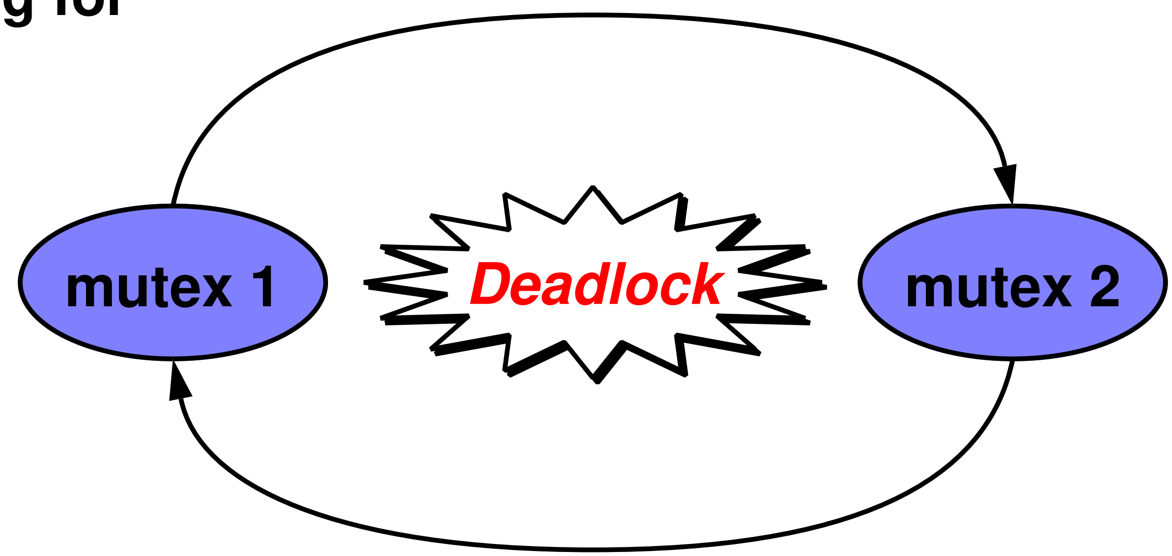
Taking Multiple Locks

```
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 ("wait-for" graph) for the entire process
  - draw an arrow from a mutex you are holding to another mutex you are waiting for



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**

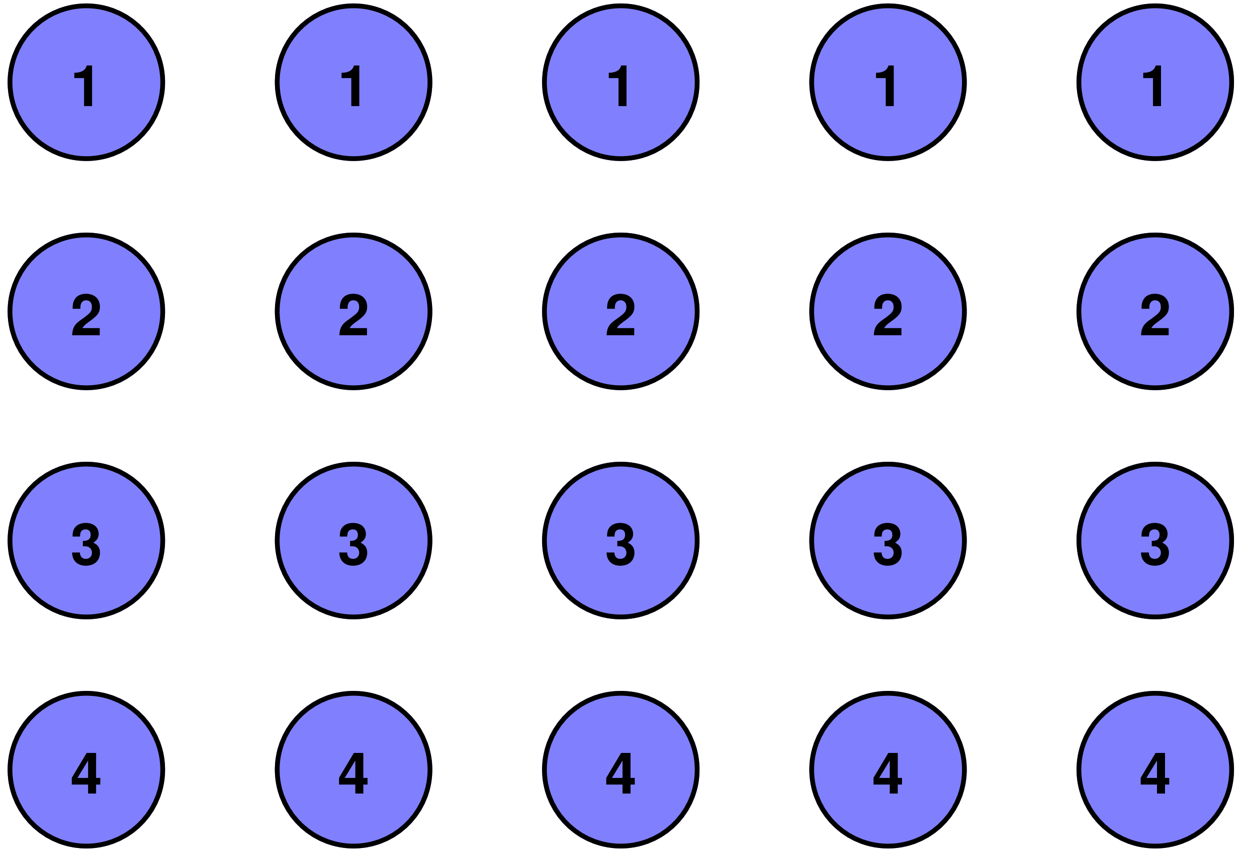
3) **No preemption**
   - **resources cannot be *revoked* from a thread**

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

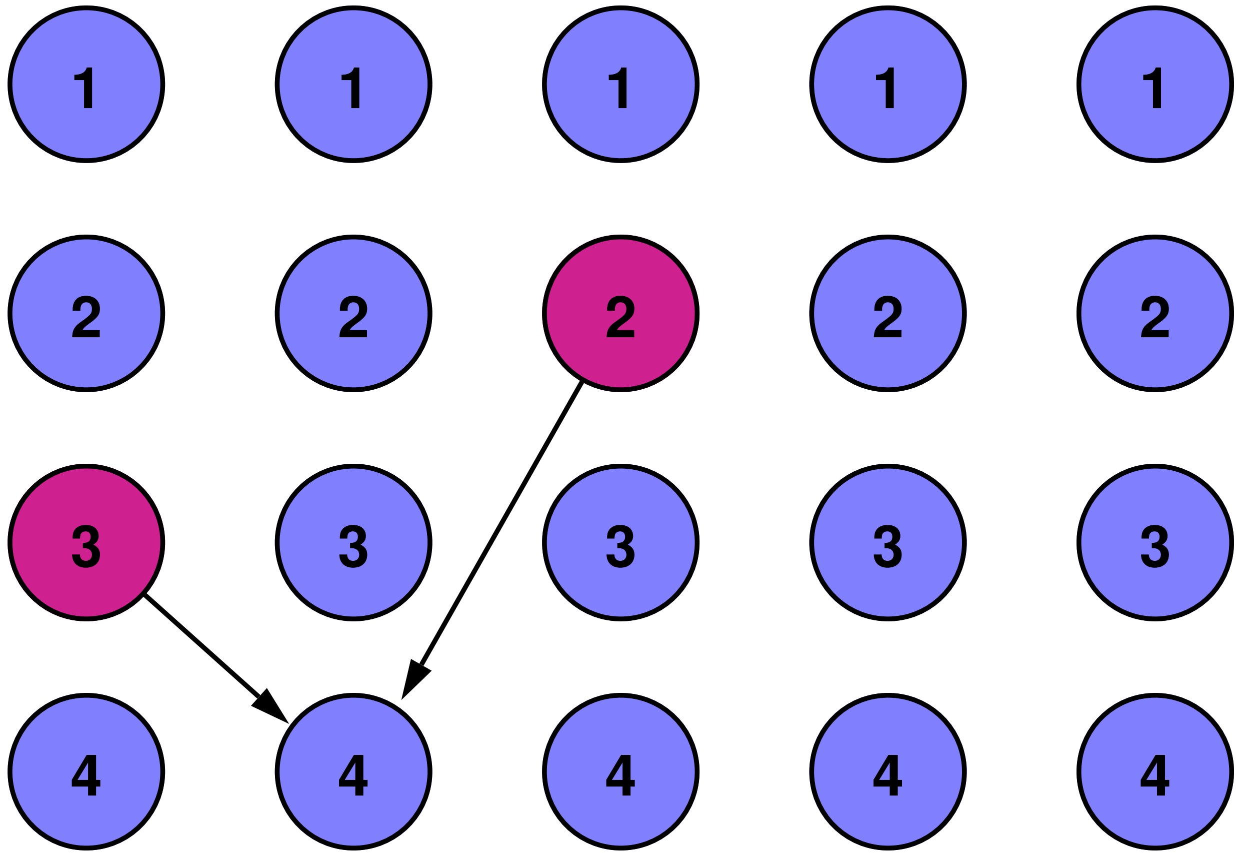
- Deadlock is a programming bug
  - one of the oldest bug
  - it's a tricky one because it only deadlocks *sometimes*
