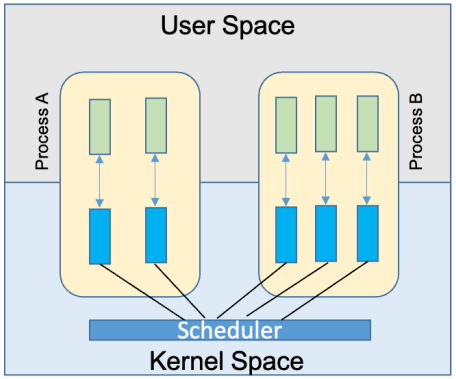
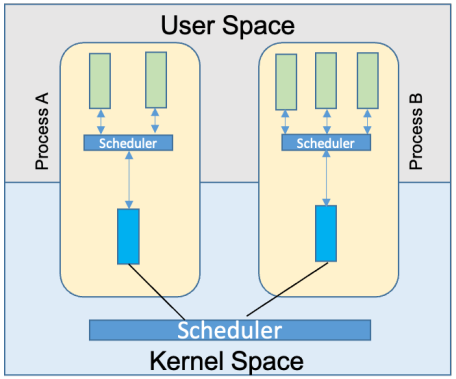
Lecture 22: Race Condition in Multithreading

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture





- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Today's Class

- Mutual exclusion
- Condition variables
- Producer-consumer problem

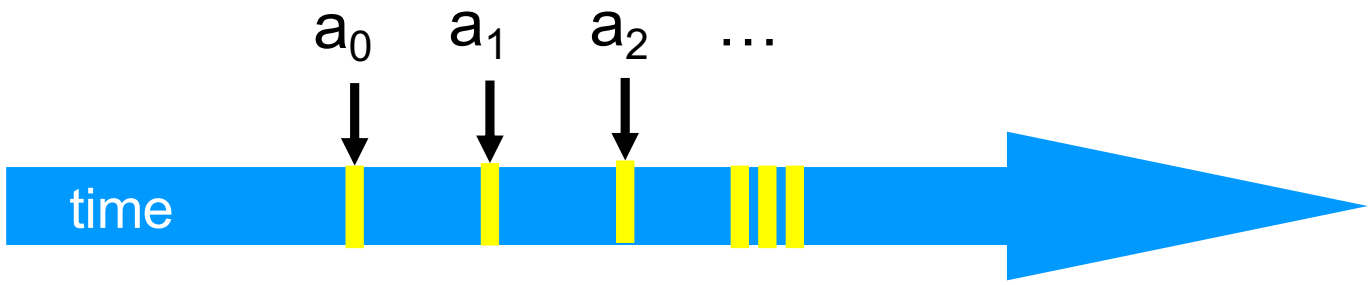
Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - *Otherwise it results in a race condition!*

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Threads

- A ***thread*** A is (formally) a sequence a_0, a_1, ... of events
  - Notation: a_0 ➔ a_1 indicates order



Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Concurrency

- Thread A

- Thread B

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Array Sum using Pthread (Version-1)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

We went through this version of the parallel array sum using pthreads in the previous lecture

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread (Version-2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum; result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Any thread interleavings?

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Do you see any issues in this version of the parallel array sum?

Array Sum using Pthread (Version-2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;   result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Race condition !!!

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Critical Sections and Mutual Exclusion

- Critical section is the code (block of code) that should be executed by only one thread at a time
- **Mutex locks** enforce mutual exclusion in Pthreads
  - Mutex lock states: locked and unlocked
  - Only one thread can lock a mutex lock at any particular time
- Using mutex locks
  - Request lock before executing critical section
  - Enter critical section when lock granted
  - Release lock when leaving critical section

Pthread Mutex Locks

- Initialize the mutex variable (statically)
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Lock the mutex
  - `pthread_mutex_lock(&mutex);`
- Unlock the mutex
  - `pthread_mutex_unlock(&mutex);`

