Multithreading Programming II

Content

- Review Multithreading programming
- Race conditions
- Semaphores
- Thread safety
- Deadlock

Review: Resource Sharing

- Access to shared resources need to be controlled to ensure deterministic operation
- Synchronization objects: mutexes, semaphores, read/write locks, barriers
- Mutex: simple single lock/unlock mechanism
    - **int** pthread_mutex_init(pthread_mutex_t ∗mutex, **const** pthread_mutexattr_t ∗ attr);
    - **int** pthread_mutex_destroy(pthread_mutex_t ∗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);

Review: Condition Variables

- Lock/unlock (with mutex) based on run-time condition variable.
- Allows thread to wait for condition to be true.
- Other thread signals waiting thread(s), unblocking them
  - **int** pthread_cond_init(pthread_cond_t ∗cond,
const pthread_condattr_t ∗attr);
  - **int** pthread_cond_destroy(pthread_cond_t ∗cond);
  - **int** pthread_cond_wait(pthread_cond_t ∗cond,
pthread_mutex_t ∗mutex);
  - **int** pthread_cond_broadcast(pthread_cond_t ∗cond);
  - **int** pthread_cond_signal(pthread_cond_t ∗cond);

Multithreaded Programming

- OS implements scheduler – determines which threads execute when
- Scheduling may execute threads in arbitrary order
- Without proper synchronization, code can execute non-deterministically
- Suppose we have two threads:
  - 1 reads a variable,
  - 2 modifies that variable
- Scheduler may execute
  - 1, then 2,
  - or 2 then 1
- Non-determinism creates a *race condition* – where the behavior/result depends on the order of execution

Race conditions

- Race conditions occur when multiple threads share a variable, without proper synchronization
- Synchronization uses special variables, like a mutex, to ensure order of execution is correct
- Example: thread T1 needs to do something before thread T2
  - condition variable forces thread T2 to wait for thread T1
  - producer-consumer model program
- Example: two threads both need to access a variable and
  - modify it based on its valuesurround access and modification with a mutex
  - mutex groups operations together to make them *atomic* – treated as one unit

Example

Consider the following program race.c:

```
unsigned int cnt = 0;
void ∗count ( void ∗arg ) { /∗ thread body ∗/
  int i ;
  for ( i = 0; i < 100000000; i ++)
      cnt ++;
  return NULL ;
}
int main ( void ) {
  pthread_t tids [ 4 ] ;
  int i ;
  for ( i = 0; i < 4; i ++)
    pthread_create (& tids [ i ] , NULL, count , NULL ) ;
  for ( i = 0; i < 4; i ++)
     pthread _join ( tids [ i ] , NULL ) ;
  printf ( " cnt=%u \ n " , cnt ) ;
  return 0;
}
```

What is the value of cnt?

[Bryant and O'Halloran. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.]

Example Results

Ideally, should increment cnt 4 × 100000000 times, so cnt = 400000000. However, running our code gives:

```
athena% ./race.o
cnt=137131900
athena% ./race.o
cnt=163688698
athena% ./race.o
cnt=163409296
athena% ./race.o
cnt=170865738
athena% ./race.o
cnt=169695163
```

So, what happened?

Race Conditions

- C not designed for multithreading
- No notion of atomic operations in C
- Increment cnt++; maps to three assembly operations:
  - load cnt into a register
  - increment value in register
  - save new register value as new cnt
- So what happens if thread interrupted in the middle?

Race condition!

Race Conditions

Let’s fix our code:

```
pthread_mutex_t mutex;
unsigned int cnt = 0;

void ∗count ( void ∗arg) {/∗ thread body ∗/
  int i;
  for ( i = 0; i < 100000000; i ++) {
    pthread_mutex_lock(&mutex );
     cnt++;
    pthread_mutex_unlock(&mutex );
  }
  return NULL ;
}
int main ( void ){
  pthread_t tids [4];
  int i;
  pthread_mutex_init(&mutex, NULL);
  for (i =0; i<4; i++)
    pthread_create(&tids[i], NULL, count, NULL);
  for (i =0; i<4; i++)
   pthread _join(tids[i], NULL);
  pthread_mutex_destroy(&mutex );
  printf ("cnt=%u\n " ,cnt );
  return 0;
}
```

