CS214 Recitation Sec.7

Nov. 14, 2017

Topics

1. HW6: Threads Synchronization

2. Condition Variables

3. Two ways to implement *Producer-consumer Problem*

HW6 - Threads Synchronization

Write a function that uses threads to *synchronize printing* between them to print out a triangle.

Make *two threads*, one to print out even rows, one to print out odd. The goal is to print out:

```
*
**
***
****
*****
******
```

where each line is printed by a different thread. Have **thread 0** print out the even lines (2 stars, 4 stars, etc) and **thread 1** print out the odd ones (1 star, 3 stars, etc).

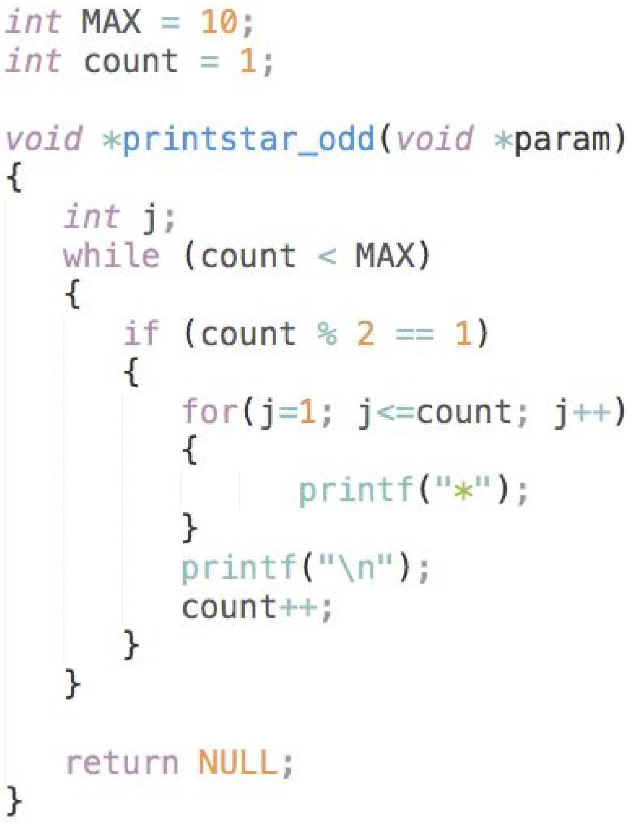
HW6 - Threads Synchronization (Cont.)

At first, just run them with no synchronization. You'll get an interleaving of rows … likely you'll get a batch of rows and then another batch of rows.

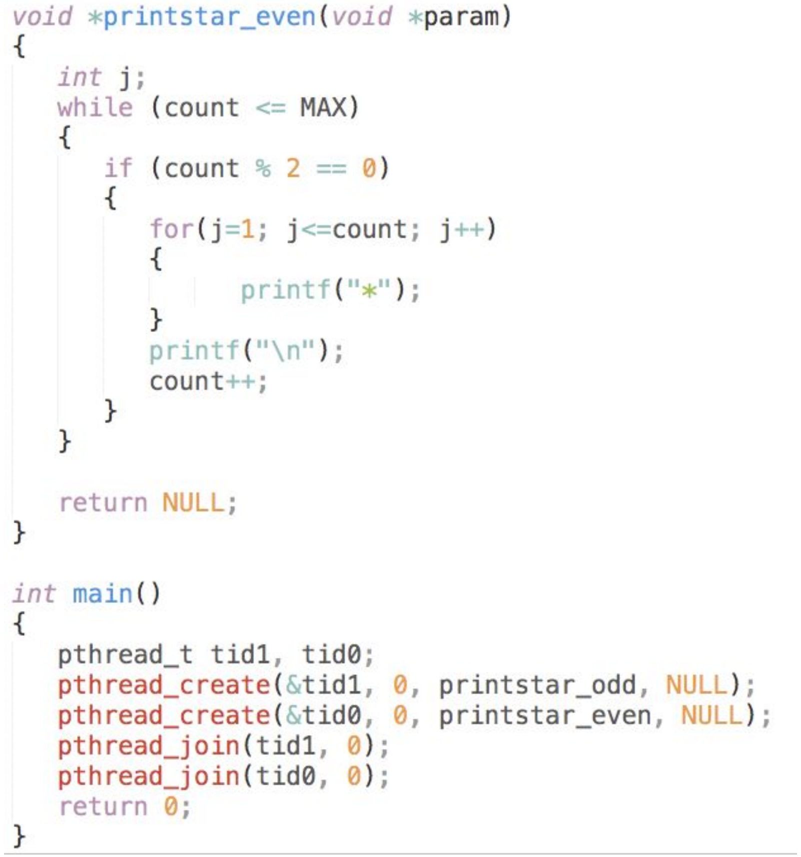
Next add a pair of mutexes to trade off control between the threads. They should trade off using the mutexes to synchronize between them.

Sometimes thread 0 may be scheduled before thread 1 and you get the rows printed in the wrong order. You need to make sure the first thread gets the mutex first - but you can't, really since you do not have control of the scheduler.

nosyc.c - correct output (why?)

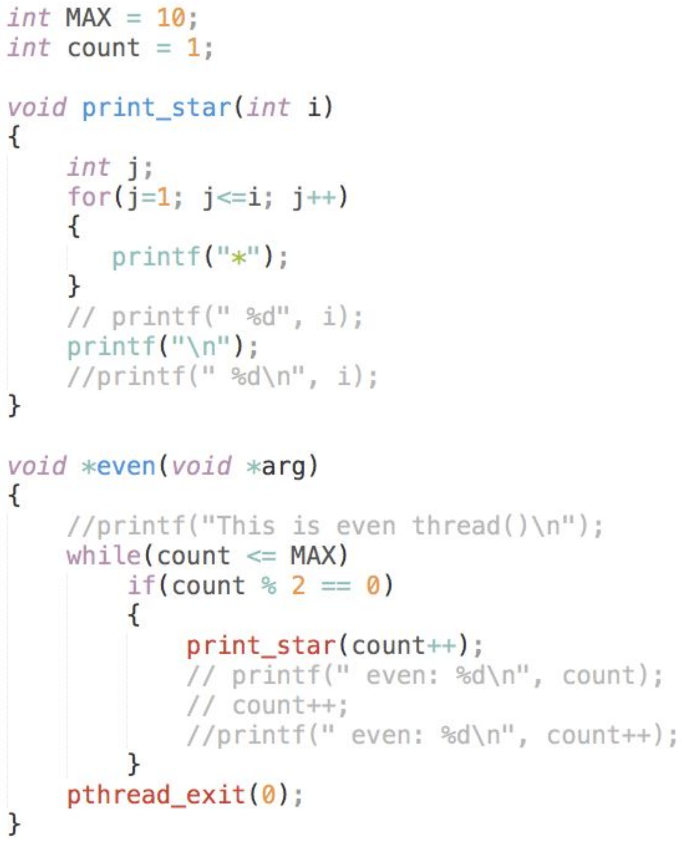


```c
int MAX = 10;
int count = 1;

void *printstar_odd(void *param)
{
    int j;
    while (count < MAX)
    {
        if (count % 2 == 1)
        {
            for(j=1; j<=count; j++)
            {
                printf("*");
            }
            printf("\n");
            count++;
        }
    }

    return NULL;
}
```

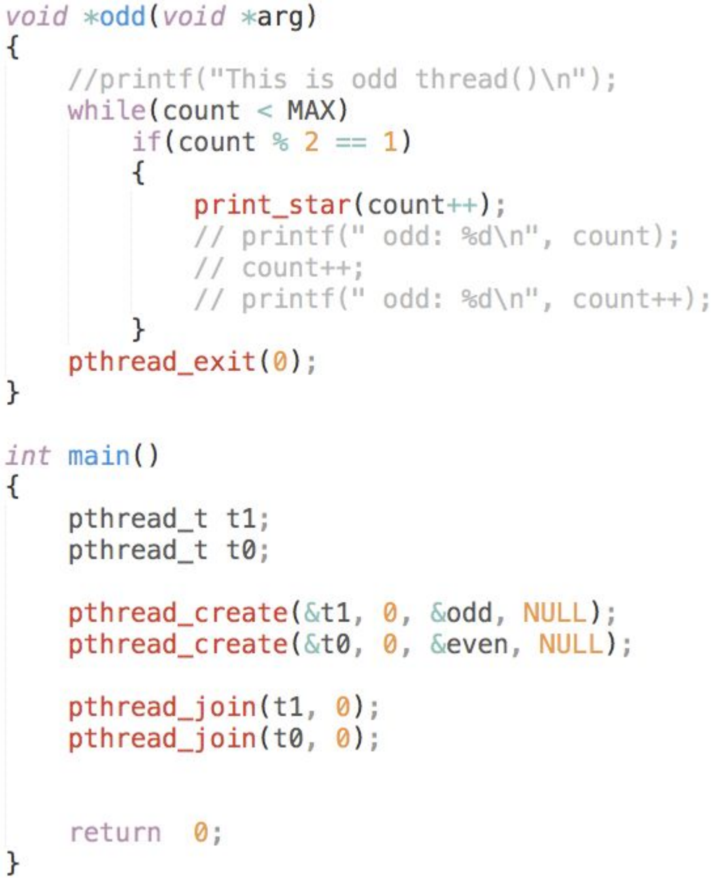


```c
void *printstar_even(void *param)
{
    int j;
    while (count <= MAX)
    {
        if (count % 2 == 0)
        {
            for(j=1; j<=count; j++)
            {
                printf("*");
            }
            printf("\n");
            count++;
        }
    }

    return NULL;
}

int main()
{
    pthread_t tid1, tid0;
    pthread_create(&tid1, 0, printstar_odd, NULL);
    pthread_create(&tid0, 0, printstar_even, NULL);
    pthread_join(tid1, 0);
    pthread_join(tid0, 0);
    return 0;
}
```

nosyc2.c - interleaving of rows (why?)

