Housekeeping (Lecture 4 - 9/9/2013)

- Warmup #1 due at 11:45pm this Friday, 9/13/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
  - get started soon
    - if you are stuck, make sure you come to see me during office hour next Monday

- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - due to our *fairness* policy
  - it's a good idea to run your code against the grading guidelines

- You need to keep up with the lecture materials
  - anything you don't understand fully, come to see me soon
    - or post a message to the class Google Group

Housekeeping (Lecture 4 - 9/9/2013)

- Please do not set your class Google Group e-mail delivery preference to "No email"

- When you have a question, please do one of the follow and ***not both***
  - post to the class Google Group
    - if your classmates have responded, I may or may not respond
      - if I don’t respond, it does not necessarily mean that your classmates got the right answer
      - it may mean that they are pointing you in the right direction
      - you need to learn how to figure out what the right answer is
  - send me an e-mail directly
    - it may take 24 hours for me to reply
    - for kernel projects, my response may simply be that you should post your question to the class Google Group because I should not be giving you a straight answer
      - not different from a HW in a regular class, if you ask for an answer, you usually won’t get a straight answer

Creating a POSIX Thread

- man pthread_create

```
SYNOPSIS
   #include <pthread.h>

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

   Compile and link with -pthread.
```

Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i=0; i<nr_of_server_threads; i++)
    pthread_create(&thread,    // thread ID
                   0,          // default attributes
                   server,     // start routine
                   argument); // argument
}

void *server(void *arg) {
  // perform service
  return(0);
}
```

- `pthread_create()` returns 0 if successful
- POSIX 1003.1c standard

Creating a POSIX Thread

- These are the same:
  - keep thread handle in the stack

```
pthread_t thread;
pthread_create(&thread, ...);
```

  - keep thread handle in the heap

```
pthread_t *thread_ptr =
    (pthread_t*)malloc(sizeof(pthread_t));
pthread_create(thread_ptr, ...);
```

    - need to make sure that eventually you will call the following to not leak memory

```
free(thread_ptr);
```

Creating a Win32 Thread

```
start_servers( ) {
  HANDLE thread;
  DWORD id;
  int i;
  for (i=0; i<nr_of_server_threads; i++)
    thread = CreateThread(
        0,        // security attributes
        0,        // default # of stack pages allocated
        server,   // start routine
        arg,      // argument
        0,        // default attributes
        0,        // creation flags
        &id);     // thread ID
}

DWORD WINAPI server(void *arg) {
  // perform service
  return(0);
}
```

- We won’t talk about Win32 much

Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  pthread_create(&in_thread,
                 0,
                 incoming,
                 r_in, l_out);
  pthread_create(&out_thread,
                 0,
                 outgoing,
                 l_in, r_out);

}
```

Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  pthread_create(&in_thread,
                 0,
                 incoming,
                 r_in, l_out); // Cannot do this ...
  pthread_create(&out_thread,
                 0,
                 outgoing,
                 l_in, r_out); // Cannot do this ...
  /* How do we wait till they are done? */
}
```

Multiple Arguments

```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
  ...
  /* How do we wait till they are done? */
}
```

Multiple Arguments

- Need to be careful how to pass multiple arguments to `pthread_create()`
  - there is no way to pass multiple arguments in either POSIX or Win32
  - passing address of a *local* variable (like the previous example) only works if we are certain if this storage doesn't go out of scope until the thread is done with it
  - passing address of a *static* or a *global* variable only works if we are certain that only one thread at a time is using the storage
  - passing address of a *dynamically* allocated storage only works if we can free the storage when, and only when, the thread is finished with it
    - this would not be a problem if the language supports garbage collection

- Ask yourself, "How can you be sure?"
  - if the answer is, "I hope it works", then you need a different solution

When Is The Child Thread Done?

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};

  pthread_create(&in_thread, 0, incoming, &in);
  pthread_create(&out_thread, 0, outgoing, &out);

  pthread_join(in_thread, 0);
  pthread_join(out_thread, 0);
}
```

Thread Termination

- Thread return values
  - which threads receive these values
  - how do they do it?
    - clearly, receiving thread must wait until the producer thread produced it, i.e., producer thread has terminated
    - so we must have a way for one thread to wait for another thread to terminate
  - must have a way to say which thread you are waiting for
    - need a unique identifier
    - tricky if it can be reused
- To wait for another thread to terminate

```
int pthread_join(thread_t thread,
                 (void **)ret_value);
```