Race Conditions

- Note that new code functions correctly, but is much slower
- C statements not atomic – threads may be interrupted at assembly level, in the middle of a C statement
- Atomic operations like mutex locking must be specified as atomic using special assembly instructions
- Ensure that all statements accessing/modifying shared variables are synchronized

Semaphores

- *Semaphore* – special nonnegative integer variable s, initially 1, which implements two atomic operations:
  - P(s) – wait until s> 0, decrement s and return
  - V(s) – increment s by 1, unblocking a waiting thread
- Mutex –
  - locking calls P(s) and
  - unlocking calls V(s)
- Implemented in `<semaphore.h>,` part of library `rt`, not `pthread`

Using Semaphores

- Initialize semaphore to `value`:

  ```
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  ```

- Destroy semaphore:

  ```
  int sem_destroy(sem_t *sem);
  ```

- Wait to lock, blocking:

  ```
  int sem_wait(sem_t *sem);
  ```

- Try to lock, returning immediately (0 if now locked, −1 otherwise):

  ```
  int sem_trywait(sem_t *sem);
  ```

- Increment semaphore, unblocking a waiting thread:

  ```
  int sem_post(sem_t *sem);
  ```

Producer and Consumer Revisited

- Use a semaphore to track available slots in shared buffer
- Use a semaphore to track items in shared buffer
- Use a semaphore/mutex to make buffer operations synchronous

Producer and Consumer Revisited

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

sem_t mutex, slots, items;

#define SLOTS 2
#define ITEMS 10

void* produce(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++)
  {
    sem_wait(&slots);
    sem_wait(&mutex);
    printf("produced(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&items);
  }
  return NULL;
}

void* consume(void* arg)
{
  int i;
  for (i = 0; i < ITEMS; i++) {
    sem_wait(&items);
    sem_wait(&mutex);
    printf("consumed(%ld):%d\n",
           pthread_self(), i+1);
    sem_post(&mutex);
    sem_post(&slots);
  }
  return NULL;
}

int main()
{
  pthread_t tcons, tpro;

  sem_init(&mutex, 0, 1);
  sem_init(&slots, 0, SLOTS);
  sem_init(&items, 0, 0);

  pthread_create(&tcons,NULL,consume,NULL);
  pthread_create(&tpro,NULL,produce,NULL);
  pthread_join(tcons,NULL);
  pthread_join(tpro,NULL);

  sem_destroy(&mutex);
  sem_destroy(&slots);
  sem_destroy(&items);
  return 0;
}
```

Other Challenges

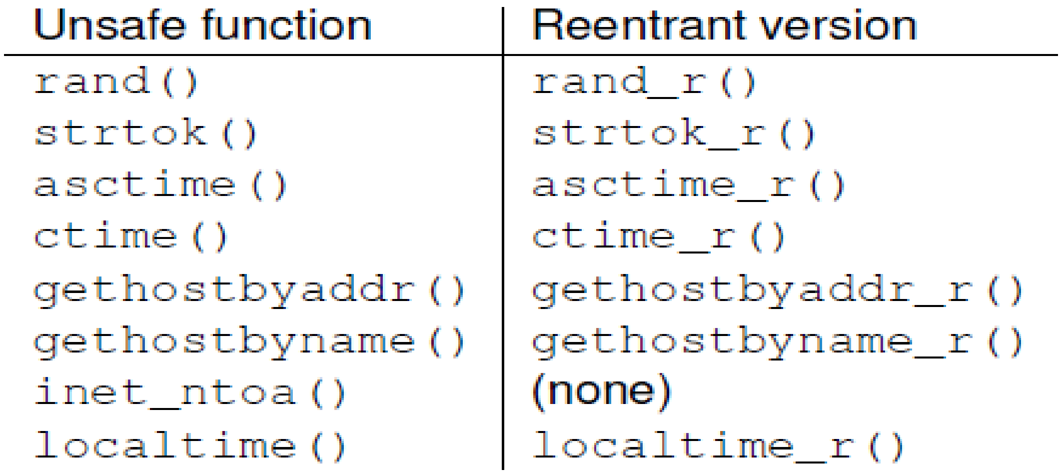
- Synchronization objects help solve race conditions
- Improper use can cause other problems
- Some common issues:
    - thread safety and reentrant functions
    - deadlock
    - starvation