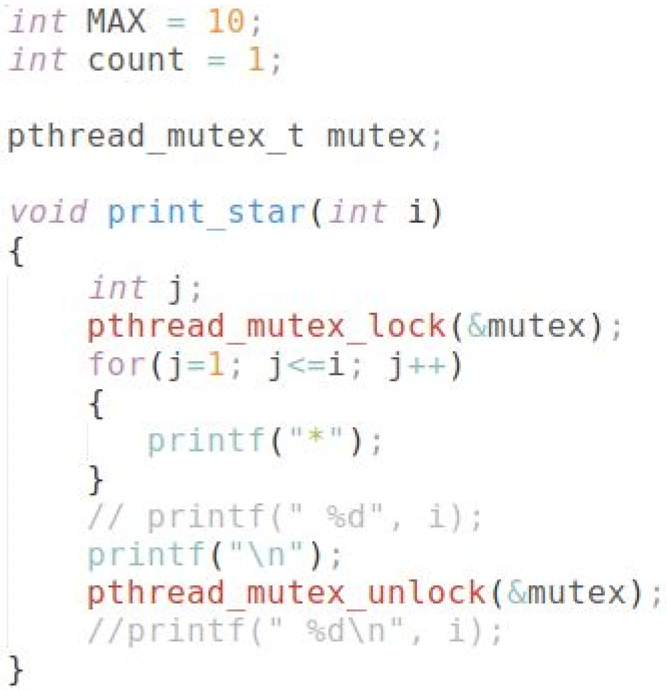


```c
int MAX = 10;
int count = 1;

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
       printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}

void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
        if(count % 2 == 0)
        {
            print_star(count++);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
    pthread_exit(0);
}
```

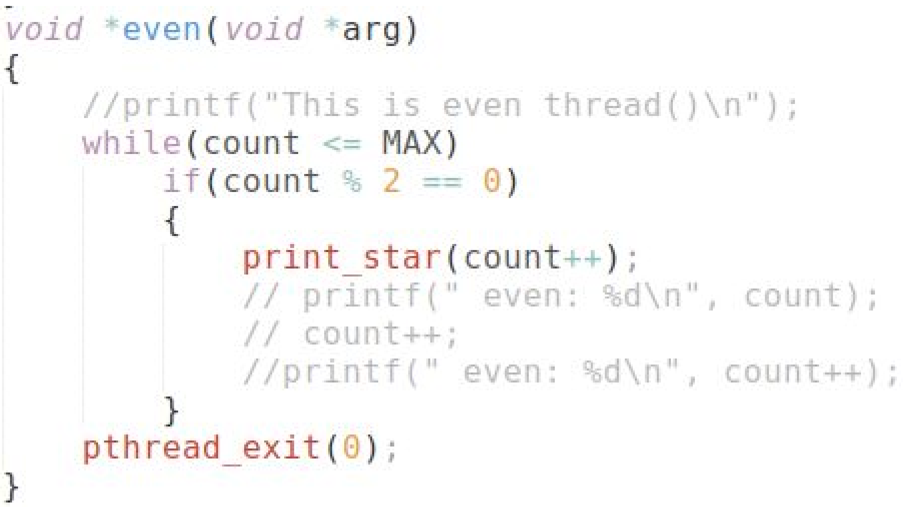


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
        if(count % 2 == 1)
        {
            print_star(count++);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
    pthread_exit(0);
}

int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);


    return  0;
}
```

syc_m.c - right # of *, wrong row order (1)

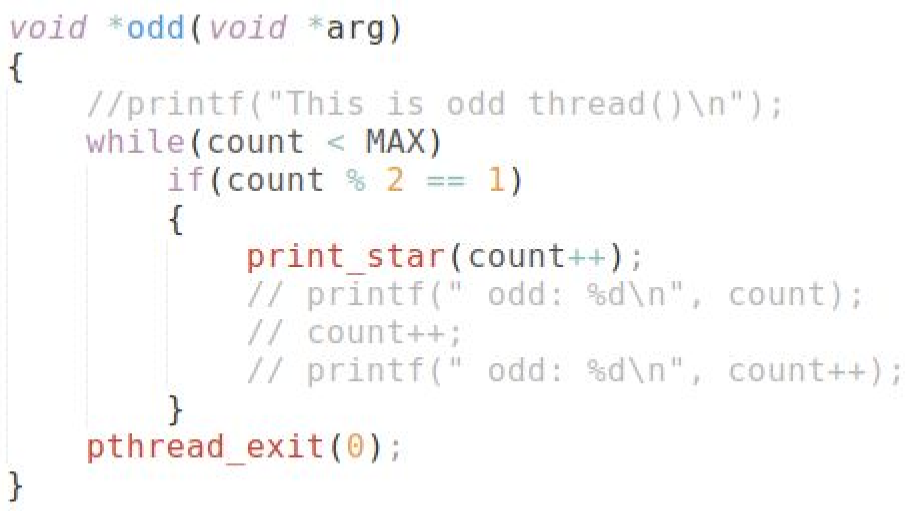


```
int MAX = 10;
int count = 1;

pthread_mutex_t mutex;

void print_star(int i)
{
    int j;
    pthread_mutex_lock(&mutex);
    for(j=1; j<=i; j++)
    {
        printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    pthread_mutex_unlock(&mutex);
    //printf(" %d\n", i);
}
```

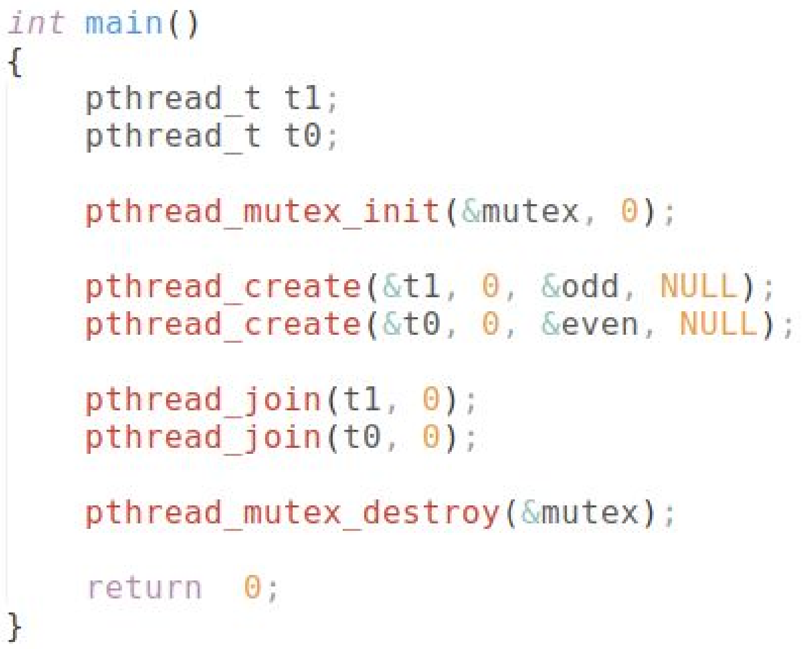


```
void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
        if(count % 2 == 0)
        {
            print_star(count++);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
    pthread_exit(0);
}
```

syc_m.c - right # of *, wrong row order (2)

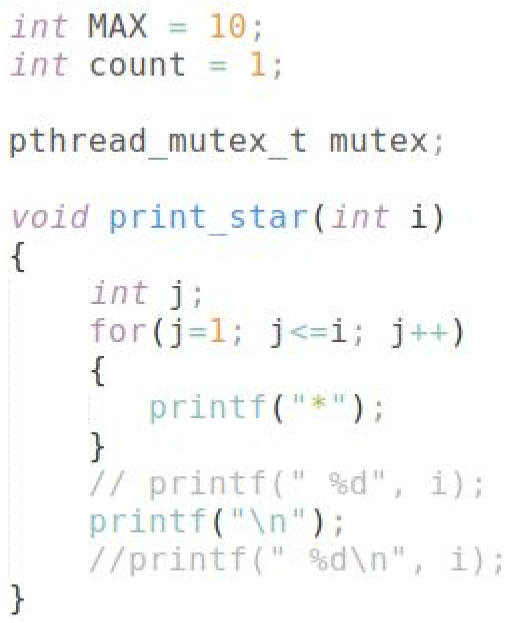


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
        if(count % 2 == 1)
        {
            print_star(count++);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
    pthread_exit(0);
}
```