Thread Termination

- How does a thread terminate?
  1) return from its "start routine"
     - return a value of type `(void*)`
  2) call `pthread_exit(ret_value)`
     - `ret_value` is of type `(void*)`

```
parent() {
  pthread_t thread;
  void *result;
  pthread_create(&thread,
      0, child, 0);
  pthread_join(thread,
      (void**)&result);
  switch ((int)result) {
  case 1: ...
  case 2: ...
  }
  ...
}
```

```
void *child(void *arg) {
  ...
  if (terminate_now) {
    pthread_exit((void*)1);
  }
  return((void*)2);
}
```

Thread Termination

- Difference between `pthread_exit()` and `exit()`
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what will this code do?

```
int main(int argc, char *argv[]) {
  // create all the threads
  return(0);
}
```

    - when `main()` returns, `exit()` will be called
      - as a result, none of the created child thread may get a chance to run

Thread Termination

- Difference between `pthread_exit()` and `exit()`
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what about this code?

```
int main(int argc, char *argv[]) {
  // create all the threads
  pthread_exit(0); // exit the main thread
  return(0);
}
```

    - here, `pthread_exit()` will terminate the main thread, so `exit()` is never called
      - as it turns out, this special case is taken care of in the pthread library implemetation
- You should use `pthread_join()` unless you are absolutely sure

Thread Termination

- **Any thread can join with any other thread**
  - there's no parent/child relationships among threads
    - unlike process termination and `wait()`
- **What happens if a thread terminates and no other thread wants to join with this thread?**
  - it also goes into a *zombie* state
    - all the thread related information is freed up, except for the thread ID and return code
- **What if two threads want to join with the same thread?**
  - after the first thread joins, the thread ID and return code are freed up and the thread ID may get reused
  - so don't do this!

Detached Threads

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i=0; i<nr_of_server_threads; i++) {
    pthread_create(&thread, 0, server, 0);
    pthread_detach(thread);
  }
  ...
}

server( ) {
  ...
}
```

Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);
```

Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);

int func = 4; // func definition 1
```

Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);

int func = 4; // func definition 1

void func(int i) { // func definition 2
  ...
}
```

Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);

int func = 4; // func definition 1

void func(int i) { // func definition 2
  ...
}

void *func(void *arg) { // func definition 3
  int i = (int)arg;
  ...
  return(0);
}
```

Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
/* establish some attributes */
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
...
```

- thread attribute only needs to be valid when a thread is created
  - therefore, it can be destroyed as soon as the thread is created

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
```

- the above code set the stack size to 20MB
- the default stack size is very large
  - if you need to create a lot of threads, you need to control the stack size
  - default stack size is probably around 1MB in Solaris and 8MB in some Linux implementations

Example

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>

#define M  3
#define N  4
#define P  5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *arg) {
  int row = (int)arg, col;
  int i, t;

  for (col=0; col < P; col++) {
    t = 0;
    for (i=0; i<N; i++)
      t += A[row][i] * B[i][col];
    C[row][col] = t;
  }
  return(0);
}
```

```
main( ) {
  int i;
  pthread_t thr[M];
  int error;

  /* initialize the matrices ... */
  ...
  // create the worker threads
  for (i=0; i<M; i++) {
    if (error = pthread_create(
        &thr[i],
        0,
        matmult,
        (void *)i)) {
      fprintf(stderr,
          "pthread_create: %s",
          strerror(error));
      exit(1);
    }
  }
  // wait for workers to finish
  for (i=0; i<M; i++)
    pthread_join(thr[i], 0)
  /* print the results ... */
}
```

Compiling It

```
% gcc -o mat mat.c -lpthread
```

2.2.3 Synchronization

Mutual Exclusion

Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
```

Thread 2:

```
x = x+1;
```

- looks like it doesn't matter how you execute, x will be incremented by 2 in the end
  - choices are
    - thread 1 executes x = x+1 then thread 2 executes x = x+1
    - thread 2 executes x = x+1 then thread 1 executes x = x+1
  - are there other choices?

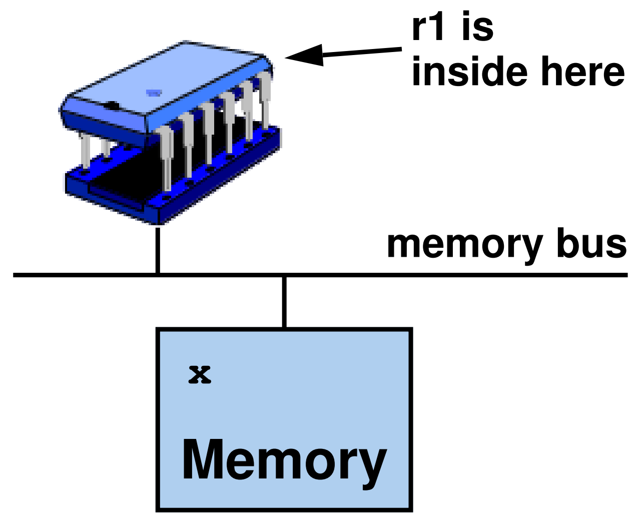
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
  ld    r1,x
  add   r1,1
  st    r1,x
  */
```