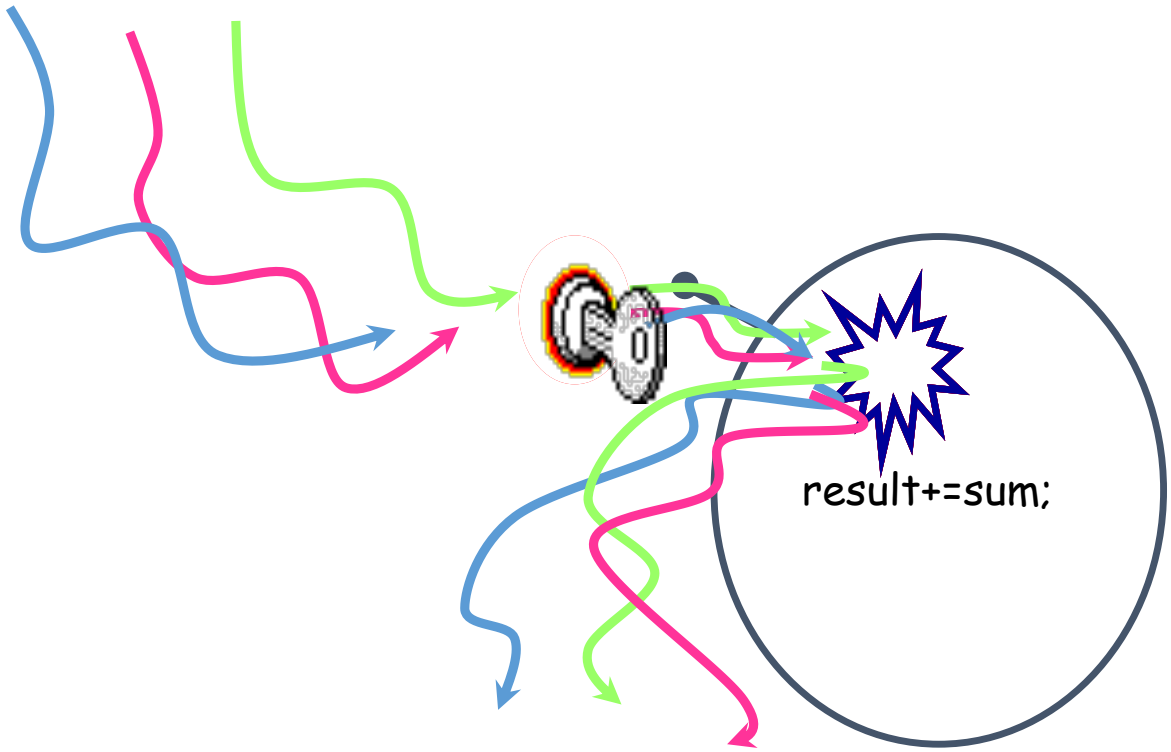
Array Sum using Pthread (Version-2)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  pthread_mutex_lock(&m);
  result += sum;
  pthread_mutex_unlock(&m);
}

typedef struct {
  int low;
  int high;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  array_sum(t->low, t->high);
  return NULL;
}
```

Race condition is fixed using mutual exclusion

```c
int main(int argc, char *argv[]) {
  if (SIZE < 1024) {
    array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i]);
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Visualizing Mutual Exclusion



- Only one **thread** can get the “key” to enter the critical section
- Rest all **threads** will be queued to get the key

The Issue with the Locks

- Threads waiting for acquiring the lock continue wasting CPU cycles while spinning to acquire lock currently owned by another thread
  - Bigger problem if there are several threads waiting to acquire the same lock
- Solution
  - Instead of blocking, try (attempt) to acquire the lock
  - If failed to acquire then do something else
  - ```
    if(pthread_mutex_trylock(&mutex) == 0) {
        /* Critical Section */
    } else {
        /* Do something else */
    }
    ```
    ```

Condition Variables for Synchronization

- **Condition variable: associated with a predicate and a mutex**
- Using a condition variable
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable

    if predicate is false, then wait on the condition variable

    waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - that thread can signal the condition variable to either

    wake one waiting thread

    wake all waiting threads
    - when thread releases the mutex, it is passed to first waiter

Pthread Condition Variable APIs