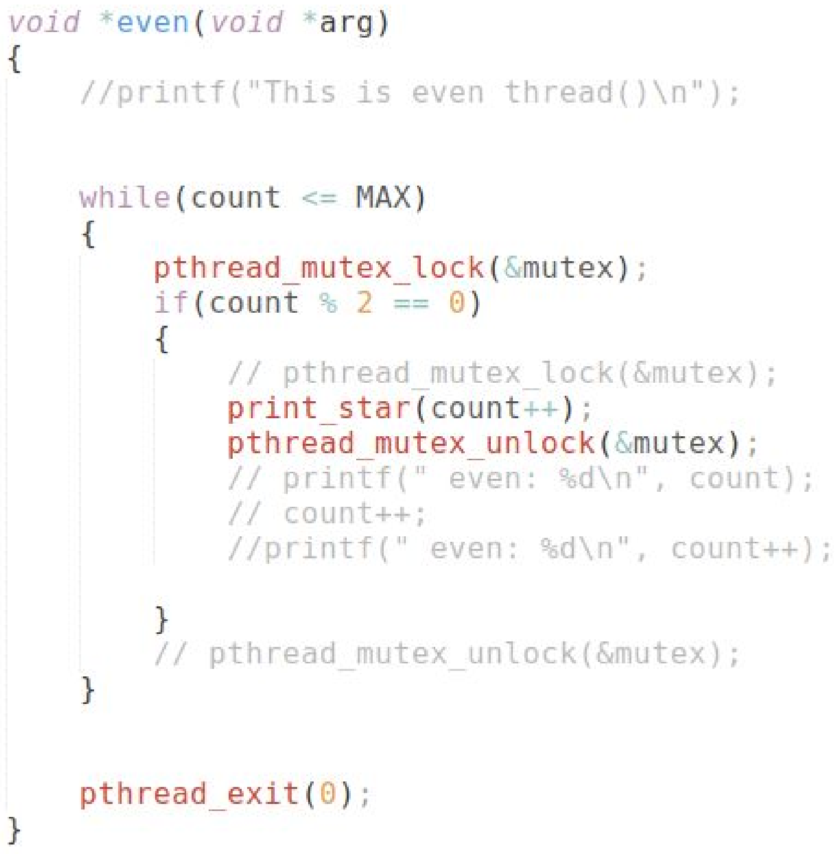


```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_mutex_init(&mutex, 0);

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);

    return  0;
}
```

syc_m2.c - get stuck (1)

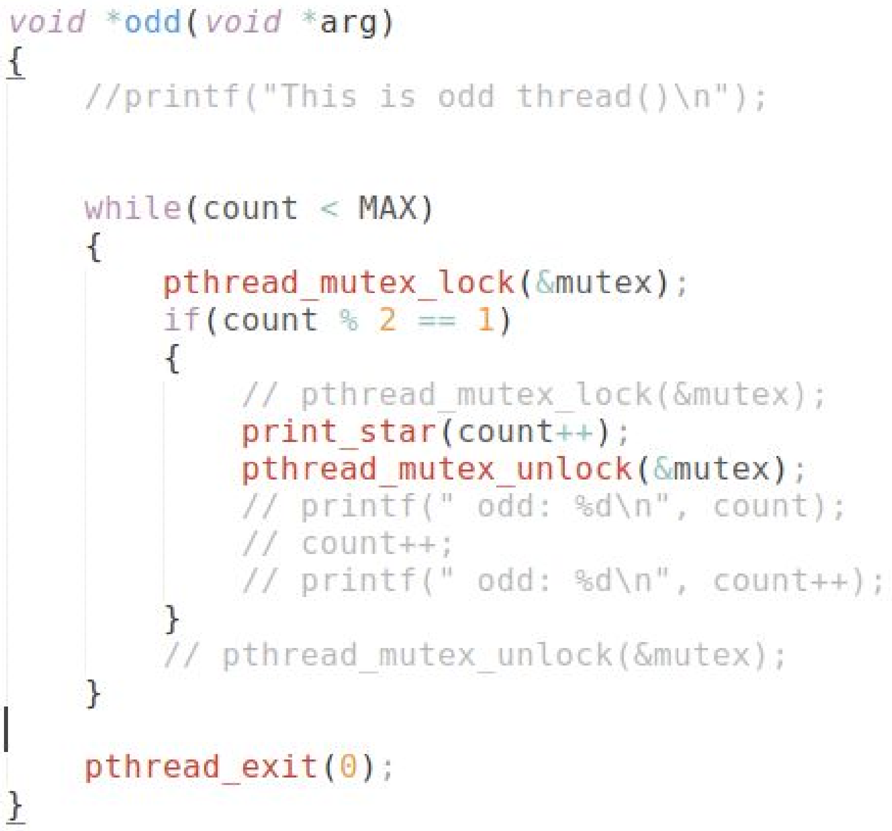


```c
int MAX = 10;
int count = 1;

pthread_mutex_t mutex;

void print_star(int i)
{
        int j;
        for(j=1; j<=i; j++)
        {
            printf("*");
        }
        // printf(" %d", i);
        printf("\n");
        //printf(" %d\n", i);
}
```

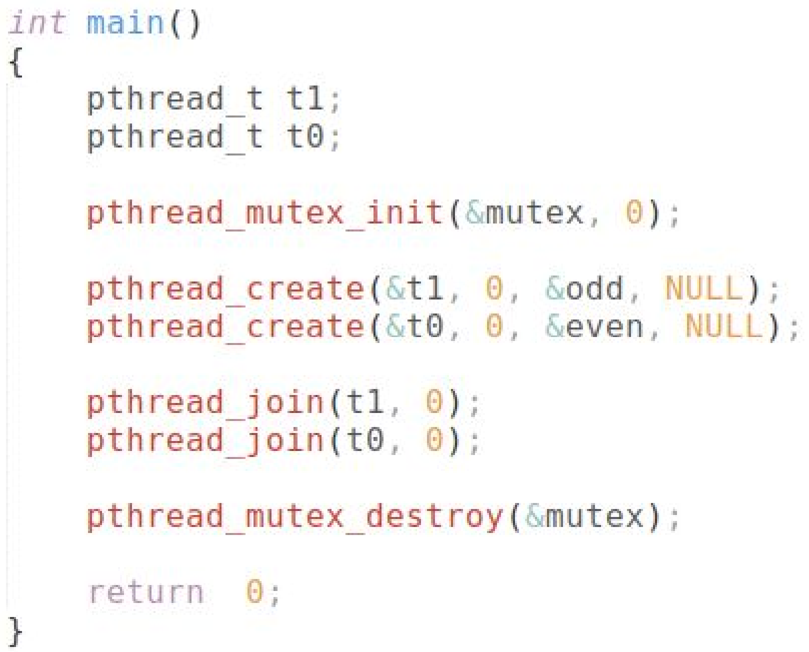


```c
void *even(void *arg)
{
    //printf("This is even thread()\n");


    while(count <= MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 0)
        {
            // pthread_mutex_lock(&mutex);
            print_star(count++);
            pthread_mutex_unlock(&mutex);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);

        }
        // pthread_mutex_unlock(&mutex);
    }


    pthread_exit(0);
}
```

syc_m2.c - get stuck (2)

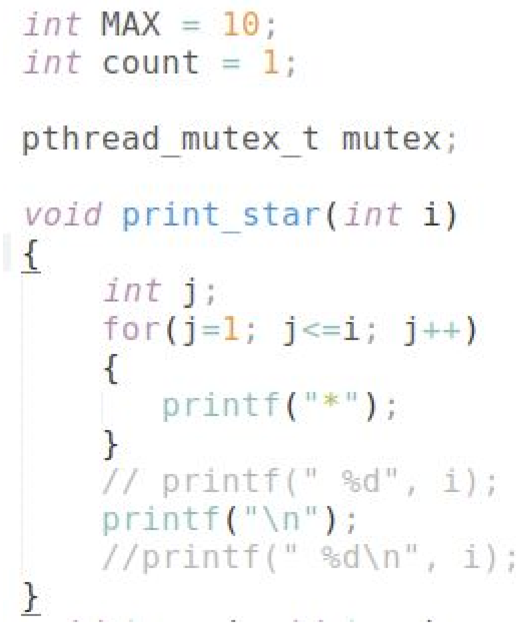


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");


    while(count < MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 1)
        {
            // pthread_mutex_lock(&mutex);
            print_star(count++);
            pthread_mutex_unlock(&mutex);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
        // pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

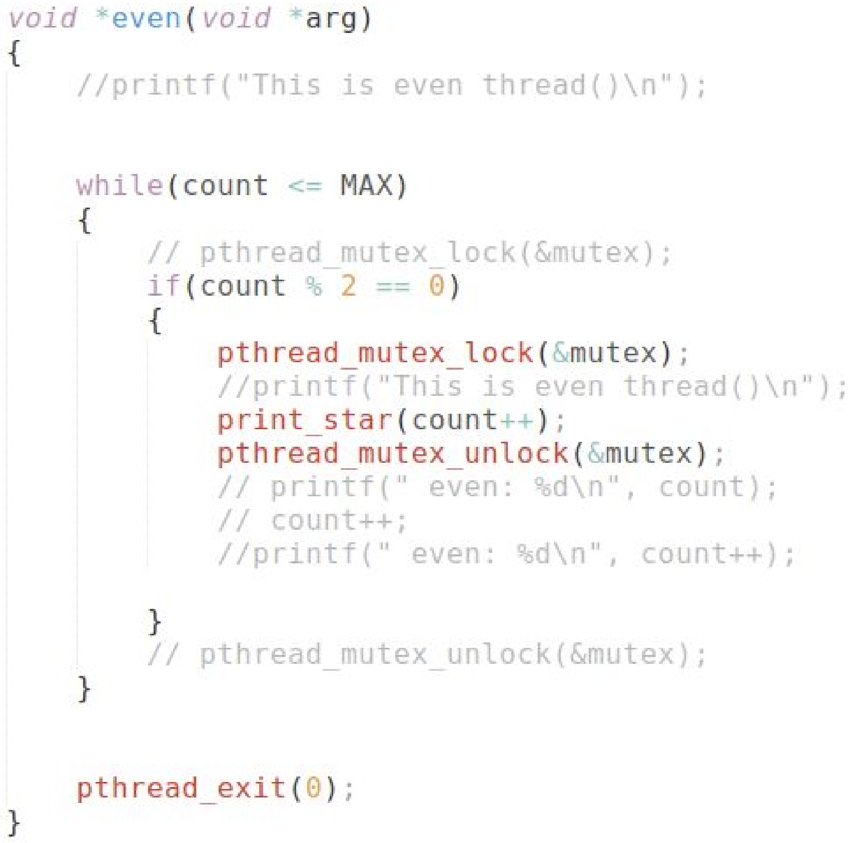


```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_mutex_init(&mutex, 0);

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);

    return  0;
}
```

syc_m3.c - right output (1)