Thread 2:

```
x = x+1;
 /*
  ld    r1,x
  add   r1,1
  st    r1,x
  */
```



- Unfortunately, machines do not execute high-level language statements
  - they execute machine instructions
  - now if thread 1 executes the first (or two) machine instructions
  - context switch!
    - how can this happen?
  - then thread 2 executes all 3 machine instructions
  - then later thread 1 executes the remaining machine instructions
  - x would have only increated by 1

Threads and Synchronization

```
// shared by both threads
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
...
pthread_mutex_lock(&m);

x = x+1;

pthread_mutex_unlock(&m);
```

critical section

Locking a mutex is like getting the key to a safe-deposit box

- code between `pthread_mutex_lock()` and `pthread_mutex_unlock()` for a particular *mutex* is called a *critical section* with respect to that mutex
  - for a mutex, the OS guarantees that only *one* critical section can be executing at any point in time
  - *all* the critical sections *with respect to a particular mutex* are *"mutually exclusive"*
  - how it's really done in the kernel will be covered in Ch 5

Set Up

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- If a mutex cannot be initialized statically, do:

```
int pthread_mutex_init(
        pthread_mutex_t *mutexp,
        pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(
        pthread_mutex_t *mutexp)
```

- Usually, mutex attributes are not used

Taking Multiple Locks

- Mutex is not a cure-all
  - when you have more than one locks, you may get into trouble

