Lecture 4: Synchronization Primitives

Created	@January 17, 2023 10:16 AM
Type	Lecture
Materials	
Reviewed	☐

Real world concurrency

- Millions of drivers on highway at once
- Student does homework while watching Netflix

Motivation for Concurrency

- CPU trend: Same speed, but multiple cores
- Goal: write application that fully utilize many cores

Concurrency - Option 1

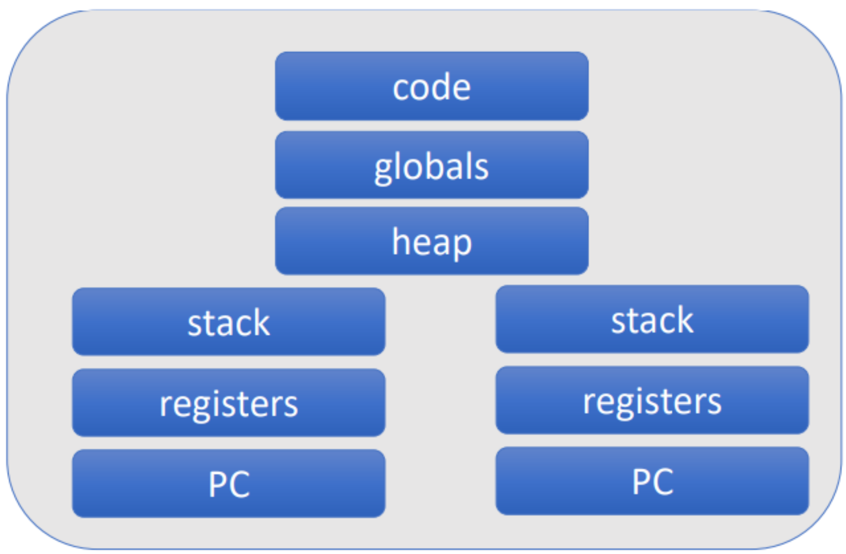
- Build apps from many communicating processes
- Communicate through ***message passing***
  - No shared memory
- Pros: If one process crashes, other processes unaffected
- Cons: High communication overheads, expensive context switching

Concurrency - Option 2

- New abstraction: ***thread***
- Multiple threads in a process

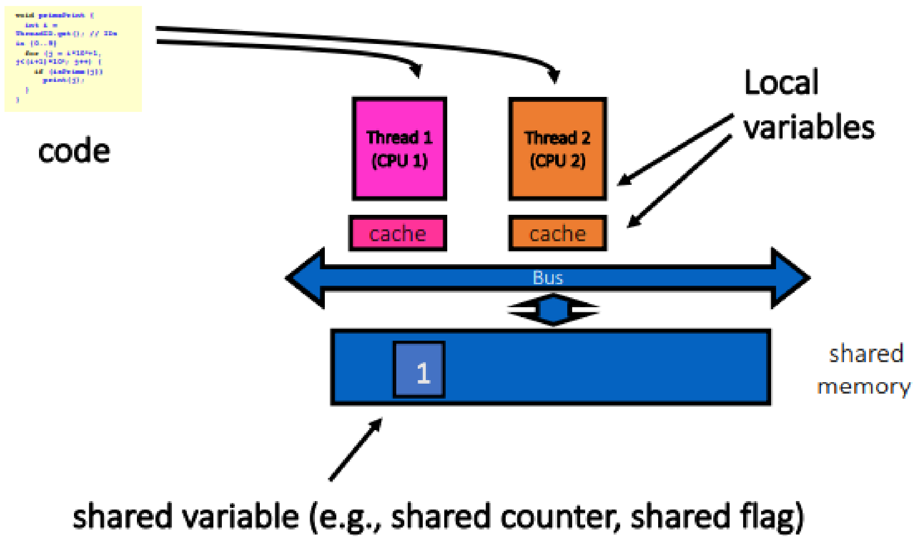
- Threads are like processes except
    - Multiple threads in the same process share an address space
    - Communicate through **<u>shared address space</u>**
    - If one thread crashes, the entire process, including other threads, crashes

Two threads in a process:



- In general
    - Processes provide separation
        - Memory separation (no shared data)
    - Threads do not provide separation
        - Threads share memory (shared data)

Where things reside



Shared Data

- Pros: Many threads can read/write it
- Cons: Many threads can read/write it, Can lead to ***data races***

Data Race

- Unexpected/unwanted access to shared data
- Result of ***interleaving*** of thread executions
- Program must be correct for all interleavings

A Single Line of Code is not Atomic

- `a = a + 1` is in reality:
  - Load a from memory into register
  - Increment register
  - Store register value in memory
- Instruction sequence may be interleaved

Non-determinism

Concurrency leads to non-deterministic results

- Different results with the same inputs
- Race conditions

Whether bug manifests or not depends on CPU schedule

How to program?