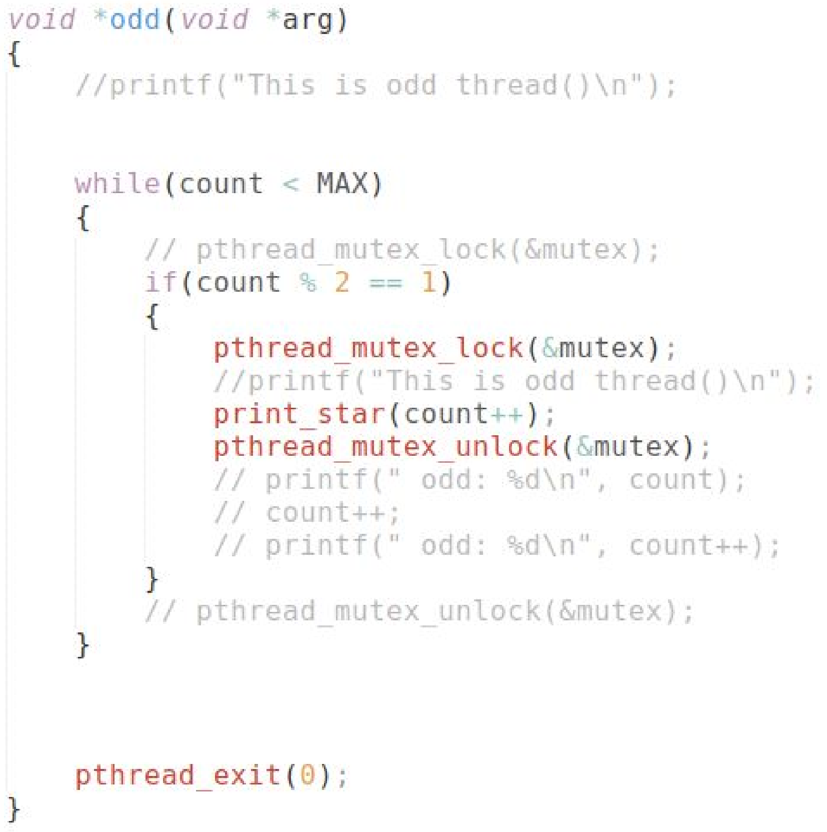


```
int MAX = 10;
int count = 1;

pthread_mutex_t mutex;

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
        printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}
```

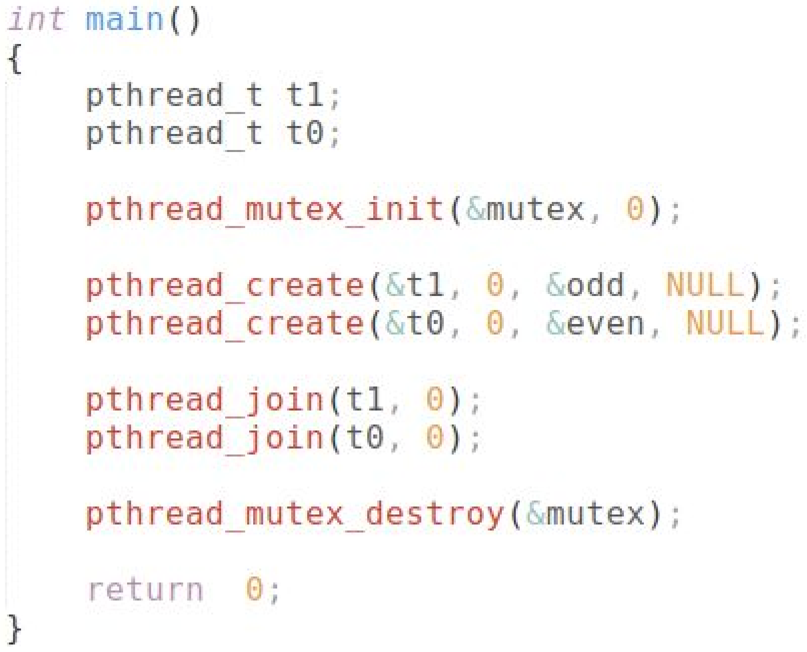


```
void *even(void *arg)
{
    //printf("This is even thread()\n");


    while(count <= MAX)
    {
        // pthread_mutex_lock(&mutex);
        if(count % 2 == 0)
        {
            pthread_mutex_lock(&mutex);
            //printf("This is even thread()\n");
            print_star(count++);
            pthread_mutex_unlock(&mutex);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);

        }
        // pthread_mutex_unlock(&mutex);
    }


    pthread_exit(0);
}
```

syc_m3.c - right output (2)



```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");


    while(count < MAX)
    {
        // pthread_mutex_lock(&mutex);
        if(count % 2 == 1)
        {
            pthread_mutex_lock(&mutex);
            //printf("This is odd thread()\n");
            print_star(count++);
            pthread_mutex_unlock(&mutex);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
        // pthread_mutex_unlock(&mutex);
    }



    pthread_exit(0);
}
```



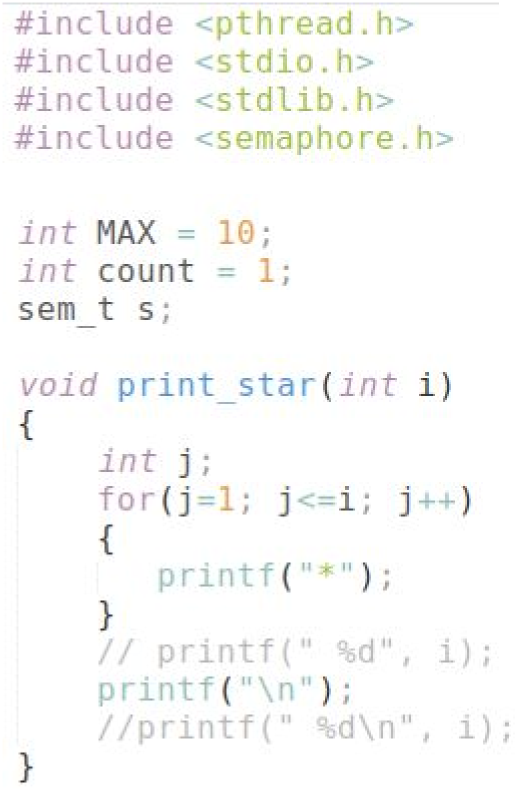
```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_mutex_init(&mutex, 0);

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);

    return  0;
}
```

HW6 - Threads Synchronization (Cont.)

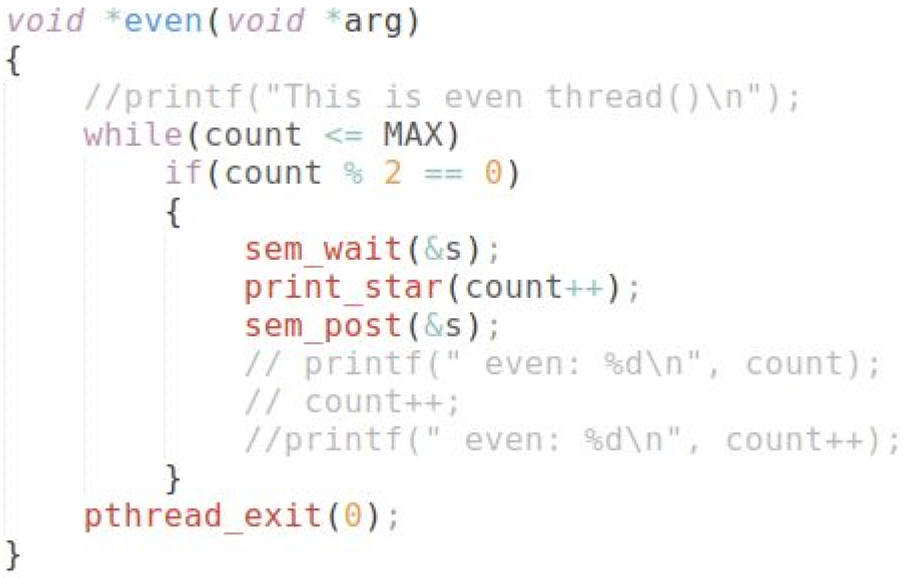
Change the mutex into a binary semaphore. This way if the wrong thread starts first, it will block until the other thread gets to run.