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

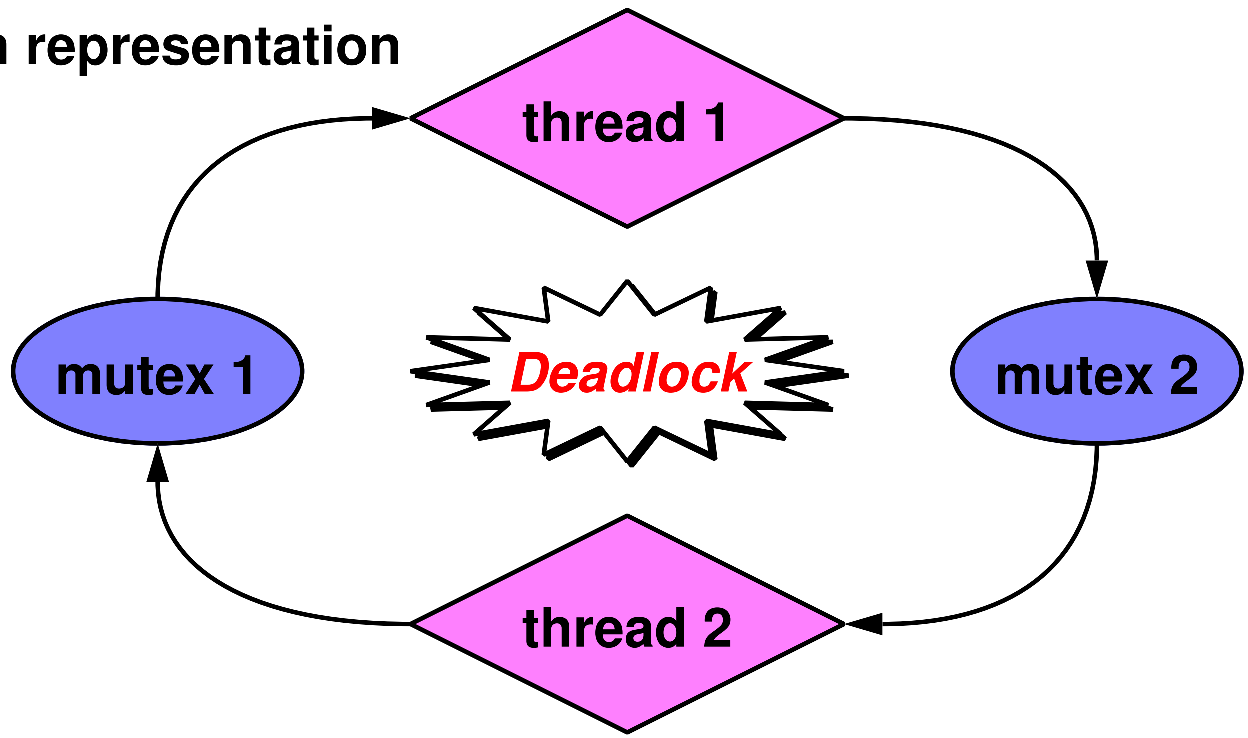
```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Taking Multiple Locks

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

- Graph representation



Necessary Conditions For Deadlocks

- **All 4 conditions below must be met in order for a deadlock to be possible (no guarantee that a deadlock may occur)**

  1) **Bounded resources**
     - only a finite number of threads can have concurrent access to a resource

  2) **Wait for resources**
     - threads wait for resources to be freed up, without releasing resources that they hold

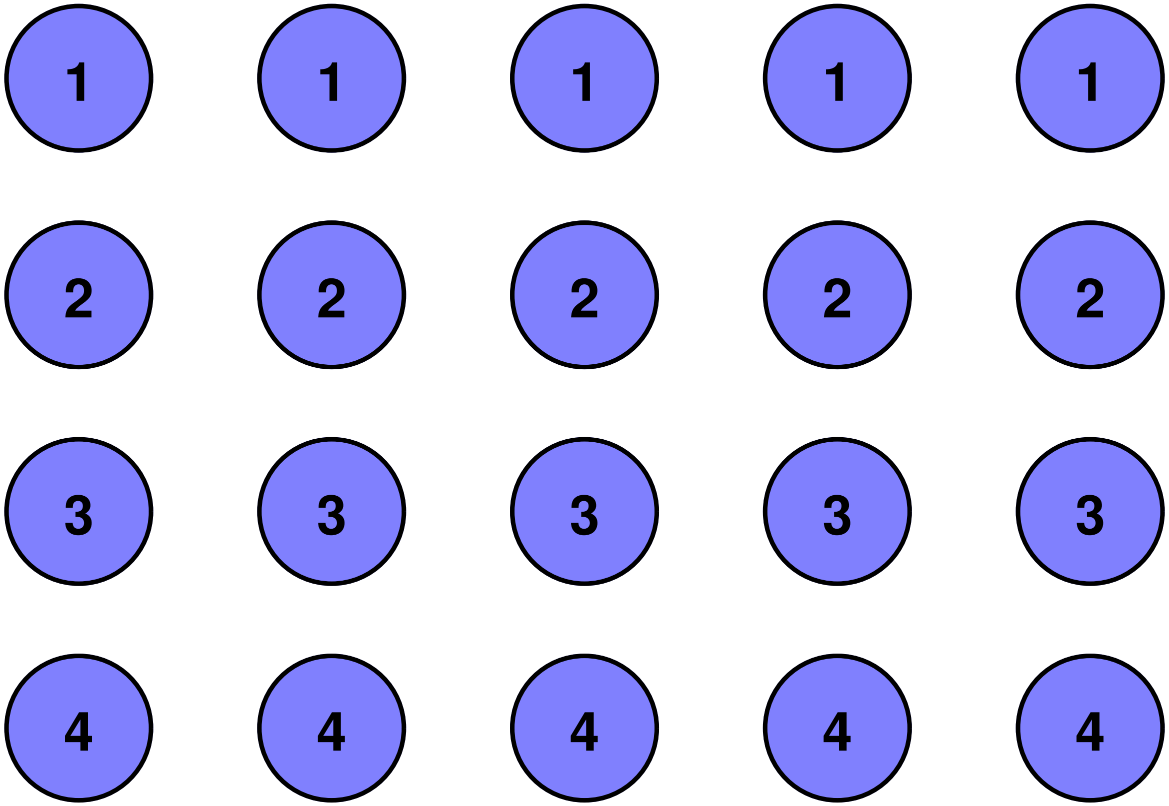
  3) **No preemption**
     - resources cannot be *revoked* from a thread

  4) **Circular wait**
     - there exists a set of waiting threads, such that each thread is waiting for a resource held by another

Dealing with Deadlock

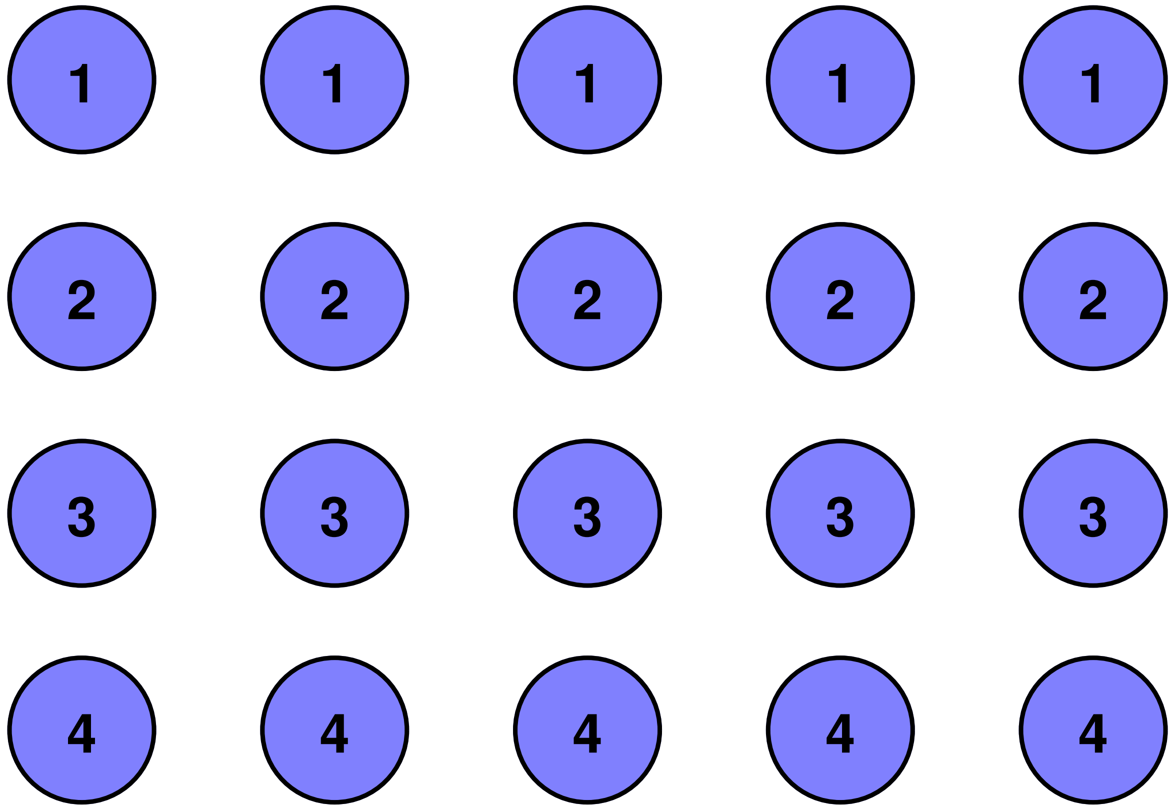
- Deadlock is a programming bug
  - one of the oldest bug
  - it's a tricky one because it only deadlocks *sometimes*
- Hard
  - is the system deadlocked?
  - will this move lead to deadlock?
  - this is *detection*
    - if you can detect deadlocks, what do you do after you have detected them?
- Easy
  - restrict use of mutexes so that deadlock cannot happen
  - this is *prevention*