- Hard
  - is the system deadlocked?
  - will this move lead to deadlock?
  - this is *detection*
    - if you can detect deadlocks, what do you do after you have detected them?
- Easy
  - restrict use of mutexes so that deadlock cannot happen
  - this is *prevention*
- Deadlock is a complicated subject
  - some textbooks spend an entire chapter on deadlocks
  - we will only look at a couple of cases

Deadlock Prevention: Lock Hierarchies



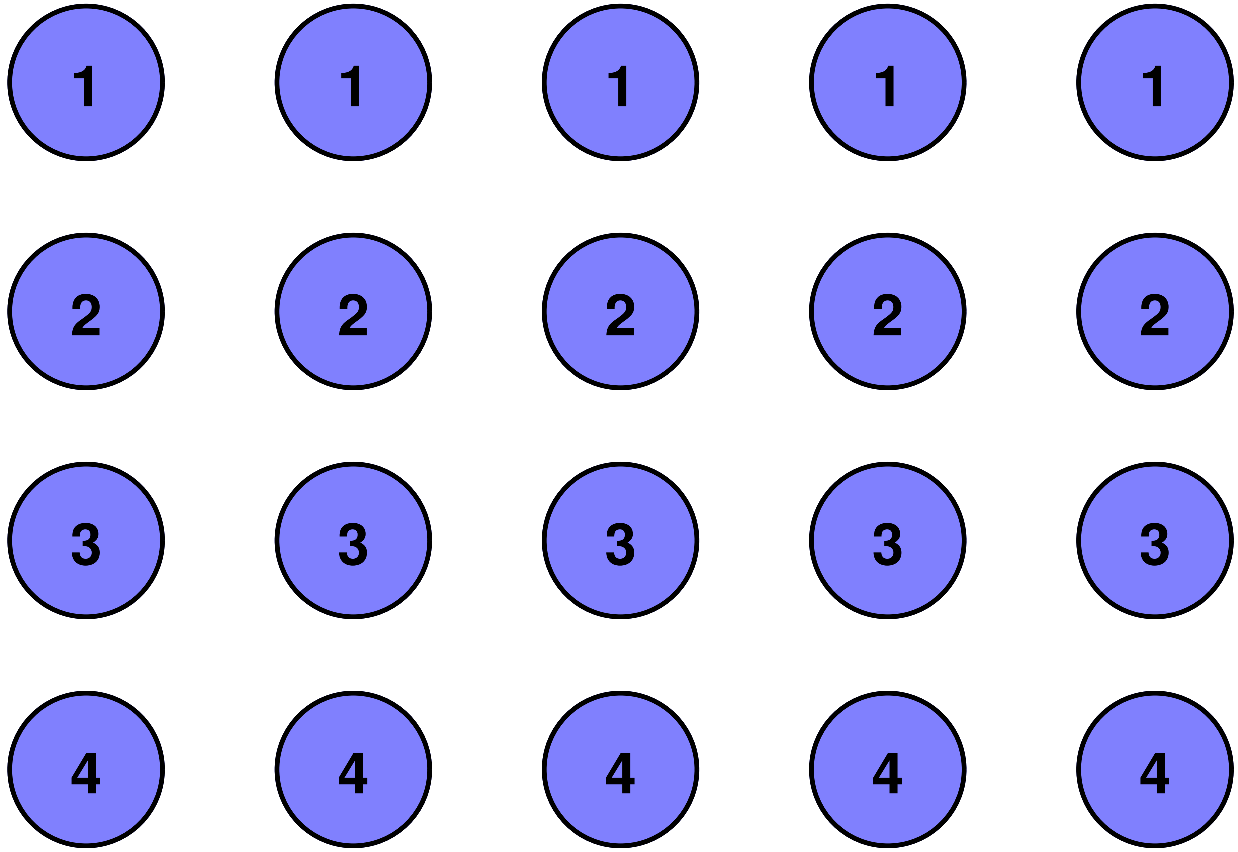
- ***If*** **you can organize mutexes into levels and satisfies:**
  - **must not try locking a mutex at level *i* if already holding a mutex at equal or higher level, otherwise it's okay**

Deadlock Prevention: Lock Hierarchies



- *If* you can organize mutexes into levels and satisfies:
  - must not try locking a mutex at level *i* if already holding a mutex at equal or higher level, otherwise it's okay
    - e.g., if holding mutexes at levels 2 and 3, can only wait for a mutex at levels 4 or higher

Deadlock Prevention: Lock Hierarchies



- What if you cannot organize your mutexes in such strict order for deadlock prevention?
  - can we avoid "necessary conditions" for deadlocks (1), (2), or (3)?

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);
}
```