Thread 1 ought to notify/produce and thread 0 ought to wait/consume. This way you can trade control in an intentional manner.

syc_s.c - right output (1)

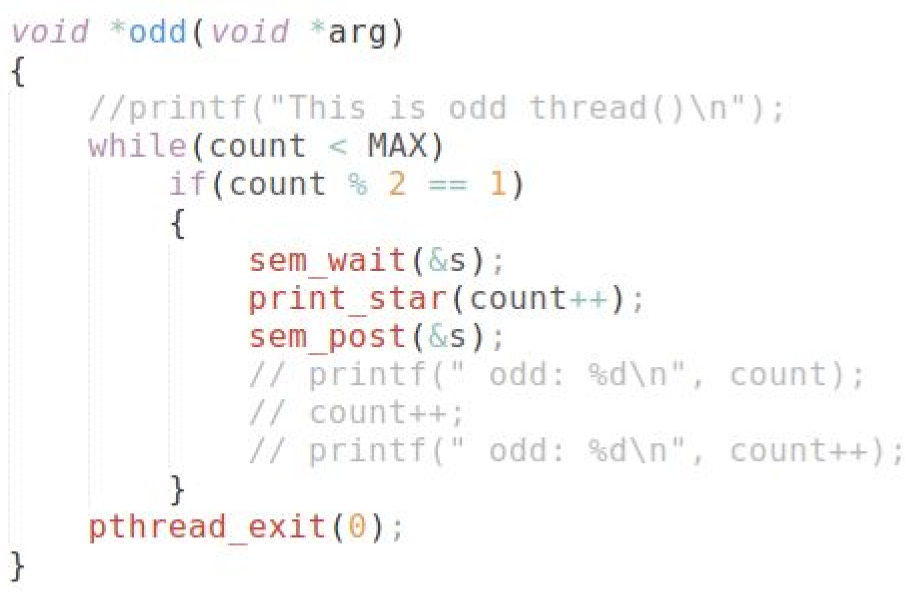


```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>

int MAX = 10;
int count = 1;
sem_t s;

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
        printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}
```

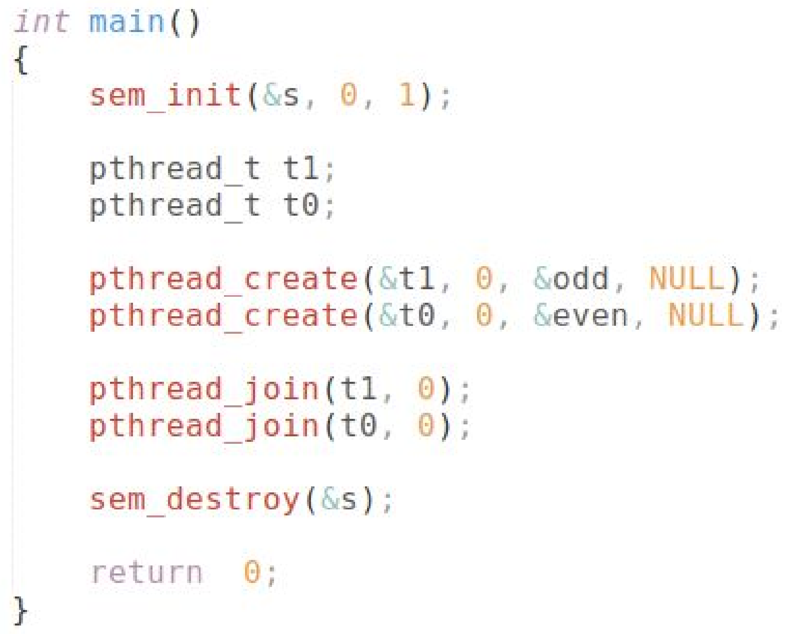


```
void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
        if(count % 2 == 0)
        {
            sem_wait(&s);
            print_star(count++);
            sem_post(&s);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
    pthread_exit(0);
}
```

syc_s.c - right output (2)



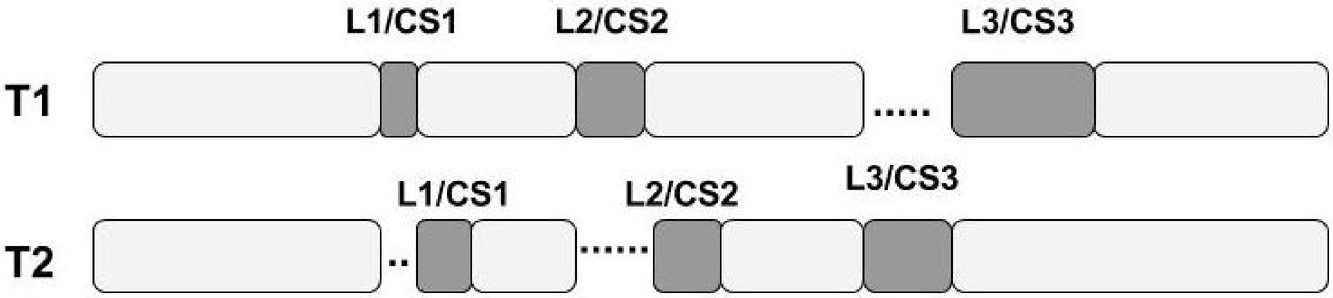
```
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
        if(count % 2 == 1)
        {
            sem_wait(&s);
            print_star(count++);
            sem_post(&s);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
    pthread_exit(0);
}
```



```
int main()
{
    sem_init(&s, 0, 1);

    pthread_t t1;
    pthread_t t0;

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    sem_destroy(&s);

    return  0;
}
```

Recall: Mutex

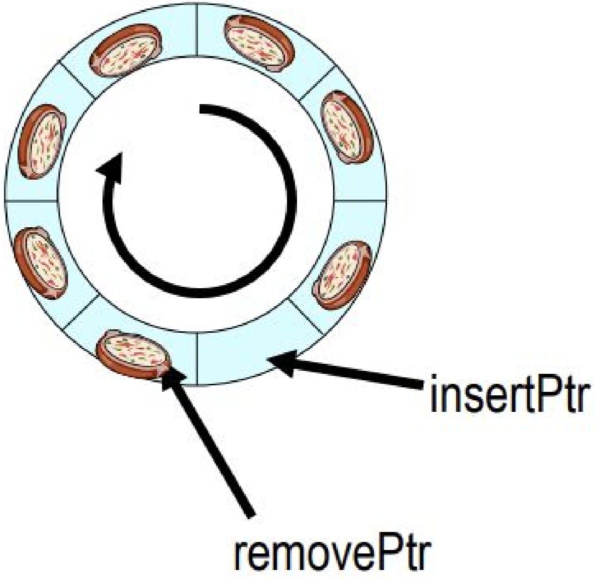
- **Mutual exclusion** (Mutex locks) : used for *exclusive access* to a shared resource (critical section)
- **operations**: *pthread_mutex_init(&m, NULL)*, *pthread_mutex_lock(&m)*, *pthread_mutex_unlock(&m)*, *pthread_mutex_destroy(&m)*,



Recall: Semaphore

- **Semaphores** : generalization of mutexes: *counting number* of available “resources”

- **operations**: decrement - *sem_wait (&s)*, increment - *sem_post (&s)*, initialize - *sem_init (&s, 0, 10)*, destroy - *sem_destroy (&s)*

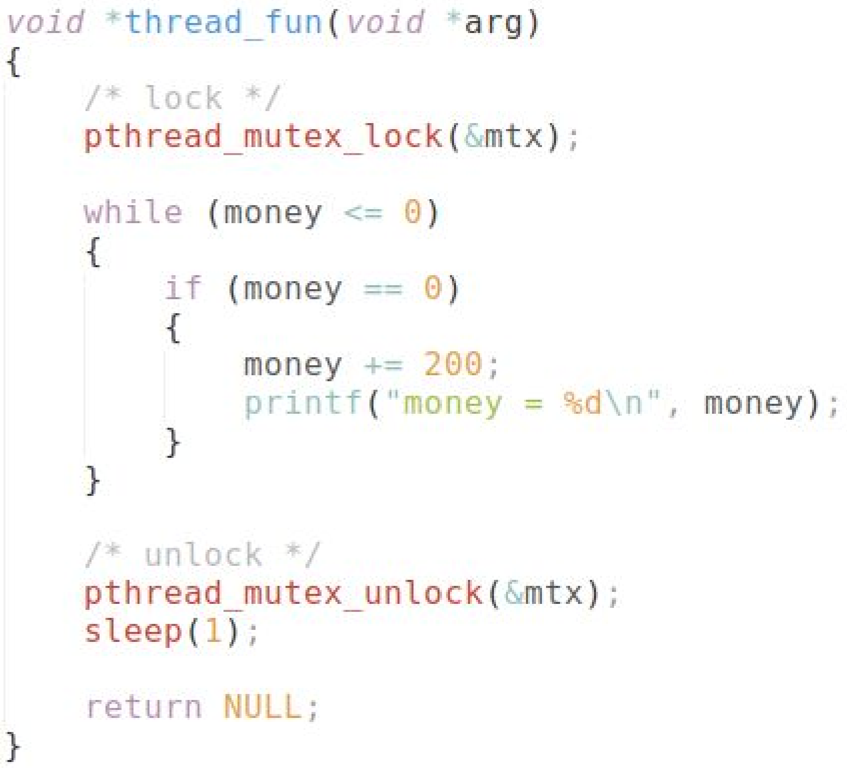


Condition Variables

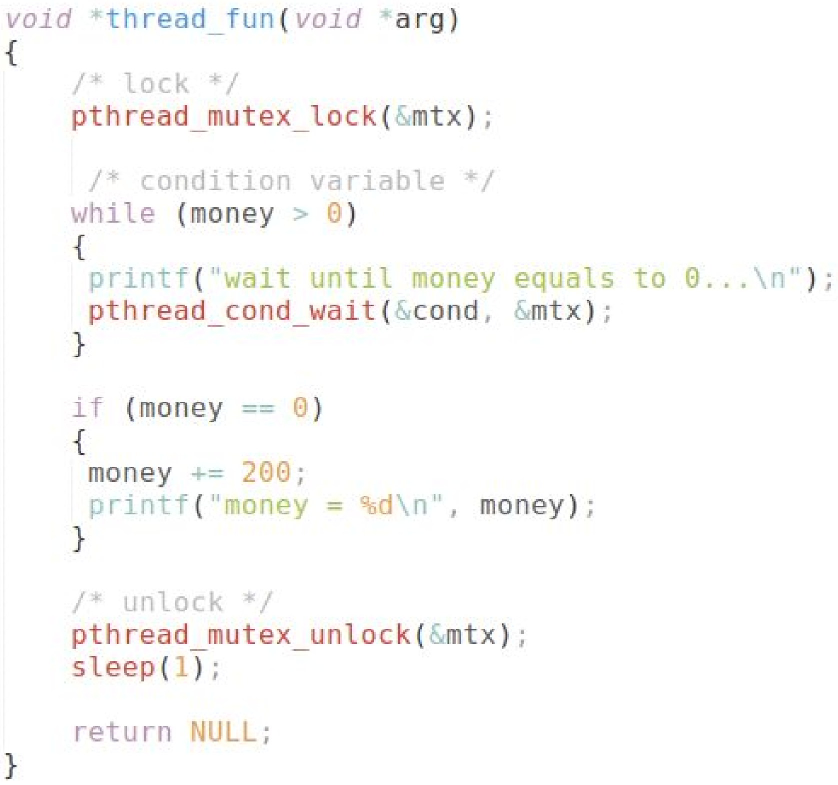
- **Conditional variables** : *wait for a specific event* to happen, tied to a mutex for exclusive access

