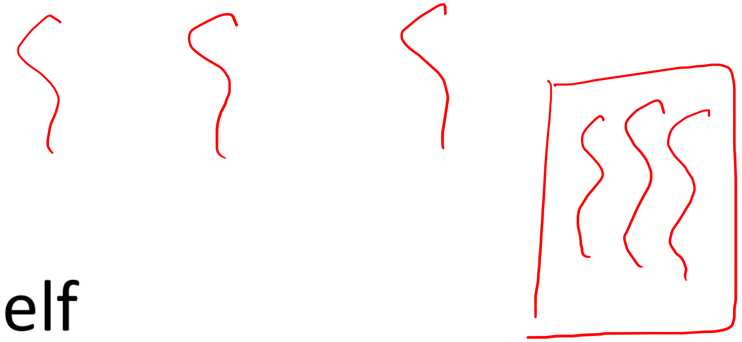
Design and Engineering of Computer Systems

Lecture 8: Threads

Mythili Vutukuru

IIT Bombay

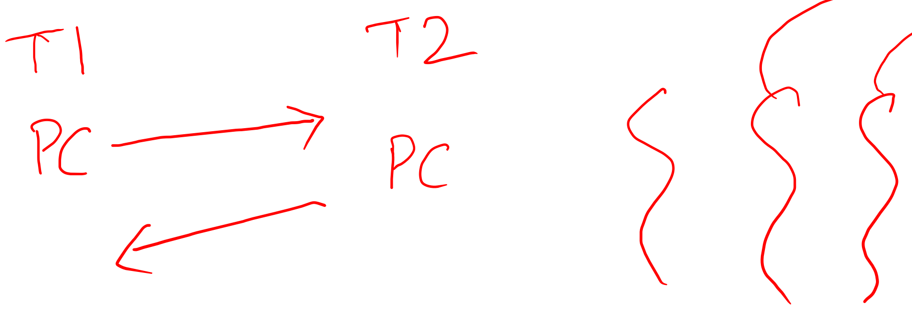
What are threads?



- A process may want to run multiple copies of itself
  - If one copy blocks due to blocking system call, another copy can still run
  - Multiple copies can run in parallel on multiple CPU cores
- Example: a web server should handle multiple requests at a time
- One option: have multiple processes running the same program
  - Disadvantage: too much memory consumed by identical memory images

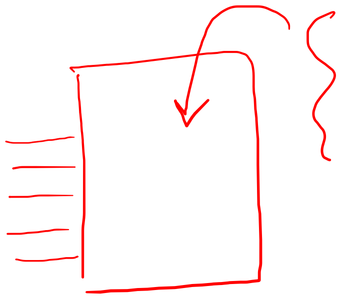
- Better option: use threads = light weight processes
- A process can create multiple threads (default: single thread)
  - Multiple threads share same memory image of process, saves memory
  - Threads run independently on same code (if one blocks, another can still run)
  - Threads can run in parallel on multiple cores at same time

Understanding threads



- Multiple threads of a process share the same code, global/static variables allocated at compile time, and heap
- Threads execute independently on the process code
  - Each thread has its own separate CPU context
  - Each thread's PC is pointing to different instructions
- As a result, each thread has a separate stack
  - Each thread calls functions independently, has to store context separately
- Inside the OS, each thread has its separate thread control block (TCB), context stored in TCB when not running
  - TCBs of threads belonging to same process share some common information of the PCB (e.g., details of memory image, I/O connections)

POSIX threads



- In Linux, POSIX threads (pthreads) library allows creation of multiple threads in a process
- Each thread is given a start function where its execution begins
  - Threads execute independently from parent after creation
  - Parent can wait for threads to finish (optional)
- Threads created with pthreads treated as separate entities by OS scheduler, can run concurrently on same CPU core, or in parallel on multiple cores
  - Kernel-level threads (OS is aware of them)
- Several such threading libraries exist
  - Not all threading libraries guarantee independent scheduling at the OS level, may exist only for ease of programming for user (user-level threads)

```
void f1() {
    …
}

void f2() {
    …
}

main() {
    …
    pthread_t t1, t2
    pthread_create(&t1, .., f1,..)
    pthread_create(&t2, .., f2,..)
    …

    pthread_join(t1, ..)
    pthread_join(t2, ..)

}
```

