Overview

● Administrative
  * HW 4 - makefile available
  * HW 4 - any questions

● Topics:
  * Threads Synchronization
  * 3 Mechanisms: mutex, semaphore, condition variable
  * Motivating Example: Producer-Consumer
  * Solution using 3 mechanisms
  * Effect of Threads on rest of POSIX
    - Signal

● Readings: Ch. 10 (pp. 365-400), Ch. 8.3.1-2 (pp. 304-307)

● Exercises: 10.1 - 10.7
What Is synchronization?

- *Sometime we use literal meaning-*
  - To take place at the same time instant
  - 1. To cause events to appear to be synchronous
  - Ex.: Synchronized swimming
  - Ex.: termination, rendezvous (meeting by appointment)

- *However, concurrency leads to many problems*
  - Race conditions, Non-determinism
  - ...need to cooperate to avoid these problems!
  - Ex. Avoid simultaneous access to shared resources

- *Chapter 10. mostly mean following:*
  - Coordinate- bring into common action; harmonize
  - Cooperate- act jointly w/ others for common benefit
Synchronization Problems: A Story

- Stories
  - Car buying
  - Hello World, Producer-Consumer
  - Automatic Teller Machine story (Thread 8, HW4)
  - O’Henry, VCs visiting Johnson & Johnson

- A busy family’s car buying story
  - Had an old car needing replacement
  - Spouse1 visits a dealer and likes a car
    - Signs paper to buy the car w/ financing
    - and trade-in old car
  - Spouse2 visits another dealer and likes a car
  - Signs paper to buy the car w/ financing
    - and trade-in old car

- Q? How many car do they have now?
  - 2
  - 3 (trade-in not legal to either)
  - an old car + legal problems
Hello World with 2 Threads

- Hello World Story: Why coordinate?

```c
void print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello"; char *message2 = "World";
    pthread_create( &thread1, pthread_attr_default,
                    (void*)&print_message_function, (void*) message1);
    pthread_create(&thread2, pthread_attr_default,
                   (void*)&print_message_function, (void*) message2);
    exit(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
}
```
Coordination Needs

- **Coordination Needs**
  * Acess to shared resource (stdout)
    - printf across 2 threads
  * Thread rendezvous for process termination
    - exit(0) in main thread if other threads are done

- **Lack of Coordination**
  * => output is not deterministic!
  * a.k.a Race Conditions

- **Fix 1.**
  * add sleep(10) after each thread_create()
  * ? Does it eliminate race conditions?
  * ? Can it be used to remove all race conditions?

- **Fix 2.**
  * add 2 pthread_join() in main() to wait
  * for threads to finish
Hello World- coordinating w/ sleep()

```c
void print_message_function( void *ptr );
main()
{   pthread_t thread1, thread2;
    char *message1 = "Hello"; char *message2 = "World";
    pthread_create( &thread1, pthread_attr_default,
        (void *) &print_message_function, (void *) message1);
    sleep(10);
    pthread_create(&thread2, pthread_attr_default,
        (void *) &print_message_function, (void *) message2);
    sleep(10);
    exit(0);
}
void print_message_function( void *ptr )
{   char *message; message = (char *) ptr;
    printf("%s", message); pthread_exit(0);
}
```
Analyzing sleep() based fix

- **Problem 1: Relying on timing delay for synchronization**
  - Not safe
  - thread scheduling may not be predictable
  - a thread may be blocked for a while

- **Problem 2: Just like exit(),**
  - sleep() is a process level system call
  - i.e. All threads in the process sleep
  - Not useful for making main thread wait

- **Footnote: Thread level wait (not portable)**
  struct timespec delay;
  delay.tv_sec = 2;
  delay.tv_nsec = 0;
  pthread_delay_np( &delay );
Hello World Example - pthread_join()

```c
void print_message_function( void *ptr );
main()
{   pthread_t thread1, thread2;
    char *message1 = "Hello"; char *message2 = "World";
    pthread_create( &thread1, pthread_attr_default,
                     (void *) &print_message_function, (void *) message1);
    pthread_join(&thread1, NULL);
    pthread_create(&thread2, pthread_attr_default,
                     (void *) &print_message_function, (void *) message2);
    pthread_join(&thread2, NULL);
    exit(0);
}
void print_message_function( void *ptr )
{   char *message; message = (char *) ptr;
    printf("%s", message); pthread_exit(0);
}
```
Analyzing pthread_join() based solution

- **Advantages**
  * remove race b/w exit(0) and printf()
  * remove race b/w printf()s from 2 threads

- **Disadvantages**
  * Sequential
  * Little concurrency across threads
  * Not useful for many situations

- **Example: Producer-Consumer Problem**
  * Fig. 10.1 (pp. 366)
  * Both producer and consumer work concurrently
  * Shared resource = buffer
  * Producer - adds items to buffer
    - if there is an empty slot
  * Consumer - removes items from buffer
    - if there is a full slot
Threads Synchronization

- Threads share process-level resources
  - Memory, e.g. global / static variables
    - global data-structures, e.g. queues
  - I/O channels (e.g. stdout) and associated buffers
  - File descriptor tables, process signal mask, ...