- Imagine schedule is mallicious
- Assume will pick bad ordering at some point

Basic Approach to Multithreading

1. "Divide" work amongst multiple threads and
2. Share data
    a. **Global variables and heap**
    b. Not local variables, not read-only variables
3. Where is shared data accessed?
    a. Shared data is accessed in ***critical section***

Critical Section

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

Want three instructions to execute as an interruptable group (we want them to be atomic)

- Need ***mutual exclusion*** for critical sections
- If thread A is in critical section C, thread B can't enter C
- Ok if other processes do unrelated work

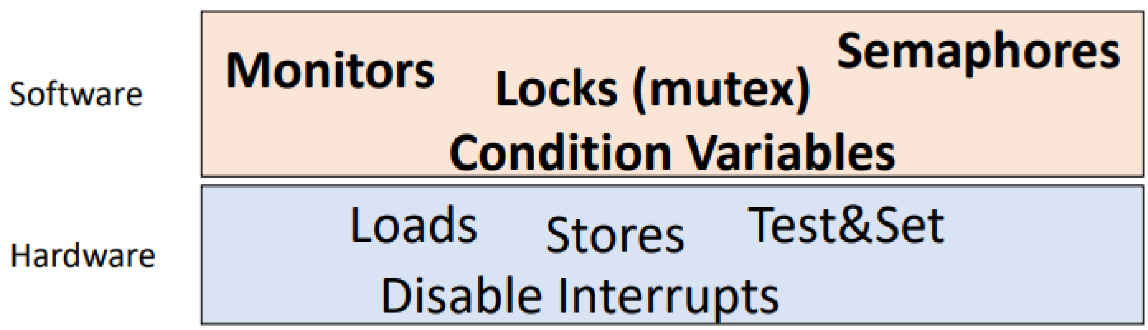
Mutual Exclusion

- Prevents simultaneous access to a shared resource
- How can we achieve mutual exclusion?
    - Library support (pthreads)
    - Implementation of synchronization primitives

This works because in the critical section, no other thread can change data

Synchronization

- Build higher-level synchronization primitives in OS
- Operations that ensure correct ordering of instructions across threads



POSIX Thread Libraries (pthreads)

- Thread API for C/C++
- User-level library: `#include <pthread.h>`
    - Compile and link with `-pthread`
- `pthread_create(), pthread_exit(), pthread_join()`

`pthread_create()`

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

- Create thread in `thread`
- Run `start_routine` with arguments `arg`

`pthread_exit()`

`void pthread_exit(void *retval);`

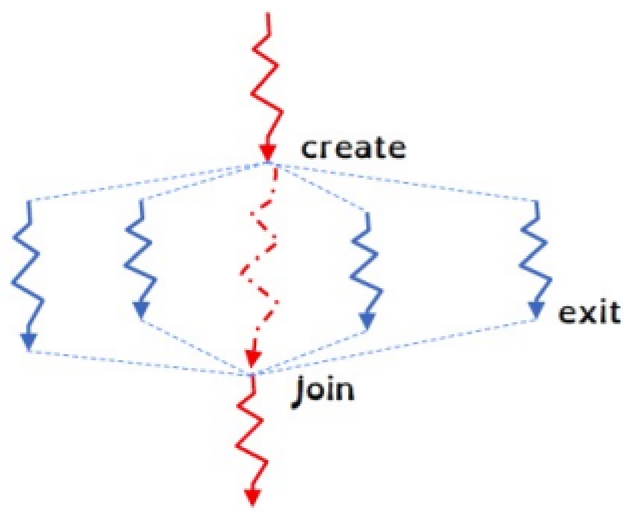
- Terminate calling thread
- Returns a value via retval

`pthread_join()`

`int pthread_join(pthread_t thread, void **retval);`

- Join with a terminated thread
- Waits for the thread specified by `thread` to exit
- If retval is not `NULL` then `pthread_join()` copies the exit status of the target thread into the locations pointed to by retval

Fork-Join Pattern for threads



Main thread creates (forks) collection of sub-threads passing them args to work on, and the joins with them, collecting the results.