Mutex Summary

- How do threads interact with each other?
  - they interact with each other by calling pthread functions
    - e.g., `pthread_mutex_lock()`, `pthread_mutex_unlock()`
  - they interact with each other using *shared variables*

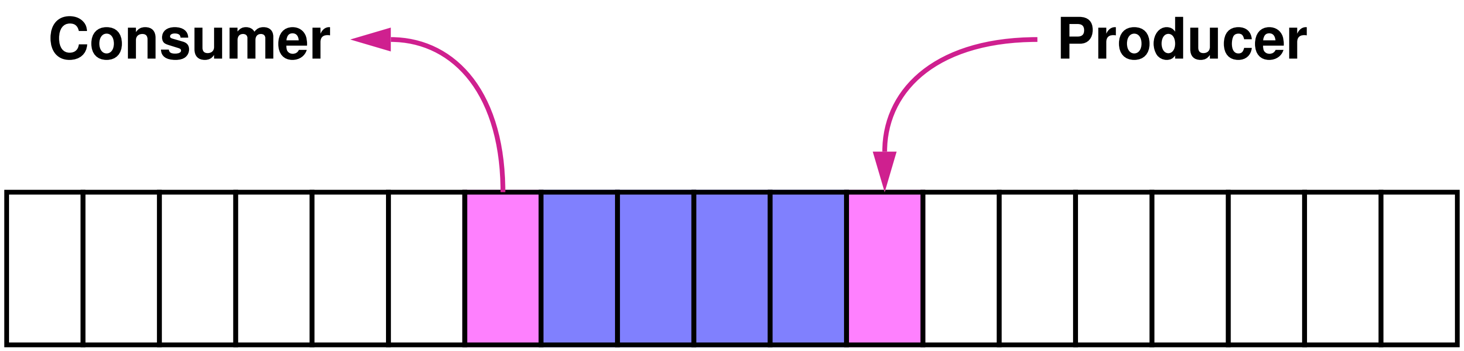
- If multiple threads want to *share read-only data* (i.e., no thread will ever modify the shared data)
  - you don't need to use mutex

- If multiple threads want to share data for *reading and writing* (or just for writing, which is unusual)
  - all these threads must share a mutex and only access the shared data using *critical section code with respect to this mutex*
  - in general, critical section code may be *nested* (as in the case of locking hierarchy)
    - also, a thread may be using different mutexes to interact with different threads

Beyond Mutexes

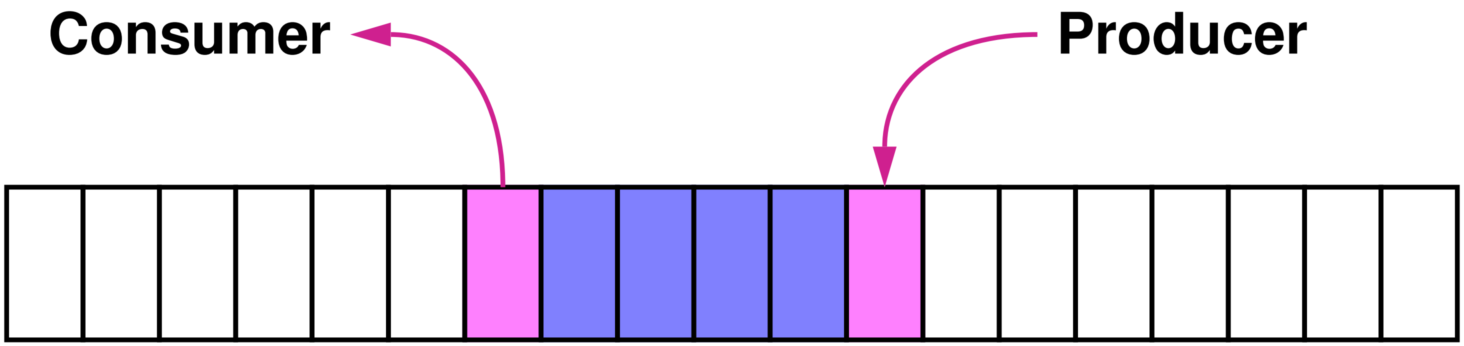
- **Mutex is necessary when shared data is being modified**
  - **although there are cases where using a mutex is an overkill (i.e., too restrictive and inefficient and would lock threads out when it's not necessary)**
    - **can always use one mutex to lock up all shared data**
      - **no parallelism (when *some parallelism can be* permitted)**
    - **we would like to have better concurrency *(i.e., "fine-grained parallelism")* when *complete mutual exclusion in not required***
  - **two major categories to illustrate this**
    1) **what if threads don't interfere one another most of the time and synchronization is only required occasionally?**
       - **e.g., *Producer-Consumer problem* (a.k.a., bounded-buffer problem)**
    2) **what if some threads just want to *look at (i.e., read)* a piece of data?**
       - **e.g., *Readers-Writers problem***
  - **will also look at *Barrier Synchronization***