Deadlock Prevention: Lock Hierarchies



- organize mutexes into levels
- must not try locking a mutex at level *i* if already holding a mutex at level *j* if *i* < *j*, otherwise it's okay
  - e.g., if hold mutexes at levels 2 & 3, can only wait for a mutex at level 3 or 4

Deadlock Prevention: Lock Hierarchies



- What if you cannot organize your mutexes in such strict order for deadlock detection?

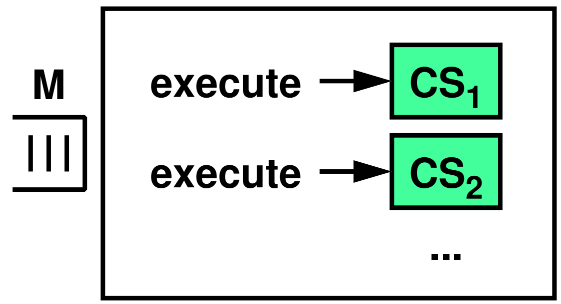
Deadlock Prevention: Conditional Locking

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}

proc2( ) {
  while (1) {
    pthread_mutex_lock(&m2);

    if (!pthread_mutex_trylock(&m1))
      break;
    pthread_mutex_unlock(&m2);
  }
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Queueing For A Mutex

```
// shared by both threads
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
...
pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

critical section



- Can think of `pthread_mutex_lock(&m)` as getting into a *queue* and *wait* there *indefinitely* for mutex `m` to become available
  - multiple threads would join this queue
  - queue is served one at a time, like a supermarket checkout
  - when it's your thread's turn, `pthread_mutex_lock()` returns with the mutex locked, your thread can execute critical section code, and then release the mutex