```
/* initialize a condition variable (statically) */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);          // Wake one thread
int pthread_cond_broadcast(pthread_cond_t *cond);      // Wake all threads
```

Wait/Notify Sequence in Pthread

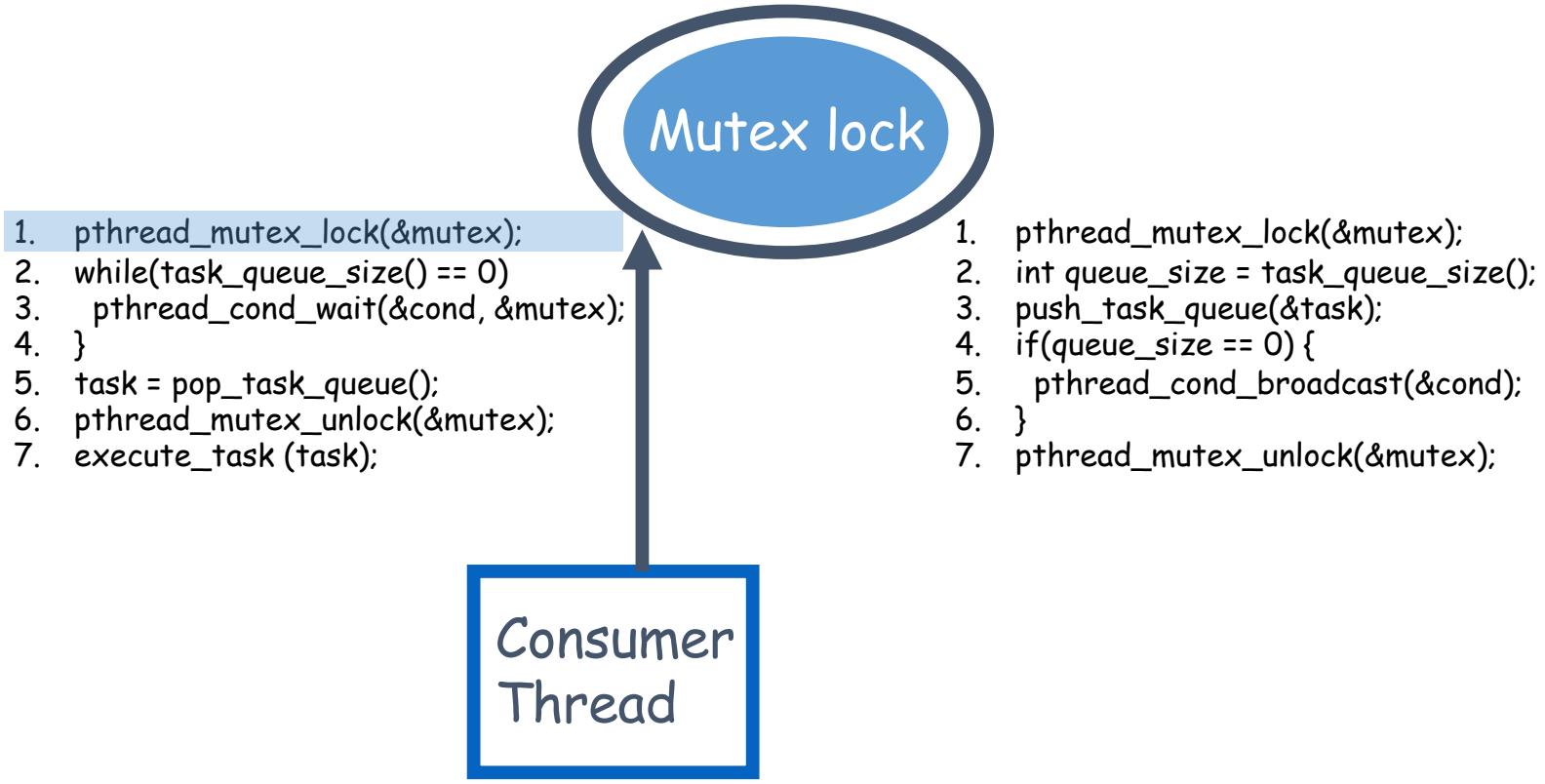
```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