- Coordination is needed to avoid problems

- Common Coordination needs
  - A. Mutual exclusion
  - B. Critical Sections (1 at a time)
  - C. Fixed number of servers (N at a time)
  - D. Wait for a general condition (or event)
Mutual Exclusion, Critical Section

- **Mutual Exclusion:**
  * At most one process/thread uses the resource at a time
  * Single server, e.g. use of 1 printer

- **Critical Sections: a segment of code**
  * that must be executed in a mutually exclusive manner.
  * Ex. Queue abstract data type
    - Implementation state in flux during steps of insert()
    - Operation insert() is a critical section!

- **Critical Section mechanism properties**
  * Mutual Exclusion
  * Progress: If no one is in the critical section, then
    - A process/thread wishing to enter can get in.
  * Bounded Waiting: No one is postpone indefinitely
  * Avoid busy waiting if possible
**Wait for a service, Conditional Wait**

- *Wait for a service*
  * fixed number of servers (N)
  * Each server attends to 1 client at a time
  * System can serve N clients at a time
  * e.g. wait till a fixed size buffer is not empty,

- *Conditional Wait:*
  * Waiting till an event happens!
  * e.g. wait till queue is not empty,
  * or wait till (producer is done) and (queue is empty)
Thread coordination in POSIX

- **POSIX tools for thread coordination**
  * mutex (M)
  * semaphore (S)
  * condition variable + mutex (CV + M)

- **Simple comparison**
  * complexity: M < S < (CV + M)

- **Matching techniques to problems**
  * Mutual exclusion - any tool
  * Critical Section - any tool
  * Wait on simple condition
    - semaphores or condition variables
    - mutex will lead to busy wait!
  * Wait on complex condition
  * Condition variables with mutex
    - mutex or semaphores will lead to busy wait!
POSIX Mutex

- * Mutex:
  * Chapter 10.1 (pp. 367-372)
  * Synopsis (pp. 367)

- * Mutex ADT
  * One Attribute: state-of-lock
  * Attribute type = Binary
  - Domain = (occupied, unoccupied)
  * Atomic Operations: lock(), unlock()

- * Implementation
  * Hardware support- atomic test-and-set instruction
POSIX Mutex - Usage

• Purpose of Mutex locks
  * Mutual Exclusion
  * Some aspects of critical section problem
  * Not for long waits due to busy wait problem.

• Typical Usage
  * Initialized to "unoccupied"
    - via macro PTHREAD_MUTEX_INITIALIZER
    - or system call pthread_mutex_init()
  * Each thread follows common protocol:
    - pthread_mutex_lock(&mutex_name) to acquire shared resource
    - pthread_mutex_unlock(&mutex_name) to release shared resource
  * Example 10.3 (pp. 368)
POSIX Mutex & Hello World Story

- Recall two problems

* Shared resource (stdout) - use mutex

* Termination - main wait for others

/* include proper header files */

```c
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;
void print_message_function(void *ptr)
{
    char *message; message = (char *) ptr;
    pthread_mutex_lock(&mx);
    printf("%s ", message);
    pthread_mutex_unlock(&mx);
}

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello"; char *message2 = "World";
    pthread_create(&thread1, pthread_attr_default,
                   (void *) &print_message_function, (void *) message1);
    pthread_create(&thread2, pthread_attr_default,
                   (void *) &print_message_function, (void *) message2);
    pthread_join(&thread1, NULL); pthread_join(&thread2, NULL);
    exit(0);
}
```
POSIX Mutex - Semantics

- Analogy: lock with a single key

- lock - the door and keep the key!
  * Blocking call, i.e. wait if key not there
    if (mutex-state == "occupied")
    then wait-for-mutex-to-be-unoccupied
    else mutex-state = "occupied";

- unlock - the door and return the key!
  if ((mutex-state == "occupied") and (it-was-locked-by-you))
  then mutex-state = "unoccupied";

- pthread_mutex_trylock()
  * Alternative to pthread_mutex_lock()
  * trylock() is non-blocking
  * returns error (EBUSY) if mutex is "occupied"
  * thread may something else instead of blocking
POSIX Mutex - Other operations

- **Initialization Methods**
  - (A) Example 10.2 (pp. 367)
    - macro PTHREAD_MUTEX_INITIALIZER
      - safer, guaranteed to execute at most once!
      - for "static" mutex, not for dynamic ones

- **Another Initialization Method**
  - (B) copy system call (Example 10.1, pp. 367)
    - for dynamically allocated mutex!
    - use before creating threads using the mutex!
      - pthread_mutex_init(&mutex_name, NULL)

- **pthread_mutex_destroy()**
  - Destructor, inverse of pthread_mutex_init()
  - assumes mutex-state = unoccupied
  - and if no thread will lock it anymore
POSIX Mutex - Exercise

- *Q? Justify the following advice on using mutex.*
  1. Do not unlock a mutex unless you locked it
  2. Do not unlock a mutex twice in sequence
  3. Do not lock a mutex twice in sequence i.e. EDEADLK
  4. Unlock all mutexes before sleep()/sched_yield()
  5. Hide lock/unlock calls within operation on an
     abstract data type!