Fork-join example

```
int count;
void *mythread(void *arg) {
```

```
    int j;
    for (j = 0; j < 1000000; j++){
      count +=1;
    }
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    count = 0;
    pthread_create(&p1, NULL, mythread, NULL);
    pthread_create(&p2, NULL, mythread, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("%d \n", count);
}
```

pthreads: Locks

- `pthread_mutex_lock(mutex)`
- `pthread_mutex_unlock(mutex)`
- If lock is held by another thread, block
- If lock is not held by another thread
  - Acquire lock
  - Proceed

Counting example revisited

```
pthread_mutex_t count_mutex;
int count;
void *mythread(void *arg) {
    int j;
    for (j = 0; j < 1000000; j++){
      pthread_mutex_lock(&count_mutex);
      count +=1;
      pthread_mutex_unlock(&count_mutex);
    }
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    count = 0;
    pthread_create(&p1, NULL, mythread, NULL);
    pthread_create(&p2, NULL, mythread, NULL);
```

```
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("%d \n", count);
}
```

Deadlocks

- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help

```
Thread A
1 lock(object1)
2 lock(object2)
3 //do stuff
4 unlock(object2)
5 unlock(object1)
  ...
```

```
Thread B
1 lock(object2)
2 lock(object1)
3 //do stuff
4 unlock(object1)
5 unlock(object2)
...
```

Code example

```
pthread_mutex_t lock1;
pthread_mutex_t lock2;
void *a_func(void *arg) {
    long j;
    for (j = 0; j < 100000000; j++) {
        pthread_mutex_lock(&lock1);
        pthread_mutex_lock(&lock2);
        printf("A");
        pthread_mutex_unlock(&lock2);
        pthread_mutex_unlock(&lock1);
    }
    return NULL;
}

void *b_func(void *arg) {
    long j;
    for (j = 0; j < 100000000; j++) {
        pthread_mutex_lock(&lock2);
        pthread_mutex_lock(&lock1);
        printf("B");
```

```
        pthread_mutex_unlock(&lock1);
        pthread_mutex_unlock(&lock2);
    }
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t a, b;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_create(&a, NULL, a_func, NULL);
    pthread_create(&b, NULL, b_func, NULL);
    pthread_join(a, NULL);
    pthread_join(b, NULL);
    printf("End!\n");
}
```

Condition Variables

- used when thread A needs to wait for an event done by thread B
- A waits until a certain conditions is true
  - Test condition, if condition not true, call `pthread_cond_wait()`
  - A blocks until condition is true
- At some point B makes the conditions true
  - Then B calls `pthread_cond_signal()`, which unblocks

Interface

- `pthread_cond_init(pthread_cond_t *cv, pthread_condattr_t *cattr)`
  - Initialize the conditional variable, cattr can be NULL
- `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex)`
  - Block thread until condition is true, and atomically unblock mutex
- `pthread_cond_signal(pthread_cond_t *cv)`
  - Unblock one thread at random that is blocked by the condition variable
- `pthread_cond_broadcast(pthread_cond_t *cv)`

- Unblock all threads that are blocked on the condition variable pointed to by cv

Condition Variable Example

```
pthread_cond_t is_zero;
pthread_mutex_t mutex;
int shared_data = 100;

void *thread_func(void *arg){
    while(shared_data > 0) {
        pthread_mutex_lock(&mutex);
        shared_data--;
        printf("%d ", shared_data);
        pthread_mutex_unlock(&mutex);
    }

    printf("Signaling main\n");
    pthread_cond_signal(&is_zero);
    return NULL;
}
int main (void){
    pthread_t tid;
    void * exit_status;
    int i;

    pthread_cond_init(&is_zero, NULL);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&tid, NULL, thread_func, NULL);

    pthread_mutex_lock(&mutex);
    printf("Start waiting in main\n");

    while(shared_data!=0)
        pthread_cond_wait(&is_zero, &mutex);
    pthread_mutex_unlock(&mutex);

    printf("Done waiting in main!\n");
    pthread_join(tid, &exit_status);
    return 0;
}
```