CS 111
Operating Systems Principles
Section 1E Week 4

Tengyu Liu

Question Answering

- Any question regarding
  - Threads vs. processes
  - Race condition
  - Critical section
  - Atomic operation
  - Synchronization
  - Locks

Project 2A Overview

- Write a program to demonstrate race condition and ways to avoid it
  - Part 1: each thread adds 1 to a shared variable, then subtracts 1 from it
  - Part 2: each thread inserts a node to a shared linked list, then remove a node from it
- If there is no race condition, we expect
  - Part 1: shared variable == 0
  - Part 2: shared linked list is empty

Project 2A – shared variable (part 1)

- You are given an add function

```
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        *pointer = sum;
}
```

- Initialize a shared variable, and run a number of threads that
  - Adds 1 to the variable for X times
  - Adds -1 to the variable for X times
- How will you notice that a race condition occurred?

Project 2A – shared variable (part 1)

- You are given an add function

```
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        *pointer = sum;
}
```

- Initialize a shared variable, and run a number of threads that
  - Adds 1 to the variable for X times
  - Adds -1 to the variable for X times
- How will you notice that a race condition occurred?
  - The final value of the shared variable will be nonzero

A thread is different from a process

- A thread is part of a process
- A process can have multiple threads
- Threads share the same memory space within a process, where processes each have its own separate memory space
- Thread management is very cheap compared to process management

Multithreading

- Initially, main() comprises only a single thread.
- Create a thread that runs a function: pthread_create()
- Wait on all other threads to complete: pthread_join()
- Time each run for each thread: clock_gettime()

Multithread API

- **`int pthread_create`**`(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
    - Creates a thread that runs [start_routine] with arguments [arg]
    - On success
        - Returns 0
        - `*thread` is thread identifier
    - On error
        - Returns an error number
        - `*thread` is left undefined

Multithread API

- `int pthread_create(`**`pthread_t *thread`**`, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
  - Output parameter
  - If call succeeded, contains the identifier of the created thread
  - Will be used for subsequent `pthread` calls

Multithread API

- `int pthread_create(pthread_t *thread,` **`const pthread_attr_t *attr`**`, void *(*start_routine) (void *), void *arg);`
  - Attribute for the created thread
  - Can be set to NULL for default values
  - Includes
    - Detached or joinable
    - Scheduling inheritance
    - Scheduling policy
    - Stack size
    - Stack address …

Multithread API

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,` **`void *(*start_routine) (void *)`**`, void *arg);`
  - The C routine to be executed on thread creating

Multithread API

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *),` **`void *arg`**`);`
  - A single argument to be passed to `start_routine`
  - Pass by reference
  - Can be set to NULL

Multithread API

- **int pthread_join**(pthread_t *thread, void **retval);
  - Blocks until thread exits
  - On success
    - Returns 0
  - On error
    - Returns error code

Multithread API

- `int pthread_join(`**`pthread_t *thread`**`, void **retval);`
  - Thread identifier for the thread we are waiting for

Multithread API

- `int pthread_join(pthread_t *thread, `**`void **retval`**`);`
  - Return value of thread we are waiting on
  - Can be set to NULL

Multithread API

- Example:
  - Create 4 threads that each prints “hello” to screen
  - Print “done” when all threads finish

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* thread_func(void* arg) {
    printf("hello\n");
    return 0;
}

int main()
{
    // create pthread_t objects
    pthread_t *threads = malloc((sizeof(pthread_t)) * 4);
    // create 4 threads that run thread_func
    for (int i = 0; i < 4; i++)
        pthread_create(&threads[i], NULL, thread_func, NULL);
    // wait for each thread to finish
    for (int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);
    printf("done\n");
    return 0;
}
```

Multithread API

- Example:
  - Create 4 threads that each prints "hello" to screen
  - Print "done" when all threads finish
- Your job
  - For each thread, add 1 to the shared variable for #iter times
  - Then, add -1 to the shared variable for #iter times
  - Observe the shared variable: what do you expect?

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* thread_func(void* arg) {
    printf("hello\n");
    return 0;
}

int main()
{
    // create pthread_t objects
    pthread_t *threads = malloc((sizeof(pthread_t)) * 4);
    // create 4 threads that run thread_func
    for (int i = 0; i < 4; i++)
        pthread_create(&threads[i], NULL, thread_func, NULL);
    // wait for each thread to finish
    for (int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);
    printf("done\n");
    return 0;
}
```

Project 2A – the main steps (part 1)

- Now that you’ve witnessed a race condition, you will be asked to create as many as possible
  - How do you modify the add() function to ensure things go wrong?