Producer-Consumer Problem



- Conveyor belt (hardware)
  - perfect parallelism between producer and consumer

Producer-Consumer Problem



- Conveyor belt (hardware)
  - perfect parallelism between producer and consumer
- A circular buffer is used in software implementation
- Most of the time, no interference
  - if you use a *single mutex* to lock the entire array of buffers, it's an overkill *(i.e., too inefficient)*
- When does it require synchronization?
  - producer needs to be blocked when all slots are full
  - consumer needs to be blocked when all slots are empty

Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

command sequence

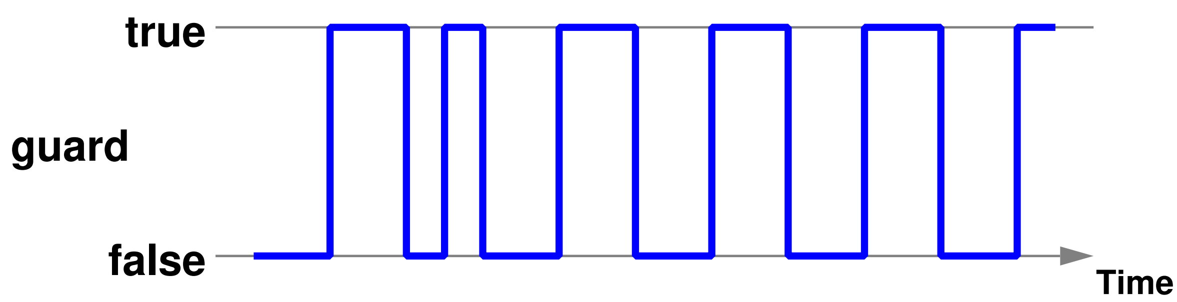
- this means that the command sequence is executed (*atomically*) *when* the *guard* is *evaluated* to be *true*
  - a *guard* is a *boolean expression* (evaluates to true or false)
  - *atomically* mean that it's executed "without interruption"
    - but with respect to what?
    - *evaluating the guard* and *executing the command sequence* altogether is an *atomic operation* *if* the *guard is true*
    - you cannot evaluate the guard if your thread is not running

- For exams, you need to know how to write simple pesudo-code in the language of *Guarded Commands*

Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

command sequence

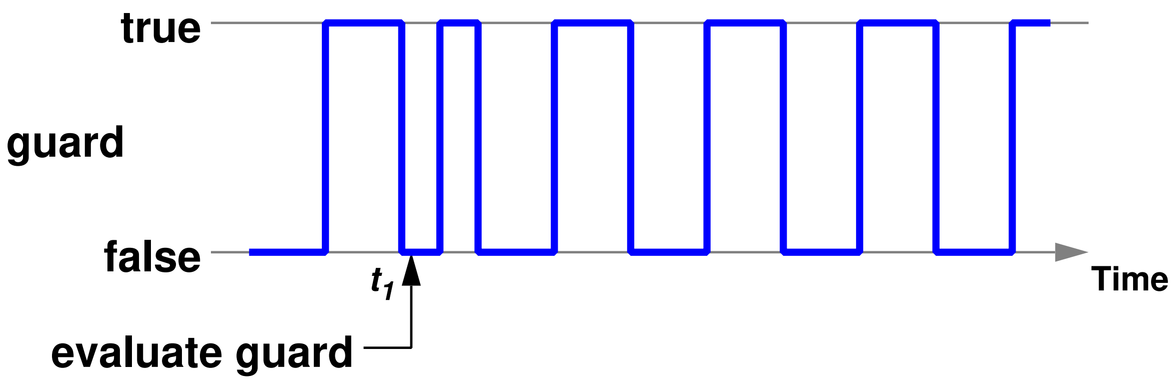


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
  /*
     once the guard is true,
     execute this code atomically
   */
  ...
]
```

command sequence

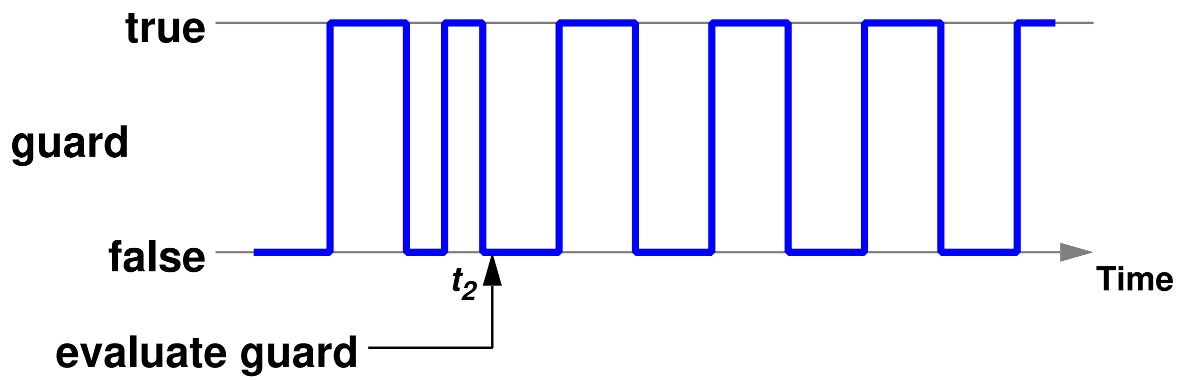


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

command sequence

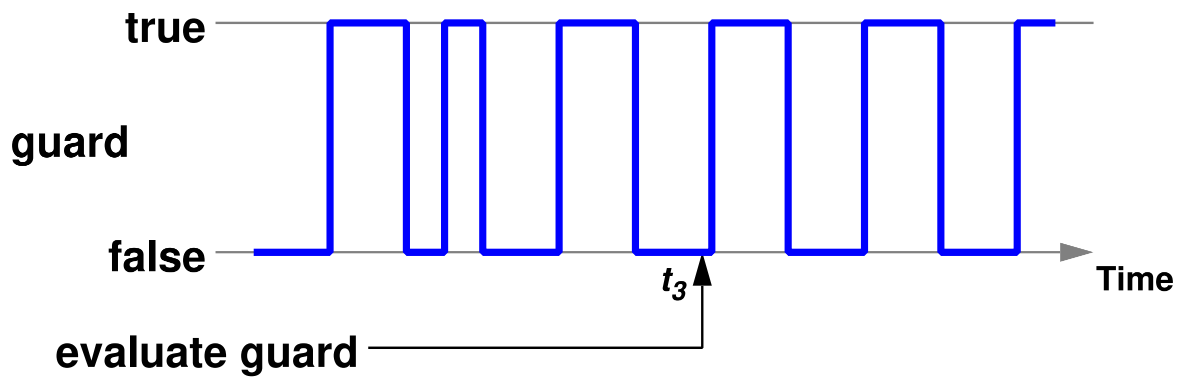


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

command sequence

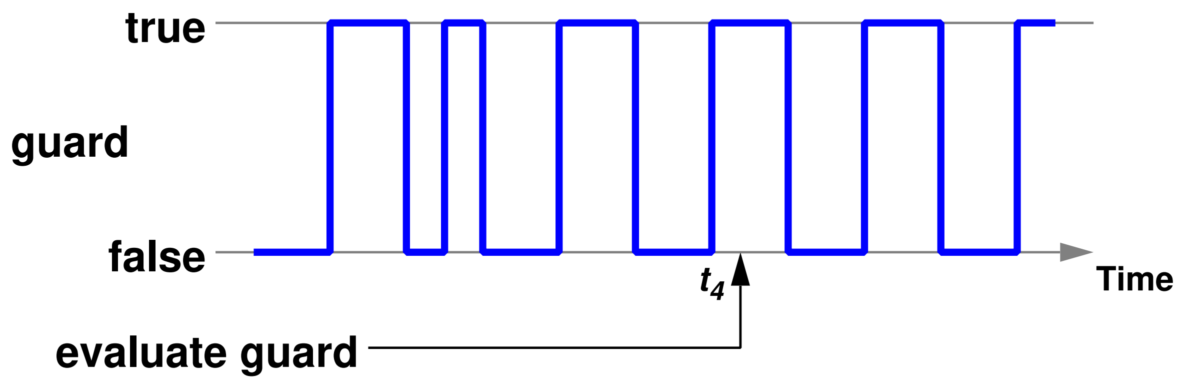


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
  ...
]
```