- **operations**: *pthread_cond_wait(&cv, &m)* - wait for event, *pthread_cond_signal(&cv) -* signal occurrence of event, *pthread_cond_broadcast(&cv) -* broadcast occurrence of event, *pthread_cond_init(&cv, NULL) -* initialize a condition variable, *pthread_cond_destroy(&cv) -* destroy a condition variable

Why use condition variables?



```c
void *thread_fun(void *arg)
{
    /* lock */
    pthread_mutex_lock(&mtx);

    while (money <= 0)
    {
        if (money == 0)
        {
            money += 200;
            printf("money = %d\n", money);
        }
    }

    /* unlock */
    pthread_mutex_unlock(&mtx);
    sleep(1);

    return NULL;
}
```

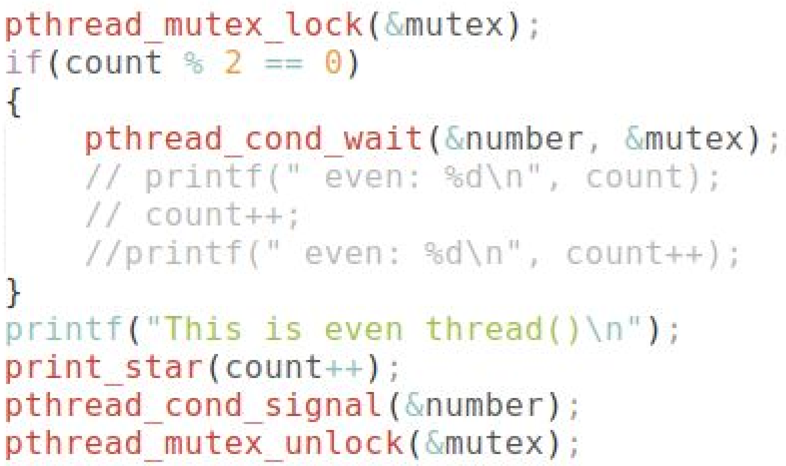


```c
void *thread_fun(void *arg)
{
    /* lock */
    pthread_mutex_lock(&mtx);

     /* condition variable */
    while (money > 0)
    {
     printf("wait until money equals to 0...\n");
     pthread_cond_wait(&cond, &mtx);
    }

    if (money == 0)
    {
     money += 200;
     printf("money = %d\n", money);
    }

    /* unlock */
    pthread_mutex_unlock(&mtx);
    sleep(1);

    return NULL;
}
```

Use with mutexes

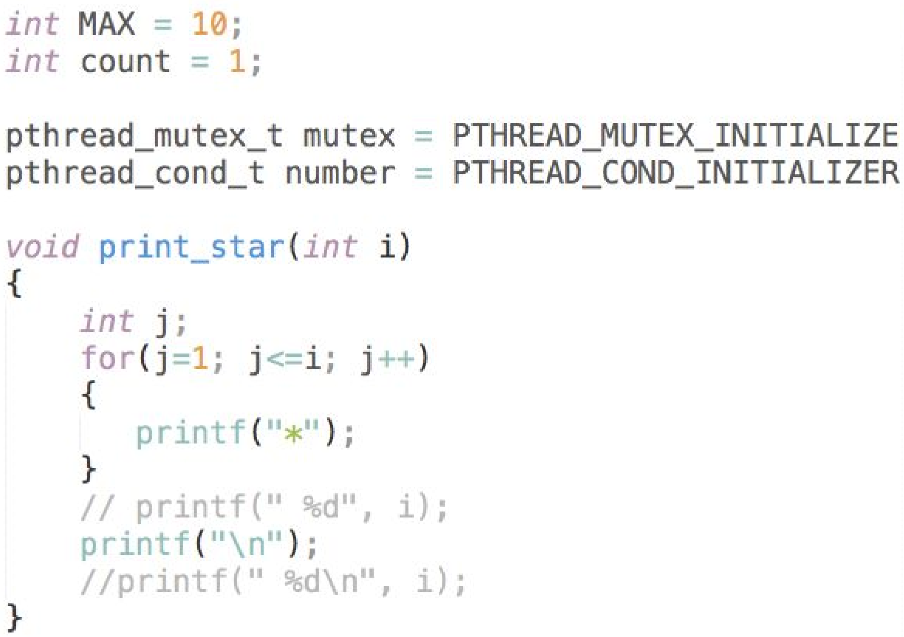
Condition variables usually uses with **mutexes**

- avoid missing *signal of occurance*
- *updating the program state* requires mutual exclusion

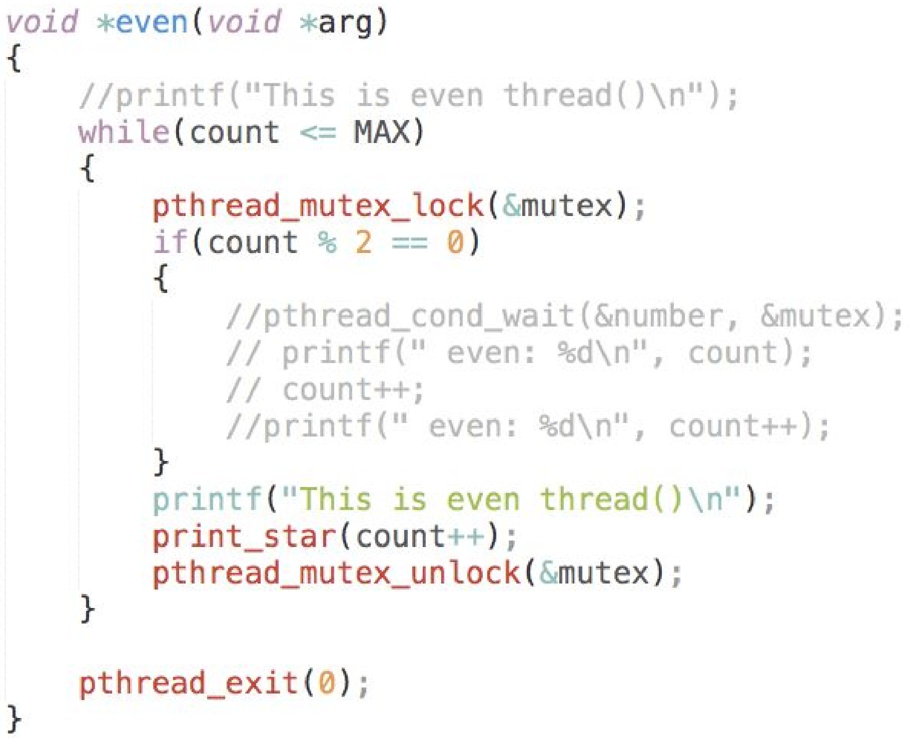


```
pthread_mutex_lock(&mutex);
if(count % 2 == 0)
{
    pthread_cond_wait(&number, &mutex);
    // printf(" even: %d\n", count);
    // count++;
    //printf(" even: %d\n", count++);
}
printf("This is even thread()\n");
print_star(count++);
pthread_cond_signal(&number);
pthread_mutex_unlock(&mutex);
```

syc_c.c - right # of *, wrong row order (1)