Project 2A – shared variable (part 1)

- Now that you’ve witnessed a race condition, you will be asked to create as many as possible
  - How do you modify the add() function to ensure things go wrong?
    - Force the thread to yield at the worst possible time

Project 2A – shared variable (part 1)

- Now that you've witnessed a race condition, you will be asked to create as many as possible
  - How do you modify the add() function to ensure things go wrong?
    - Force the thread to yield at the worst possible time

```
int opt_yield;
void add(long long *pointer, long long value) {
        long long sum = *pointer + value;
        if (opt_yield)
                sched_yield();
        *pointer = sum;
}
```

  - Your final value should be further away from 0.

Project 2A – shared variable (part 1)

- Then you need to counter the race condition with three different mechanics
  - Compare-and-swap (CAS) add
  - Spin-locks
  - Mutex locks
- Compare their performances

Protection 1: Mutex Lock

- Mutex lock
  - **Mut**ual **ex**clusion lock
  - A mutex lock can have one of two states: free or locked
- There are two **atomic operations** you can do on a lock
  - lock and unlock

Mutex lock state	Operation	Behavior
Free	Lock	Set mutex lock state to locked
Free	Unlock	Nothing happens
Locked	Lock	Blocks until mutex lock becomes free, then set mutex lock state to locked
Locked	Unlock	Set mutex lock state to free

Protection 1: Mutex Lock

- Pseudo-code inside thread_func:
  - `mutex_lock.lock()`
  - `// critical section`
  - `mutex_lock.unlock()`

Atomic operation guarantees you have the exclusive access of the critical section

Synchronization API – mutex

- **`int pthread_mutex_init`**`(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
  - Initializes a mutex
  - On success
    - Returns 0
  - On error
    - Returns error code
  - Note: if no attr is required, you can use
    - `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
    - To statically declare and initialize a mutex.

Synchronization API – mutex

- `int pthread_mutex_init(`**`pthread_mutex_t *restrict mutex`**`, const pthread_mutexattr_t *restrict attr);`
    - Mutex object

Synchronization API – mutex

- `int pthread_mutex_init(pthread_mutex_t *restrict mutex,` **`const pthread_mutexattr_t *restrict attr`**`);`
  - Attributes include
    - Type (deadlocking, deadlock-detecting, recursive, …)
    - Robustness (error handling capacity)
    - Sharing across processes
    - How a thread behaves when a higher priority thread wants the mutex
    - …
  - Use NULL for default attributes

Synchronization API – mutex

- **`int pthread_mutex_destroy`**`(pthread_mutex_t *mutex);`
  - Destroys a mutex
  - On success
    - Returns 0
  - On error
    - Returns error code
    - If mutex is not released, returns EBUSY

Synchronization API – mutex

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - On success
    - Return 0
  - On error
    - Returns error code
  - If the mutex is not available
    - `pthread_mutex_lock` blocks
    - `pthread_mutex_trylock` returns EBUSY immediately

Protection 2: Spin Lock

- Constantly checking if the lock is occupied
- Pseudo-code:
  - while (lock is not successful);
  - // critical section
  - unlock;

Non-blocking expression, but has to be atomic.

Synchronization API – spin lock

- `type __sync_lock_test_and_set(type *ptr, type value)`
  - Writes `value` to `*ptr`, and returns previous content of `*ptr`
  - Atomic operation
- `void __sync_lock_release(type *ptr)`
  - Sets content of `*ptr` to 0, and therefore releases the lock
- How to implement a spin lock with these two functions?

Synchronization API – spin lock

- `type __sync_lock_test_and_set(type *ptr, type value)`
  - Writes `value` to `*ptr`, and returns previous content of `*ptr`
  - Atomic operation
- `void __sync_lock_release(type *ptr)`
  - Sets content of `*ptr` to 0, and therefore releases the lock
- How to implement a spin lock with these two functions?

```
static int lock;

while( __sync_lock_test_and_set( &lock, 1 ) );
// critical section
__sync_lock_release( &lock );

```

Synchronization API – spin lock

- `type __sync_lock_test_and_set(type *ptr, type value)`
  - Writes `value` to `*ptr`, and returns previous content of `*ptr`
  - Atomic operation
- `void __sync_lock_release(type *ptr)`
  - Sets content of `*ptr` to 0, and therefore releases the lock
- How to implement a spin lock with these two functions?
  - If lock is available (`lock == 0`), we break from while loop and set `lock` to 1

```c
static int lock;

while( __sync_lock_test_and_set( &lock, 1 ) );
// critical section
__sync_lock_release( &lock );