- *Consider "Hello World" solution w/ mutex*
  1. Analyze the consequences of following changes:
     1. program is run on a multi-processor hardware
     2. mutex "mx" is local variable in print_message_function()
     3. mutex "mx" is local to main()
     4. lock() and unlock() statements swapped in code

- *Recitation: More detailed exercise (lock.c)*
POSIX Mutex - Risks

- **Risks**
  
  * Protocol is voluntary, no enforcement!
  
  * A uncooperative thread may violate the protocol
    - putting everyone else in jeopardy

- **Suggestion: Combine with Abstract data types (ADTs)**
  - operation on ADT should use mutex properly
  - threads access ADTs via operations
    * Case Study: Producer-Consumer problem!
      - Example: Program 10.1 (pp. 368-9)

- **We will revisit case study next week!**
  
  * Compare mutex, semaphores, condition variables
Chapter 8.3.1 - 8.3.2

* Synopsis (pp. 305-6)

* Semaphores

* One Attribute: state-of-semaphore (a.k.a. count)

* Attribute type = positive integer

* Atomic Operations: sem_wait(), sem_post()

* \texttt{sem\_wait()}

\begin{verbatim}
if (count == 0) wait-till-count-is-positive;
count--; 
\end{verbatim}

* \texttt{sem\_post()}

\begin{verbatim}
count++; 
\end{verbatim}

* Implementation of \texttt{sem\_wait()} and \texttt{sem\_post} requires

* Software- mutex locks

* or Hardware test-and-set instruction
**POSIX Semaphore**

- *Purpose of Semaphore*
  
  * Wait for simple condition w/o busy waiting
  - e.g. (count = 0), (count > 0), etc.
  - e.g. queue full, buffer empty, etc.

  * Also for critical section, mutual exclusion

  * Not for waits on complex condition (busy wait problem)

- *Typical Usage*

  * Initialized to the max. number of resources

  * Each thread:
    - `sem_wait(S1)` to acquire shared resource
    - `sem_post(S2)` to release shared resource
    - wait & post may be on different semaphore
Example: Program 10.3 (pp. 373-4)

* 1. How many semaphores are used?
* 2. What are the initial values of each?
* 3. How many threads are there?
* 4. Does each thread follow the protocol?
* 5. What is the shared resource?
* 6. What are the conditions monitored?
* 7. What are the race conditions?
  - Which conditions are handled by semaphores?
POSIX semaphores - Example

- *Analysis*
  * 1. Two (items, slots)
  * 2. items = 0, slots = BUFSIZE
  * 3. Two (prodtid/producer, constid/consumer)
  * 4. Yes - wait ... post
  * 5. buffer with BUFSIZE slots
    - buffer[], bufin, bufout
  * 6. changes to buffer[], bufin, bufout
    - producer overwriting item if buffer is full
    - consumer reads illegal item if buffer is empty
  * 7. full_buffer halts producer
    - empty_buffer halts consumer
POSIX semaphores - Risks

- **Risks**
  - Protocol is voluntary, no enforcement!
  - A uncooperative thread may violate the protocol
    - putting everyone else in jeopardy

- **Suggestion: Combine with Abstract data types (ADTs)**
  - operation on ADT should use semaphore properly
  - threads access ADTs via operations
    - Case Study: Producer-Consumer problem!

- **Other Protocols are possible!**
  - See example semaphore.c in recitation!
POSIX Semaphores - Other Operations

- **Initialization/copy operation (Synopsis (pp. 305))**
  ```c
  int sem_init(sem_t *sem, int pshared, unsigned int value)
  
  * Argument 1: pshared = 0 for threads in a process
    - pshared != 0 for a process group
  * Argument 2: (value >= 0)
    - initializes the "count" of resources
  * Dynamic memory allocation and initialization
  * Usage Ex.: Program 8.2 (pp. 307)
  ```

- **Recycling operation: int sem_destroy(sem_t *sem)**
  * destroy a previously initialized semaphore
  * ensure no one is waiting on it

- **Non-blocking wait: int sem_trywait(sem_t *sem)**
  * Alternative to blocking sem_wait()
  * Return -1 and (errno = EAGAIN) instead of blocking

- **Getting value of semaphore**
  ```c
  int sem_getvalue(sem_t *sem, int *sval)
  
  * No guarantee on the time when sval is read!
  ```
POSIX Semaphores - Exercises

- **Example. Program 8.2 (pp. 307)**
  
  * Q? How many semaphores are used?
  * Q? What are the initial values of each?
  * Q? What is the shared resource, race condition?
  * Q? How many threads can "fputc()" at the same time?

- **Mutex is a special case of semaphore.**
  
  * Q? What initial value for semaphore will
  * sem_wait() behave like lock()
  * and sem_post() behave like unlock()

- **Rewrite Hello World using semaphores**

  ```c
  pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;
  void print_message_function( void *ptr )
  {
    char *message; message = (char *) ptr;
    pthread_mutex_lock(&mx);
    printf("%s ", message);
    pthread_mutex_unlock(&mx);
  }
  ```