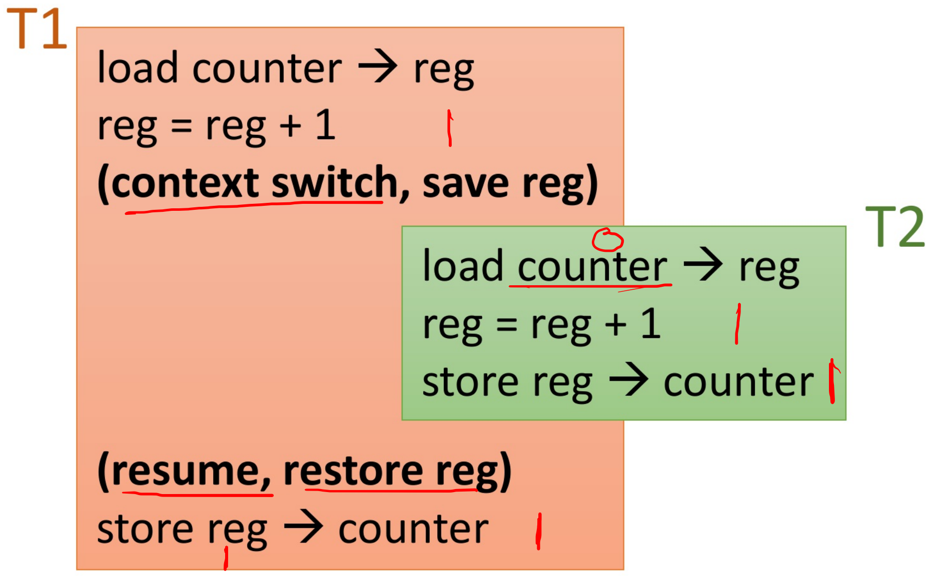
Shared data access

- Threads of a program share global/static variables and heap data
- What happens when threads concurrently access shared data?
- Example: two threads created, each increments shared counter 1000 times
  - We expect final counter value to be 2000
  - In reality: value slightly smaller than 2000
- Concurrent access of shared data is tricky!

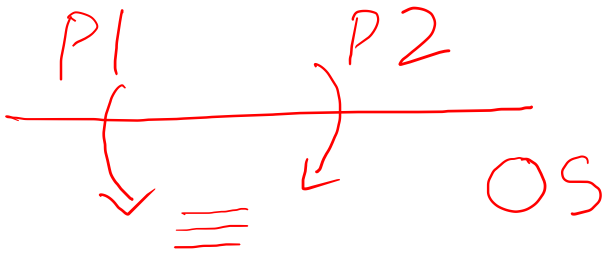
```
int counter;

void start_fn() {

    for(int i=0; i < 1000; i++)
        counter = counter + 1
}

main() {
    counter = 0

    pthread_t t1, t2
    pthread_create(&t1,..,  start_fn, ..)
    pthread_create(&t2, .., start_fn,..)

    pthread_join(t1, ..)
    pthread_join(t2, ..)

    print counter
}
```

Understanding shared data access

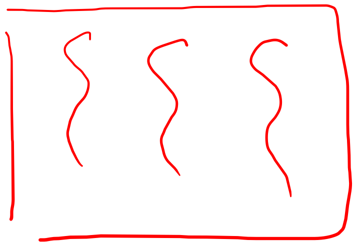
- The C code “counter = counter + 1” is compiled into multiple instructions
  - Load counter variable from memory into register
  - Increment register
  - Store register back into memory of counter variable
- What happens when two threads run this line of code concurrently?
  - Counter is 0 initially
  - T1 loads counter into register, increment reg
  - Context switch, register (value 1) saved
  - T2 runs, loads counter 0 from memory
  - T2 increments register, stores to memory
  - T1 resumes, stores register value to counter
  - Counter value rewritten to 1 again
  - Final counter value is 1, expected value is 2



Race conditions, critical sections



- Incorrect execution of code due to concurrency is called race condition
  - Due to unfortunate timing of context switches, atomicity of data update violated
  - Not just counters, can happen with any data structures
  - User code cannot disable interrupts or context switches
- Race conditions happen when we have concurrent execution on shared data
  - Threads sharing common data in memory image
  - Processes in kernel mode sharing OS data structures
  - (Single-threaded processes in user mode do not share any data)



- We require mutual exclusion on some parts of code
  - Concurrent execution by multiple threads should not be permitted
- Parts of program that need to be executed with mutual exclusion for correct operation are called critical sections
  - Present in multi-threaded programs, OS code

Using locks

- Locks are special variables that provide mutual exclusion
  - Provided by threading libraries
  - Can call lock/acquire and unlock/release functions on a lock
- When a thread T1 acquires a lock, another thread T2 cannot acquire same lock
  - Execution of T2 stops at the lock statement
  - T2 can proceed only after T1 releases the lock
- Acquire lock → critical section → release lock ensures mutual exclusion in critical section

```
int counter;
pthread_mutex_t m;

void start_fn() {

   for(int i=0; i < 1000; i++) {
      pthread_mutex_lock(&m)
      counter = counter + 1
      pthread_mutex_unlock(&m)
}

main() {
   counter = 0

   pthread_t t1, t2
   pthread_create(&t1,.., start_fn, ..)
   pthread_create(&t2, .., start_fn,..)

   pthread_join(t1, ..)
   pthread_join(t2, ..)

   print counter
}
```