```

Synchronization API – spin lock

- `type __sync_lock_test_and_set(type *ptr, type value)`
  - Writes `value` to `*ptr`, and returns previous content of `*ptr`
  - Atomic operation
- `void __sync_lock_release(type *ptr)`
  - Sets content of `*ptr` to 0, and therefore releases the lock
- How to implement a spin lock with these two functions?
  - If lock is available (`lock == 0`), we break from while loop and set `lock` to 1
  - When someone else tries to acquire `lock`, he is stuck at the loop until `__sync_lock_release` is called

```c
static int lock;

while( __sync_lock_test_and_set( &lock, 1 ) );
// critical section
__sync_lock_release( &lock );

```

Protection 3: Compare and Swap

- Set value to a variable in an atomic operation
- `type __sync_val_compare_and_swap(type *ptr, type oldval, type newval)`
  - If the content of `*ptr` is `oldval`, then write `newval` to `*ptr` and return `oldval`

Compare performance of protection methods

- Measure and report run time
- **`int clock_gettime`**`(clockid_t clk_id, struct timespec *tp)`
  - Gets the time of a specific clock identified by `clk_id`
  - On success
    - Returns 0
  - On error
    - Returns -1
    - Sets errno

clock_gettime

- `int clock_gettime(`**`clockid_t clk_id`**`, struct timespec *tp)`
  - Specifies which clock we are getting time from
    - CLOCK_REALTIME: System-wide realtime clock. Setting this clock requires appropriate privileges.
    - CLOCK_MONOTONIC: Clock that cannot be set and represents monotonic time since some unspecified starting point.
    - CLOCK_PROCESS_CPUTIME_ID: High-resolution per-process timer from the CPU.
    - CLOCK_THREAD_CPUTIME_ID: Thread-specific CPU-time clock.
  - Which clock do we want?

clock_gettime

- `int clock_gettime(clockid_t clk_id, `**`struct timespec *tp`**`)`
  - Output parameter

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

What to do next?

- Run your program with different parameters

What to do next?

- Run your program with different parameters
- For each run, print to stdout
  - the name of the test (add-none for the most basic usage)
  - the number of threads (from --threads=)
  - the number of iterations (from --iterations=)
  - the total number of operations performed
  - the total run time (in nanoseconds)
  - the average time per operation (in nanoseconds).
  - the total at the end of the run (0 if there were no conflicting updates)

What to do next?

- Run your program with different parameters
- For each run, print to stdout
    - **add-none** ... no yield, no synchronization
    - **add-m** ... no yield, mutex synchronization
    - **add-s** ... no yield, spin-lock synchronization
    - **add-c** ... no yield, compare-and-swap synchronization
    - **add-yield-none** ... yield, no synchronization
    - **add-yield-m** ... yield, mutex synchronization
    - **add-yield-s** ... yield, spin-lock synchronization
    - **add-yield-c** ... yield, compare-and-swap synchronization

What to do next?

- Run your program with different parameters
- For each run, print to stdout
  - the total number of operations performed
    - #thread x #iteration x 2
  - the average time per operation (in nanoseconds).
    - Total run time / total number of operations

What to do next?