Consumer(s)

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Producer

Wait/Notify Sequence in Pthread



Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Wait/Notify Sequence in Pthread

Mutex lock

```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Wait/Notify Sequence in Pthread

Mutex lock

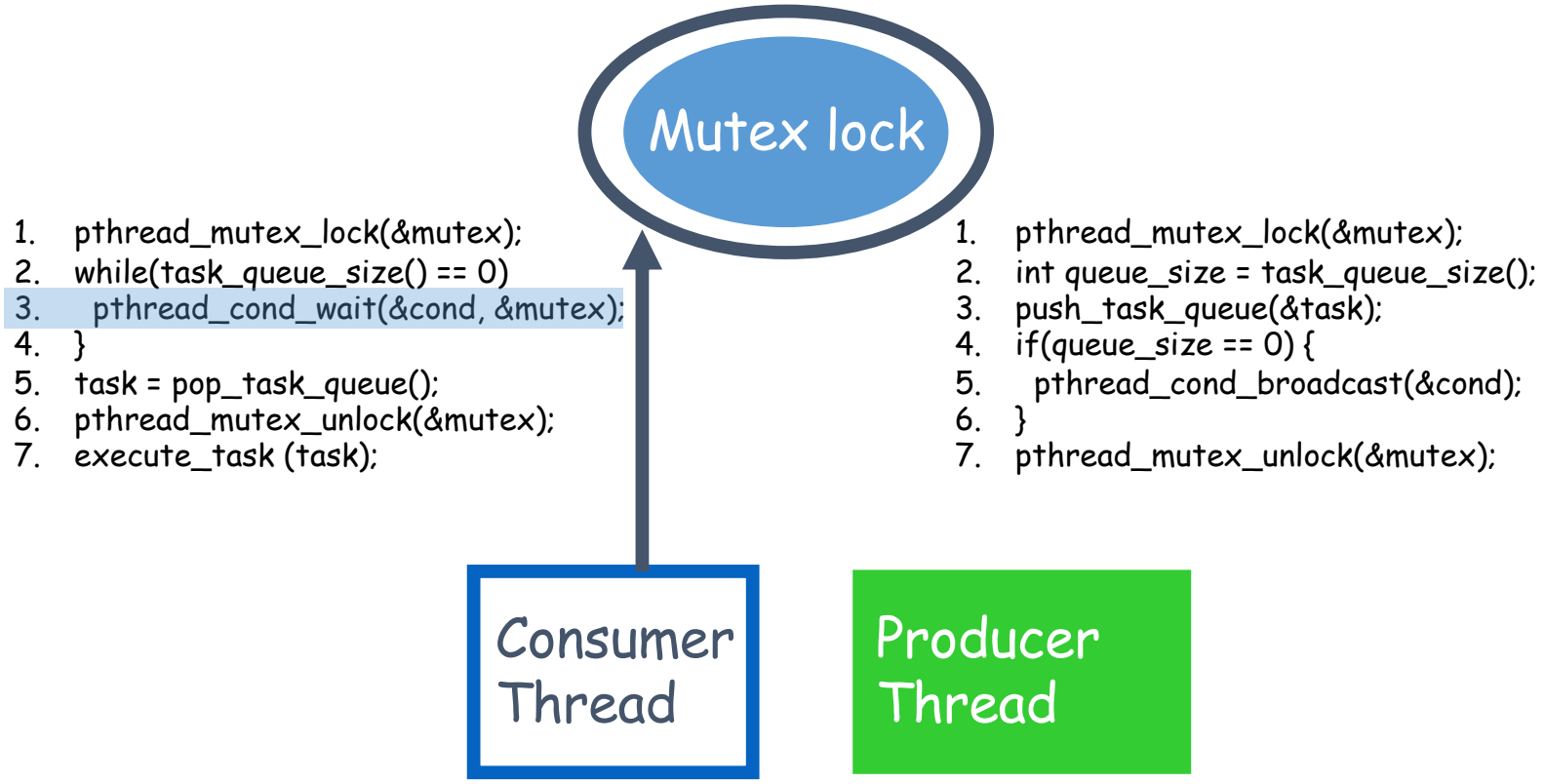
```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Wait/Notify Sequence in Pthread



