same process, different threads' "address 0" would be identical

Comparison

Process	**Thread**
Heavyweight	Lightweight
Own memory space	Share memory space
slow inter communication	fast inter communication
expensive context switching	less expensive context switching
don't share memory	share memory if within the same process

Multi-threads

- For parallel execution, higher CPU utilization in the presence of I/O blocking

Parallelism & Concurrency

- Parallelism: Running multiple threads/processes in parallel over different CPU cores
- Concurrency: Running multiple threads/processes at the same time, even with single core, by interleaving their executions

User Thread vs Kernel Threads

- User Threads: Thread management done by user-level threads library (e.g. POSIX *Pthreads*, Java threads)
- Kernel Threads: Managed by the kernel
- User threads are mapped to kernel threads

Multithreading Models

- One-to-one: Each user-level thread mapped to a kernel-level thread
  - Different user threads can execute in parallel simultaneously
  - Expensive in resource: Creating a user thread requires a kernel thread
- Many-to-one: Many user threads mapped to one kernel thread
  - one thread can access kernel at a time
  - multiple thread can no run in parallel
- Many-to-many: Allows many user level threads to be mapped to many kernel threads
  - Allow OS to create a sufficient number of kernel threads
- Combined Model: Coexistence of one-one and many-many

User Thread & Kernel Thread

User Thread	Kernel Thread
Implemented by the users	Implemented inside OS
Oriented to users, users do not worry about the implementation details	Oriented to kernel, OS manages the threads (e.g. scheduling and synchronization)
Possible to schedule threads without entering kernel mode (many-to-one), but preemption is impossible in this case	Full functionalities
One thread perform blocking, then the entire process is blocked (in many-to-one mapping)	If one kernel thread perform blocking, other threads could still continue execution
Lower context switch overhead	Large context switch overhead

Thread Cancellation

- Terminating a thread before it has finished (e.g. pthread_cancel())
- Two general approaches:
  - Asynchronous Cancellation: Terminates the target at any time
  - Deferred Cancellation: Allows the target thread to be cancelled at certain cancellation point

Thread pools

- Create a number of threads in a pool where they await to do tasks
- Advantages:
  - Usually faster than creating a new thread
  - Allows to bound the number of threads in the application

Pthread

- Thread calls:
    - Pthread_create: create a new thread
    - Pthread_exit: terminate the calling thread
    - Pthread_join: wait for a specific thread to exit
    - Pthread_yield: release the CPU to let another thread run
    - Pthread_attr_init: create and initialize a thread's attribute strucure
    - Pthread_attr_destroy: remove a thread's attribute structure
- Thread Creation:

```
pthread_t p1, p2;

pthread_create(&p1,     // &p1 specifies the thread;
               NULL,    // NULL specifies the attribute;
               mythread, // mythread specifies the thread function;
               "A");    // "A" is the argument
```

- Thread Join:
    - "Joining" is one way to accomplish synchronization between threads.

```
int pthread_join(pthread_t, void **retval);
pthread_join(p1,    // p1 specifies the thread
             NULL); // NULL is the return value
```

    - It's logic error to attempt simultaneous multiple joins on the same worker thread
        - some thread may already joined other threads, so they'll wait forever

Context Switch

- Context of a thread:
    - CPU registers
    - Stack
- Context of a process:
    - CPU registers
    - Entire address space (code, data, heap, stack...)
    - Many other resources: kernel resource, I/O, files

Context Protection for interrupts

- When an interrupt occurs, before switching to the ISR, registers are saved to the stack (hardware assisted), including PC(program counter), SP(stack pointer)
- Reason: Protect the register
- Using two stacks for context switching, one stack per task.