Threads

• Chapter 2 Modern Operating Systems, Andrew Tanenbaum

• Also: Chapter 11 Advanced Programming in Unix Environment, By Richard Stevens
If you want to do one task

- Start one process
If you want to do two task “concurrently”

- Start two processes
  - Maybe P1 forks P2
  - and P3…PN etc if more than two tasks

- Problem:
  - fork is expensive
  - cold-start penalty
If P1 and P2 want to talk to each other?

- E.g. access the same data or synchronize?
- Two different address spaces
  - Need to use IPC
  - shared memory, pipes, sockets, signals
- Problem
  - kernel transitions are expensive
  - May need to copy data
    - user—>kernel—>user
  - Inter-process Shared memory is a pain to set up.
Option 1: Event-driven programming

- Make one process do all the tasks
- Busy loop polls for events and executes tasks for each event
- No IPC needed
- Length of the busy loop determines response latency
- Stateful event responses complicate the code
  - What if $i^{th}$ occurrence of event 1 effects the $j^{th}$ event processing?

```c
while(1)
{
    if (event 1) do task 1;
    if (event 2) do task 2;
    ...
    if (event N) do task N;
}
```
Option 2: Use threads

- Multiple threads of execution per process

- Each thread has its own
  - Program counter
  - Stack, stack pointer
  - Registers

- All threads share
  - one virtual address space
    - code, heap and static data

- Lower context switching overhead
- No IPC
  - Zero data transfer cost
  - Only need inter-thread synchronization
Other Shared and non-shared components

- Shared components
  - Open descriptors (files, sockets etc)
  - Signals and Signal handlers

- Not shared
  - Thread ID
  - Errno
  - Priority
Example: A word processor with three threads

- First thread handles keyboard input
- Second thread handles screen display
- Third thread handles saving the document to disk
Example: a multi-threaded web server

- A dispatcher thread waits for and accepts network connections
- Several worker threads
  - Each worker processes one network connection concurrently
Disadvantages of Threads

• Shared State!
  • Global variables are shared between threads.
  • Accidental data changes can cause errors.

• Threads and signals don’t mix well
  • Common signal handler for all threads in a process
  • Which thread to signal? Everybody!
  • Royal pain to program correctly.

• Lack of robustness
  • Crash in one thread will crash the entire process.

• Some library functions may not be thread-safe
  • Library Functions that return pointers to static internal memory. E.g. gethostbyname()
  • Less of a problem these days.
Two types of threads: user-level and kernel-level

**User-level threads**
- User-level libraries provide multiple threads,
- OS kernel does not recognize user-level threads
- Threads execute when the process is scheduled

**Kernel-level threads**
- OS kernel provides multiple threads per process
- Each thread is scheduled independently by the kernel’s CPU scheduler
Hybrid Implementations

Multiplexing user-level threads within each kernel-level threads
Local Thread Scheduling

- Next thread is picked from among the threads belonging to the current process.
- Each process gets a timeslice from kernel.
- Then the timeslice is divided up among the threads within the current process.

- Local scheduling can be implemented with either:
  - Kernel-level threads OR
  - User-level threads.

- Scheduling decision requires only local knowledge of threads within the current process.

- For example, say process timeslice may be 50ms, and each thread within the process runs for 5 msec/CPU burst.
Global Thread scheduling

- Next thread to be scheduled is picked up from ANY process in the system.
  - Not just the current process

- Timeslice is allocated at the granularity of threads
  - No notion of per-process timeslice

- Global scheduling can be implemented only with kernel-level threads
  - Picking the next thread requires global knowledge of threads in all processes.

- For example each thread runs for 10msec per CPU burst
Thread Creation and termination

• Creation
  • int pthread_create( pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);

• Two ways to perform thread termination
  1. Return from initial function.
  2. void pthread_exit(void * status)

• Waiting for child thread in parent
  • pthread_join(…)
  • equivalent to waitpid
Threaded program - example

// shared counter to be incremented by each thread
int counter = 0;

main()
{
    pthread_t tid[N];

    for (i=0;i<N;i++) {
        /*Create a thread in thread_func routine*/
Pthread_create(&tid[i], NULL, thread_func, NULL);
    }

    for(i=0;i<N;i++)
        /* wait for child thread */
Pthread_join(tid[i], NULL);
}

void *thread_func(void *arg)
{

    /* unprotected code - race condition*/
    counter = counter + 1;

    return NULL; // thread dies upon return
}
pthread synchronization operations

- Mutex operation
  - `pthread_mutex_init(…)`
  - `pthread_mutex_lock(…)`
  - `pthread_mutex_unlock (…)`
  - `pthread_mutex_trylock (…)`

- Condition variables
  - `pthread_cond_wait (…)`
  - `pthread_cond_signal (…)`
  - `pthread_cond_broadcast (…)`
  - `pthread_cond_timedwait (…)`