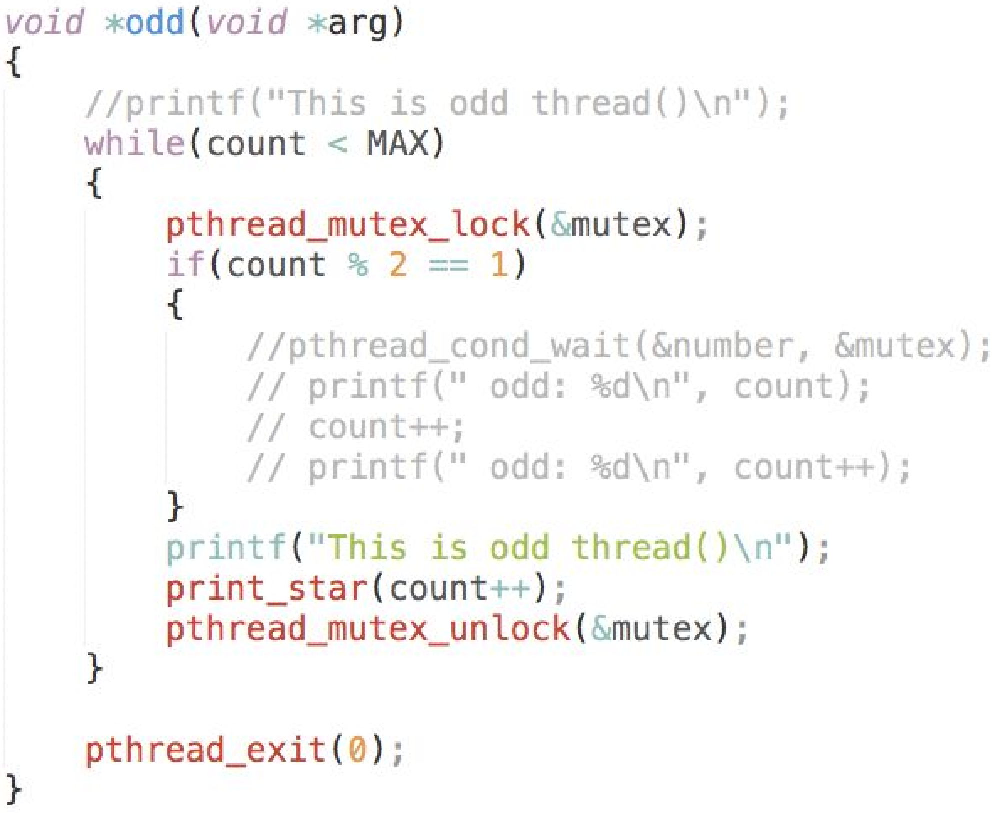


```c
int MAX = 10;
int count = 1;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZE
pthread_cond_t number = PTHREAD_COND_INITIALIZER

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
       printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}
```

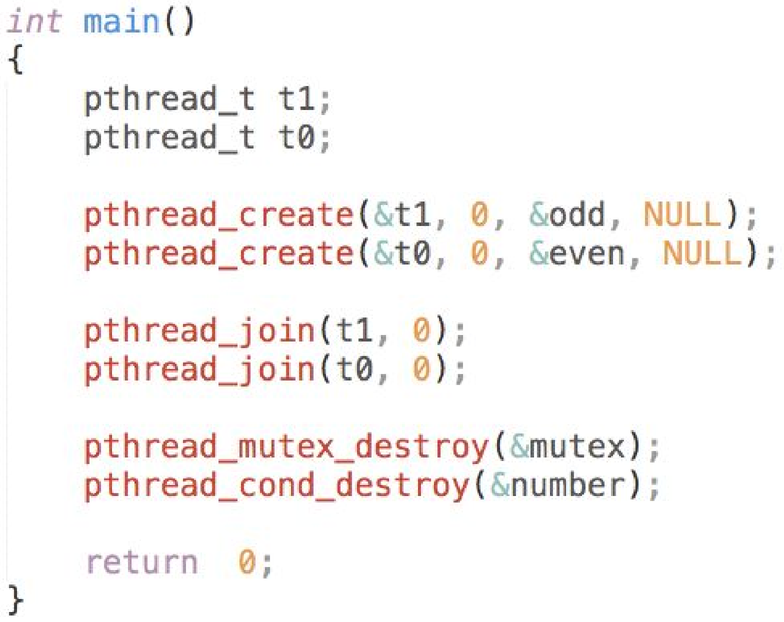


```c
void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 0)
        {
            //pthread_cond_wait(&number, &mutex);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
        printf("This is even thread()\n");
        print_star(count++);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

syc_c.c - right # of *, wrong row order (2)

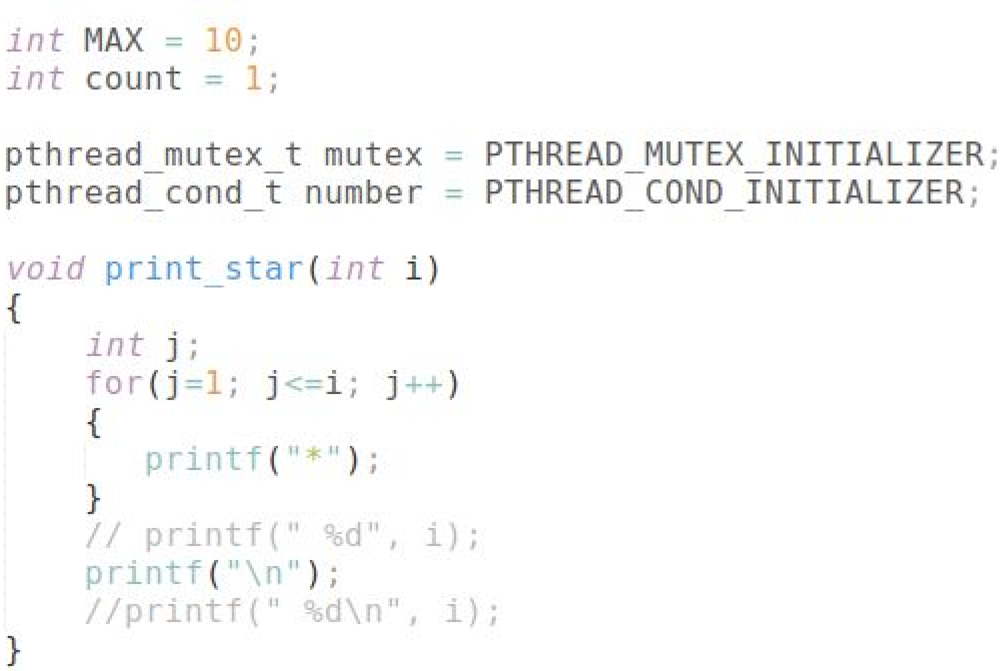


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 1)
        {
            //pthread_cond_wait(&number, &mutex);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
        printf("This is odd thread()\n");
        print_star(count++);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

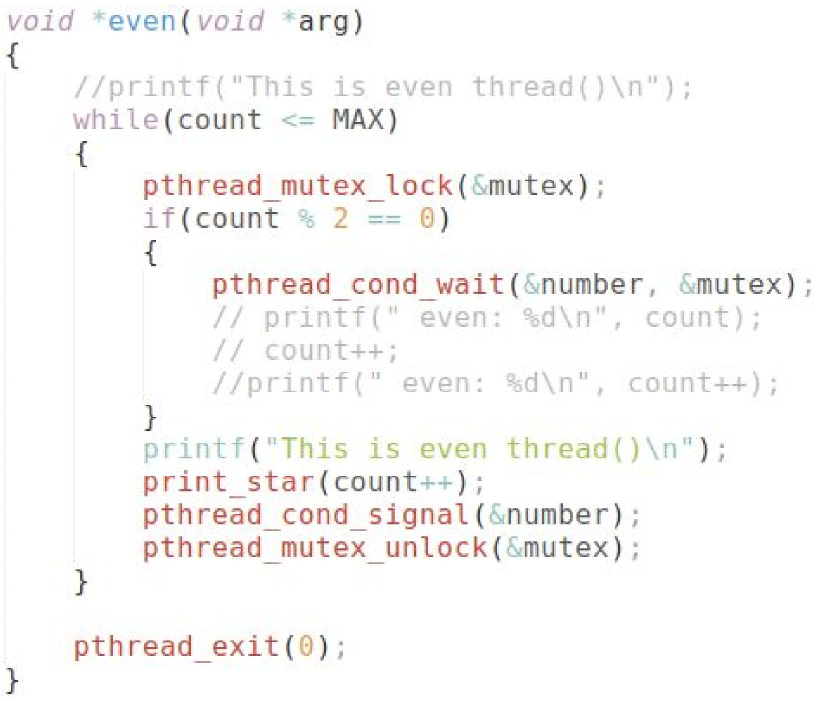


```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&number);

    return  0;
}
```

syc_c2.c - opposite order (1)

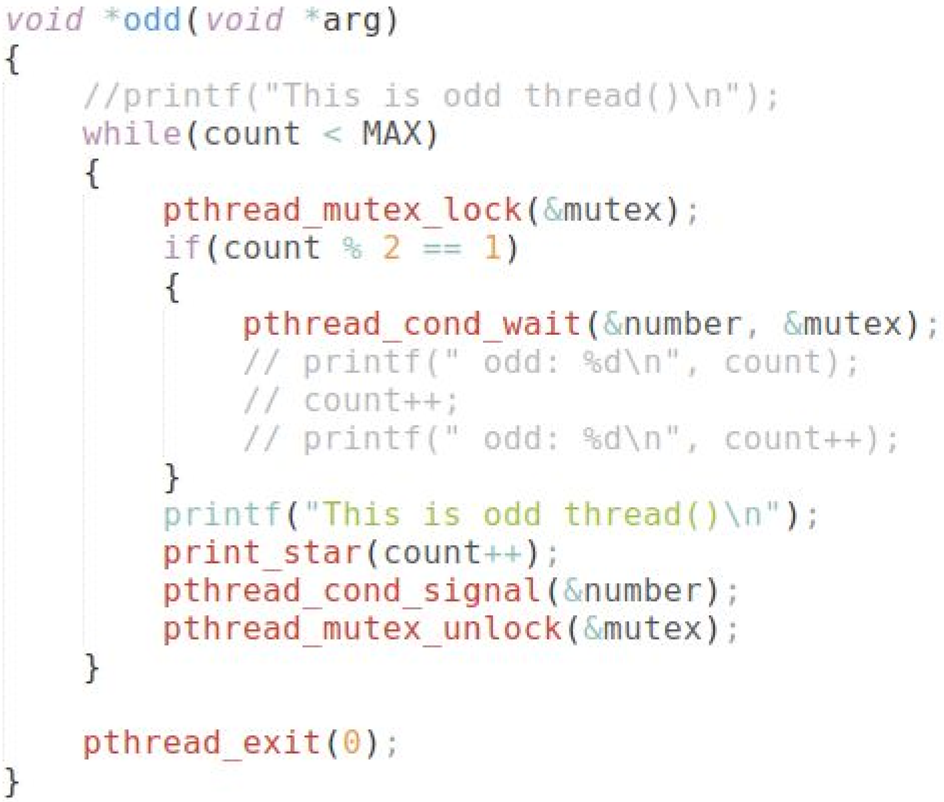