Wait/Notify Sequence in Pthread


```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Wait/Notify Sequence in Pthread

Mutex lock

```
1.  pthread_mutex_lock(&mutex);
2.  while(task_queue_size() == 0)
3.    pthread_cond_wait(&cond, &mutex);
4.  }
5.  task = pop_task_queue();
6.  pthread_mutex_unlock(&mutex);
7.  execute_task (task);
```

```
1.  pthread_mutex_lock(&mutex);
2.  int queue_size = task_queue_size();
3.  push_task_queue(&task);
4.  if(queue_size == 0) {
5.    pthread_cond_broadcast(&cond);
6.  }
7.  pthread_mutex_unlock(&mutex);
```

Consumer Thread

Producer Thread

Let's Revisit Producer-Consumer (Lecture #8)

```
typedef struct cookiejar_t {
    int cookie;
    sem_t jar_empty;
    sem_t jar_full;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    sem_init(&cookiejar->jar_empty, 1, 1);
    sem_init(&cookiejar->jar_full, 1, 0);
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    sem_destroy(&cookiejar->jar_empty);
    sem_destroy(&cookiejar->jar_full);
    cleanup();
    return 0;
}
```

```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

Same Program using Threads

```
typedef struct cookiejar_t {
    int cookie;
    sem_t jar_empty;
    sem_t jar_full;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = (cookiejar_t*)malloc(sizeof(cookiejar_t));
    sem_init(&cookiejar->jar_empty, 1, 1);
    sem_init(&cookiejar->jar_full, 1, 0);
    pthread_t homer_tid, marge_tid;
    pthread_create(&homer_tid, NULL, homer, NULL);
    pthread_create(&marge_tid, NULL, marge, NULL);
    pthread_join(&homer_tid, NULL);
    pthread_join(&marge_tid, NULL);
    sem_destroy(&cookiejar->jar_empty);
    sem_destroy(&cookiejar->jar_full);
    free(cookiejar);
    return 0;
}
```

```
void *homer(void* args) {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }

}
```

```
void *marge(void* args) {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }

}
```

Same Program using Threads + Mutex/Cond

```
typedef struct cookiejar_t {
    int cookie;
    int empty;
} cookiejar_t;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
cookiejar_t* cookiejar;

int main() {
    cookiejar = (cookiejar_t*)malloc(sizeof(cookiejar_t));
    cookiejar->empty=1;
    pthread_t homer_tid, marge_tid;
    pthread_create(&homer_tid, NULL, homer, NULL);
    pthread_create(&marge_tid, NULL, marge, NULL);
    pthread_join(&homer_tid, NULL);
    pthread_join(&marge_tid, NULL);
    free(cookiejar);
    return 0;
}
```

```
void *homer(void* args) {
    for(int i=0; i<5; i++) {
        pthread_mutex_lock(&mutex);
        while(cookiejar->empty == 1) {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        cookiejar->empty = 1;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *marge(void* args) {
    for(int i=0; i<5; i++) {
        pthread_mutex_lock(&mutex);
        while(cookiejar->empty == 0) {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        cookiejar->empty = 0;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

Semaphore vs. Condition Variables

- Both are used in situations where thread(s) wait on some conditions to be satisfied for starting/resume their execution
- Synchronization implementation
  - Semaphore can be used alone
  - Condition variable must be used along with a mutex variable
- Unblocking multiple threads in one shot
  - Not possible using semaphore (each sem_wait decrements the semaphore)
  - Easily achieved using pthread_cond_broadcast

Next Lecture

- Deadlock avoidance and thread pools
  - Last remaining topic in multithreading