command sequence

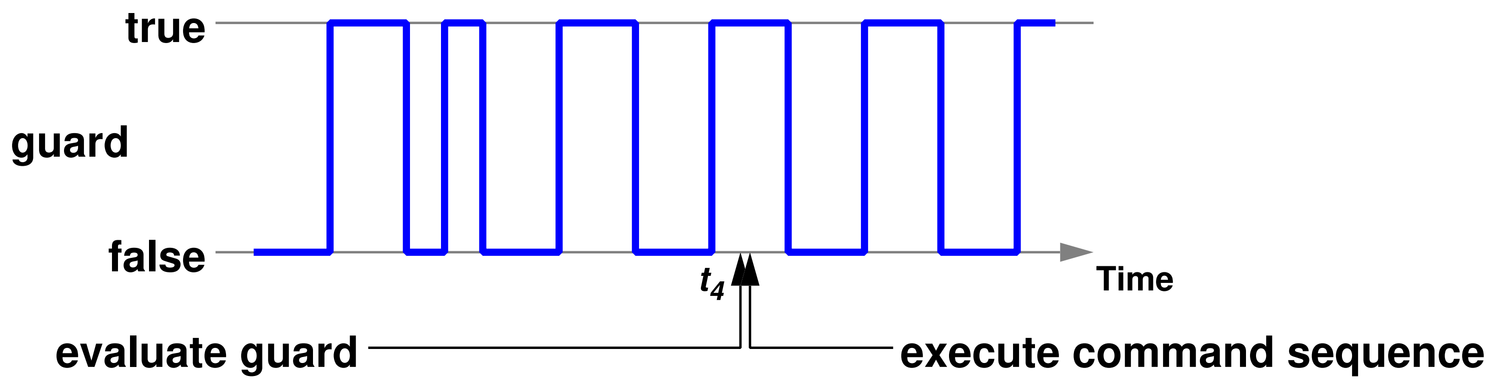


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

command sequence

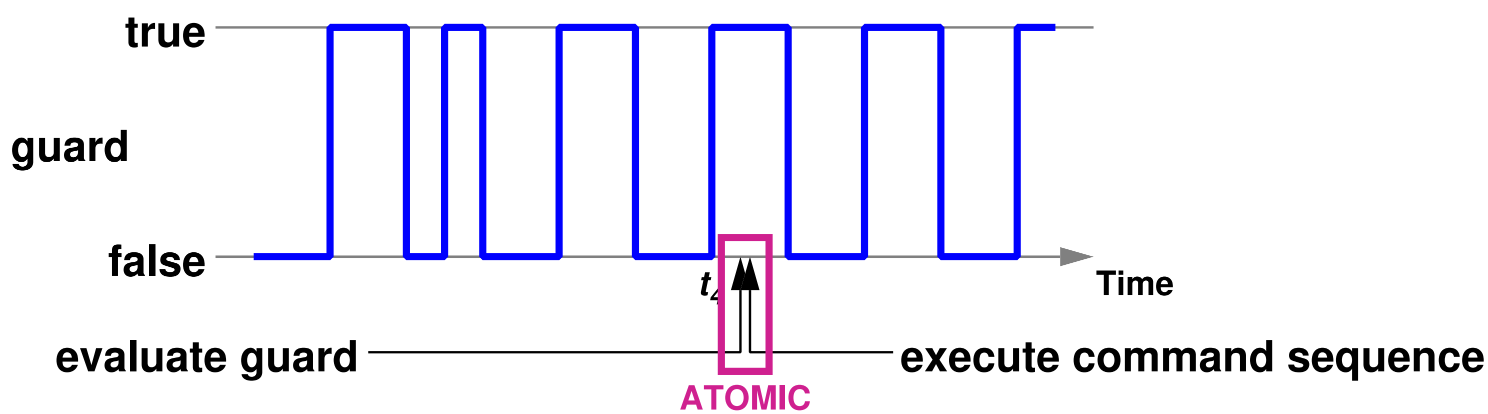


- you can only evaluate the guard if your thread is running

Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

command sequence

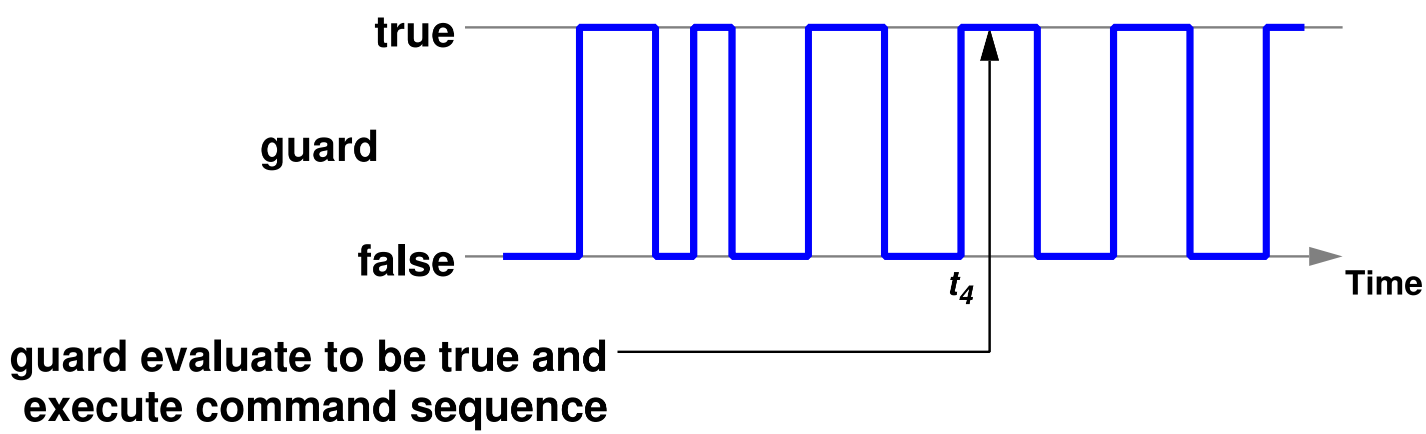


- please understand that *command sequence ≠ critical section*
  - evaluate the guard to be true *and* execute command sequence *together* is done inside *one critical section*

Guarded Commands

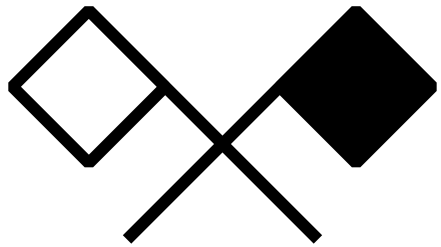
```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

command sequence



- ***atomic:*** as if it's executed in an instance of time (duration = 0)
  - this is okay because it's just pseudo-code

Semaphores



- A *semaphore*, `S`, is a *nonnegative integer* on which there are exactly two operations defined by two garded commands
  - `P(S)` operation (implemented as a guarded command):
    - ```
      when (S > 0) [
        S = S - 1;
      ]
      ```
  - `V(S)` operation (implemented as a guarded command):
    - `[S = S + 1;]`
  - there are no other means for manipulating the value of `S`
    - other than initializing it
      ```

Mutexes with Semaphores

```