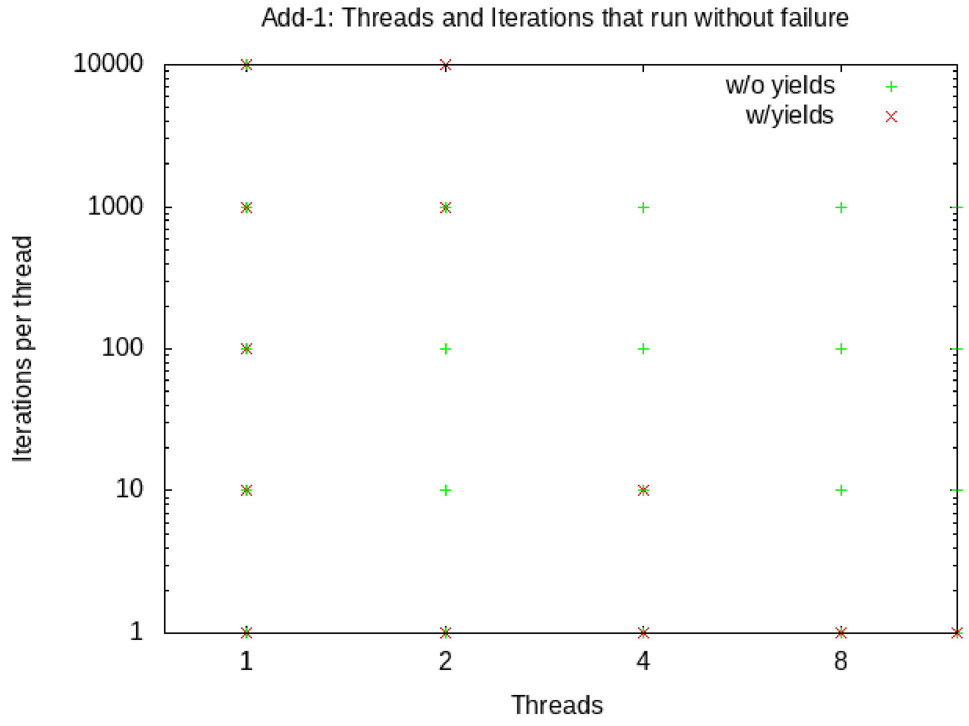
- Run your program with different parameters
- For each run, print to stdout
- Examine the results and answer questions in your README file
- Run gnuplot lab2_add.gp to generate graphs for submission

Answering questions and generating graphs

- Q 2.1.1
  - Run your program with a small number of threads (for example, 2 or 4) and a range of iterations (100, 1000, 10000, 100000)
    - 4 runs in total
  - Observe when errors have emerged
  - You should observe that error is more likely to happen with larger #iterations

Answering questions and generating graphs

- lab2_add-1.png
  - Run your program with and without the --yield option for threads (2,4,8,12) and iterations (10, 20, 40, 80, 100, 1000, 10000, 100000).
    - 64 runs in total
    - lab2_add-1.png will plot a dot for a successful run
    - lab2_add-1.png will NOT plot a dot for an error
    - You should see less dots to the top-right corner
    - Example graph has wrong #threads and #iters



Answering questions and generating graphs

- lab2_add-2.png and Q 2.1.2
  - Run with and without --yield for threads (2,8) and iterations (100,1000,10000,100000)
    - 16 runs in total
  - lab2_add-2.png will plot time/op vs. #iterations
  - You should observe that for the same experiment, with --yield is much slower than without --yield
  - Why?

Answering questions and generating graphs

- lab2_add-3.png and Q 2.1.3
  - Run without --yield, #threads=1, with different values of #iters
  - lab2_add-3.png plots average time/op vs. #iters
  - You can choose your own #iters values, just make sure that the values can show a trend in the graph
    - At least 4 values that span over a large range
  - You should expect the average time drops as #iters increases
  - Why?

Answering questions and generating graphs

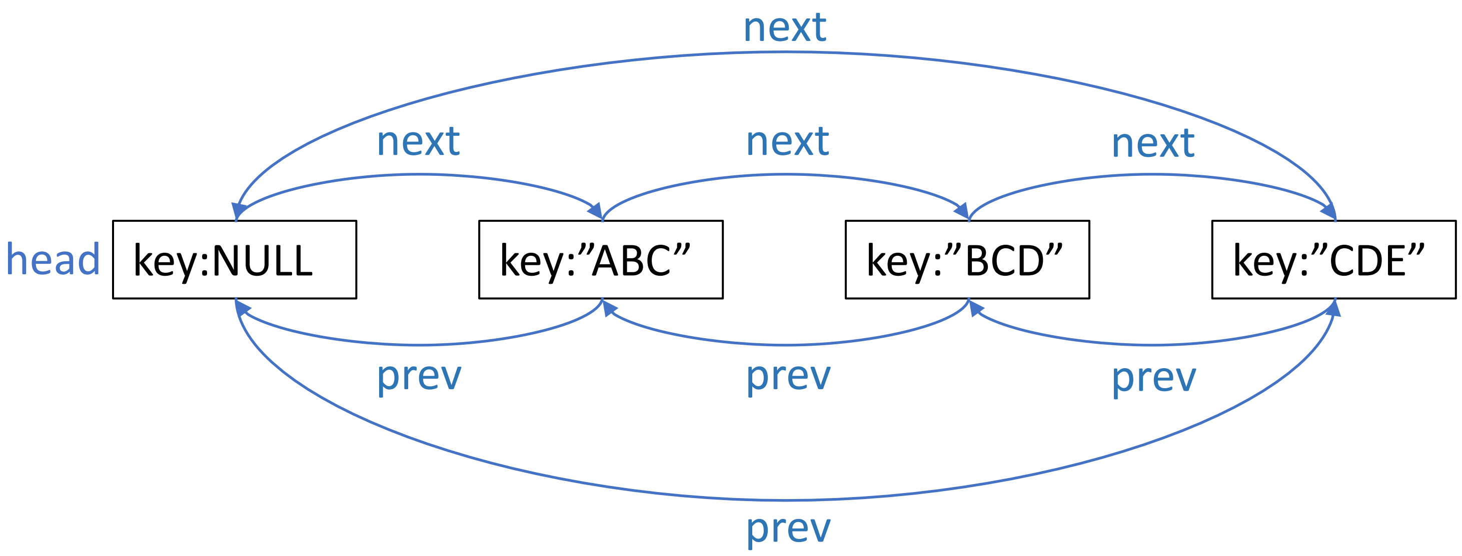
- lab2_add-4.png
  - Run with --yield, for threads (2,4,8,12) and protection method (none, mutex, spin-lock, compare-and-swap), with #iters=10000 (1000 for spin-lock)
    - 16 runs in total
    - lab2_add-4.png contains whether each run succeeds
    - Think about what you should observe