Thread Safety

- Function is *thread safe* if it always behaves correctly when called from multiple concurrent threads
- Unsafe functions fail in several categories:
  - accesses/modifies unsynchronized shared variables
  - functions that maintain state using static variables – like `rand(), strtok()`
  - functions that return pointers to static memory – like `gethostbyname()`
  - functions that call unsafe functions may be unsafe

Reentrant functions

- Reentrant function – does not reference any shared data when used by multiple threads
- All reentrant functions are thread-safe
- Reentrant versions of many unsafe C standard library functions exist:



Unsafe function	Reentrant version
`rand()`	`rand_r()`
`strtok()`	`strtok_r()`
`asctime()`	`asctime_r()`
`ctime()`	`ctime_r()`
`gethostbyaddr()`	`gethostbyaddr_r()`
`gethostbyname()`	`gethostbyname_r()`
`inet_ntoa()`	(none)
`localtime()`	`localtime_r()`

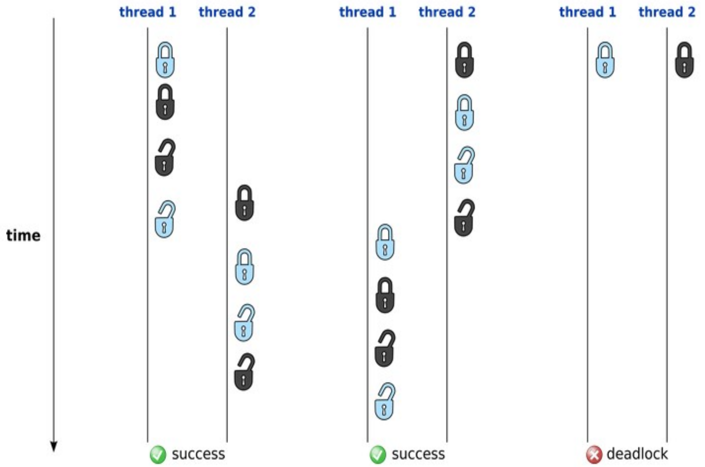
Thread safety

To make your code thread-safe:

- Use synchronization objects around shared variables
- Use reentrant functions
- Use synchronization around functions returning pointers to shared memory (*lock-and-copy*):
    1. lock mutex for function
    2. call unsafe function
    3. dynamically allocate memory for result; (deep) copy result into new memory
    4. unlock mutex

Deadlock

```
#include <assert.h>
#include <pthread.h>

static void * simple_thread(void *);

pthread_mutex_t mutex_1= PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_2= PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_t tid = 0;

pthread_create(&tid, 0, &simple_thread, 0); // create a thread

pthread_mutex_lock(&mutex_1);           // acquire mutex_1
pthread_mutex_lock(&mutex_2);           // acquire mutex_2
pthread_mutex_unlock(&mutex_2);         // release mutex_2
pthread_mutex_unlock(&mutex_1);         // release mutex_1

pthread_join(tid, NULL);

return 0;
}

static void * simple_thread(void * dummy)
{
pthread_mutex_lock(&mutex_2);          // acquire mutex_2
pthread_mutex_lock(&mutex_1);          // acquire mutex_1
pthread_mutex_unlock(&mutex_1);        // release mutex_1
pthread_mutex_unlock(&mutex_2);        // release mutex_2

return NULL;
}
```



Deadlock

- Deadlock – happens when every thread is waiting on another thread to unblock
- Usually caused by improper ordering of synchronization objects
- Tricky bug to locate and reproduce, since schedule-dependent
- Can visualize using a progress graph – traces progress of threads in terms of synchronization objects

Deadlock

- Defeating deadlock extremely difficult in general
- When using only mutexes, can use the "*mutex lockordering rule*" to avoid deadlock scenarios:

  A program is deadlock-free if,

  for each pair of mutexes (s, t) in the program,

  each thread that uses both s and t simultaneously

  locks them in the same order

Starvation and Priority Inversion

- Starvation is similar to deadlock
- Scheduler never allocates resources (e.g. CPU time) for a thread to complete its task
- Happens during priority inversion
    - example:
        - highest priority thread T1 waiting for low priority thread T2 to finish using a resource,
        - while thread T3, which has higher priority than T2, is allowed to run indefinitely
    - thread T1 is considered to be in starvation

Thank you,