semaphore S = 1;

void OneAtATime( ) {
  P(S);
  ...
  /* code executed mutually
     exclusively */
  ...
  V(S);
}
```

- `P(S)` operation:
  - `when (S > 0) [`
    `S = S - 1;`
    `]`
- `V(S)` operation:
  - `[S = S + 1;]`

- this is known as a *binary semaphore*

Implement A Mutex With A Binary Semaphore

- Instead of doing

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

  - do:

```
semaphore S = 1;
P(S);
x = x+1;
V(S);
```

- So, you can lock a data structure using a binary semaphore
  - this looks just like mutex, what have we really gained?
    - if you use it this way, nothing

Mutexes with Semaphores

```
semaphore S = N;

void NAtATime( ) {
  P(S);
  ...
  /* no more than N threads
     here at once */
  ...
  V(S);
}
```

- P(S) operation:
  - `when (S > 0) [`
    `S = S - 1;`
    `]`
- V(S) operation:
  - `[S = S + 1;]`

- this is known as a *counting semaphore*
- can be used to solve the producer-consumer problem

Main difference between a semaphore and a mutex

- if a thread locks a mutex, it's holding the lock
  - therefore, it must be that thread that unlocks that mutex
- one thread performs a `P` operation on a semaphore, *another thread* performs a `V` operation on the same semaphore
  - this is often why you would use a semaphore