Answering questions and generating graphs

- lab2_add-5.png and Q 2.1.4
  - Run without --yield, for threads (1,2,4,8,12) and protection method (none, mutex, spin-lock, compare-and-swap), with #iters=10000 (1000 for spin-lock)
    - 20 runs in total
  - lab2_add-5.png plots average time/op vs. #threads
  - You should observe operations become slower with more threads
  - Why?

Project 2A – shared list (part 2)

```
struct SortedListElement {
    struct SortedListElement *prev;
    struct SortedListElement *next;
    const char *key;
};
```



- Operations
  - SortedList_insert(list, element)
    - Inserts an element to a list
    - Maintain sort order
  - SortedList_lookup(list, key)
    - Search sorted list for a key
    - Return the correct element
  - SortedList_delete(element)
    - Remove element from its list
  - SortedList_length(list)
    - Count elements in a list
    - Check all prev/next pointers for corruption

Project 2A – shared list (part 2)

- Given a shared linked list, each thread
  - Inserts #iter elements to the list
  - Get the list length
  - Deletes the elements inserted by current thread
- What would a race condition look like?

Project 2A – shared list (part 2)

- Given a shared linked list, each thread
  - Inserts #iter elements to the list
  - Get the list length
  - Deletes the elements inserted by current thread
- What would a race condition look like?
  - List has length > 0 after all threads finish
  - Key not found for delete
  - Dereferencing a null pointer (segfault)

Project 2A – shared list (part 2)

- Given a shared linked list, each thread
  - Inserts #iter elements to the list
  - Get the list length
  - Deletes the elements inserted by current thread
- What would a race condition look like?
  - List has length > 0 after all threads finish
  - Key not found for delete
  - Dereferencing a null pointer (segfault)
- Catch all of these, print error messages and return an error

Yielding and protection in part 2

- There are three places that you can place a yield
  - Insert
  - Delete
  - Lookup
  - --yield=[i,d,l,id,il,dl,idl]
- You need to implement two protection methods
  - Mutex lock
  - Spin lock

Protecting list operations: insert

- Step 1: acquire lock
- Step 2: list_insert(element)
- Step 3: release lock

Protecting list operations: delete

- Two operations:
  - element = list_lookup(key)
  - list_delete(element)
- Where do we put protection?

Protecting list operations: delete

- Two operations:
  - element = list_lookup(key)
  - list_delete(element)
- Where do we put protection?
  - Option 1:
    - Acquire lock
    - Lookup and delete
    - Release lock
  - Option 2:
    - Acquire lock
    - Lookup
    - Release lock
    - Acquire lock
    - Delete
    - Release lock

Protecting list operations: delete

- Two operations:
  - element = list_lookup(key)
  - list_delete(element)
- Where do we put protection?
  - Option 1:
    - Acquire lock
    - Lookup and delete
    - Release lock
  - Option 2:
    - ~~Acquire lock~~
    - ~~Lookup~~
    - ~~Release lock~~
    - ~~Acquire lock~~
    - ~~Delete~~
    - ~~Release lock~~

Note

- Project 2A takes a lot of resources to run.
- When many students are running their code on lnxsrv09, it is likely that your performance will vary significantly between runs, and therefore failing the sanity check script.
- To avoid this, start early and finish early.