Implementing locks

- What is happening inside the lock/unlock functions? How are locks implemented?
- Example of incorrect lock implementation
  - Use bool isLocked to indicate lock status
  - To acquire lock, a thread waits until lock is free and then proceeds to acquire it

```
bool isLocked = false

void acquire_lock() {
    while(isLocked); //wait
    isLocked = true
}

void release_lock() {
    isLocked = false
}
```

PC

T1

```
while(isLocked);
isLocked = true

CRITICAL SECTION

isLocked = false
```

T2

```
while(isLocked);
while(isLocked);
while(isLocked);
while(isLocked);
isLocked = true

CRITICAL SECTION
```

T1

while(isLocked);
(context switch, PC saved)

(resumes execution)
isLocked = true
CRITICAL SECTION

T2

while(isLocked);
isLocked = true
CRITICAL SECTION

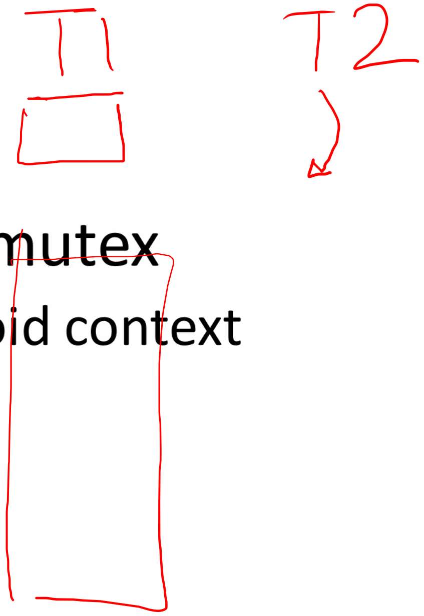
Hardware atomic instructions

- Need a way to check a variable and set its value atomically
  - No context switch between checking lock variable to be free and setting it to be true
  - But user programs have no control over context switches
- Solution: use hardware atomic instructions
- Example: test-and-set sets value of variable and returns old value
- Simple lock can be implemented using test-and-set instruction
  - If test-and-set(isLocked, true) returns true, it means lock is held by someone, wait
  - If test-and-set(isLocked, true) returns false, lock was free and has been acquired
- Single CPU instruction is both checking lock to be free and setting it to be true atomically, cannot be interrupted in between

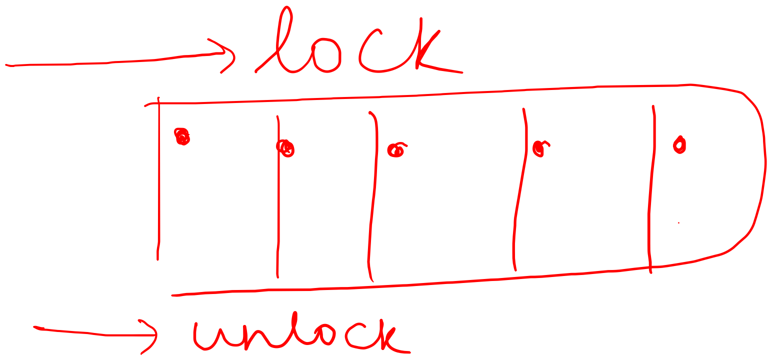
```
bool isLocked = false

void acquire_lock() {
   while(test-and-set(isLocked, true) == true); //wait
}
```

Spinlock vs. sleeping mutex

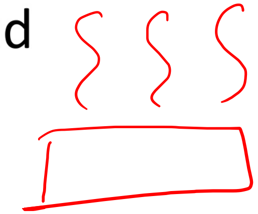
- Simple lock implementation seen here is a spinlock
  - If thread T1 has acquired lock, and thread T2 also wants lock, then T2 will keep spinning in a while loop till lock is free
- Another implementation option: thread can go to sleep (be blocked) while waiting for lock, saving CPU cycles
  - OS blocks waiting thread, context switch to another thread/process
  - Such locks are called (sleeping) mutex
- Threading libraries provide APIs for both spinlocks and sleeping mutex
  - Better to use spinlock if locks are expected to be held for short time, avoid context switch overhead
  - Better to use sleeping mutex if critical sections are long



Guidelines for using locks



- When writing multithreaded programs, careful locking discipline
  - Protect each shared data structure with one lock
  - Locks can be coarse-grained (one big fat lock) or fine-grained (many smaller locks)
  - Any thread wanting to access shared data must acquire corresponding lock before access, release lock after access
- Good practice to acquire locks for both reading and writing data
  - Why locks for reading? We do not want to read incorrect data while another thread is concurrently updating the data
  - Some libraries provide separate locks for reading and writing, allowing multiple threads to concurrently read data if no other thread is writing



- If using third-party libraries in multi-threaded programs, check if the library is thread-safe
  - Thread-safe implementations work correctly with concurrent access

Summary

- In today's lecture:
  - Threads for concurrency and parallelism
  - Race conditions, critical sections
  - Locks: usage and implementation
  - Hardware atomic instructions
- Try to write simple multi-threaded programs, observe race conditions, and fix them using locks
  - Pthreads API is simple and easy to use