```c
int MAX = 10;
int count = 1;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t number = PTHREAD_COND_INITIALIZER;

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
        printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}
```

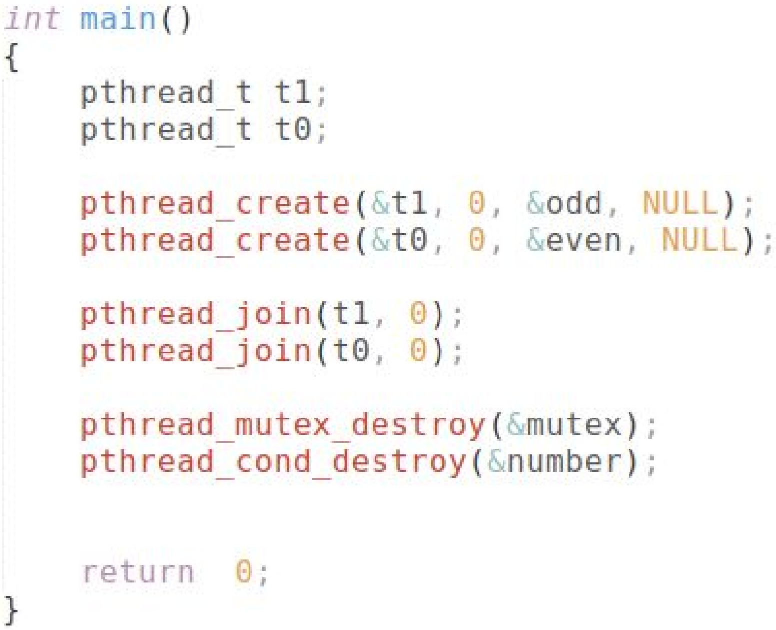


```c
void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 0)
        {
            pthread_cond_wait(&number, &mutex);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
        printf("This is even thread()\n");
        print_star(count++);
        pthread_cond_signal(&number);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

syc_c2.c - opposite order (2)

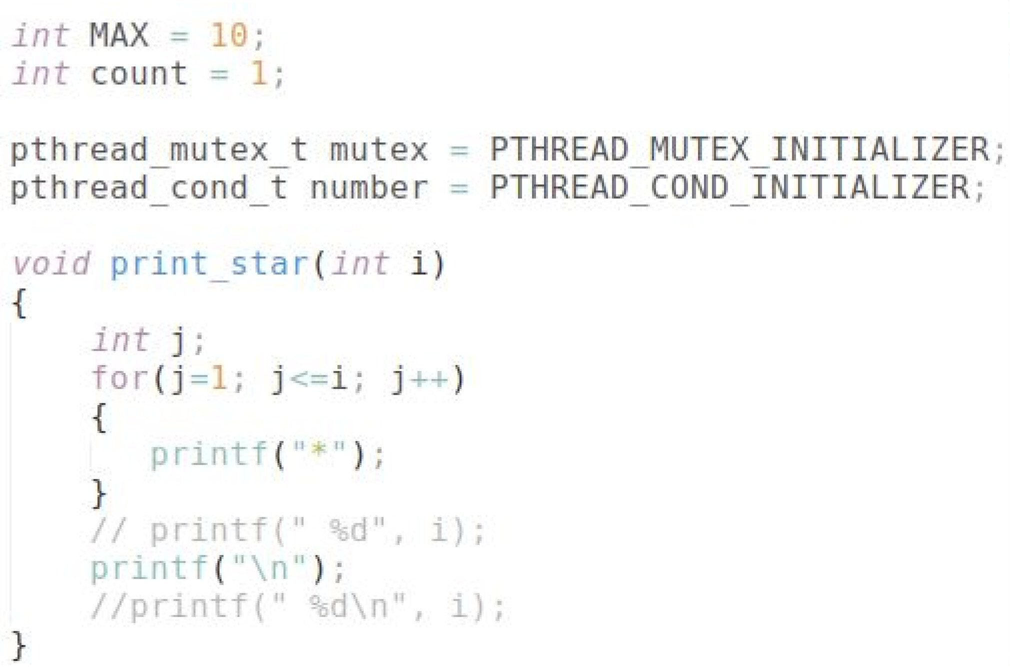


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 1)
        {
            pthread_cond_wait(&number, &mutex);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
        printf("This is odd thread()\n");
        print_star(count++);
        pthread_cond_signal(&number);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

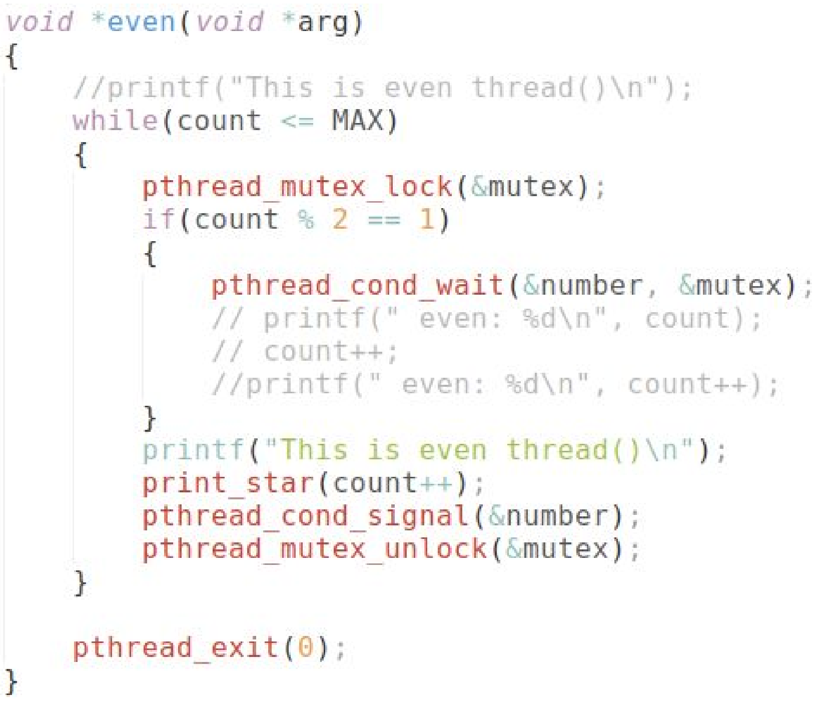


```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&number);


    return  0;
}
```

syc_c3.c - right output (1)

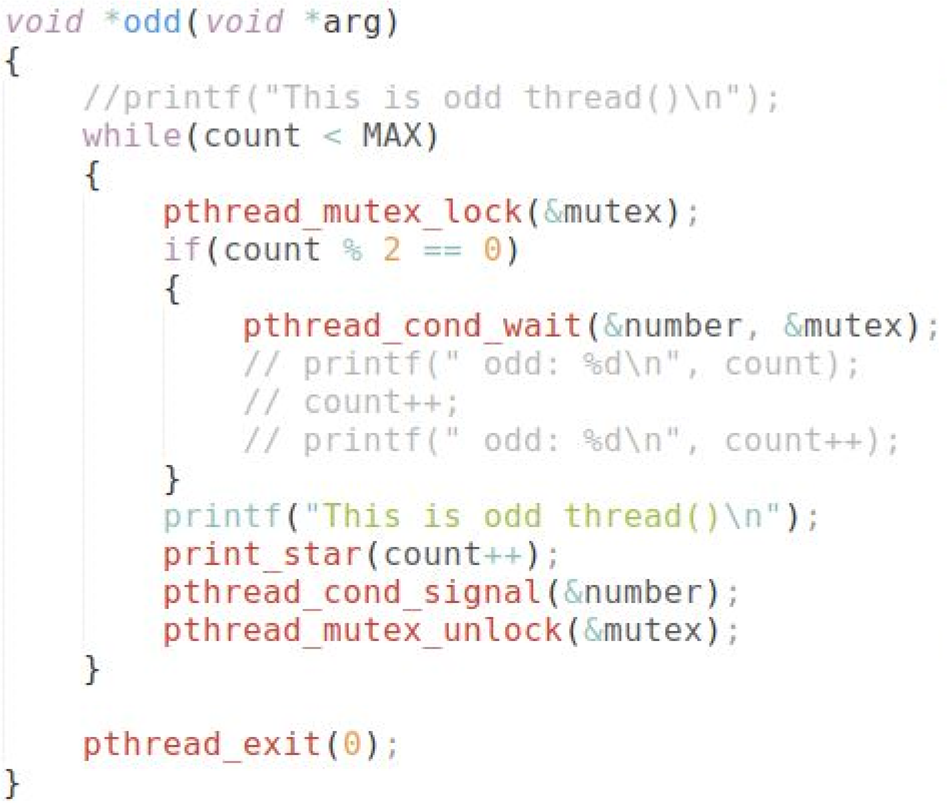


```c
int MAX = 10;
int count = 1;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t number = PTHREAD_COND_INITIALIZER;

void print_star(int i)
{
    int j;
    for(j=1; j<=i; j++)
    {
        printf("*");
    }
    // printf(" %d", i);
    printf("\n");
    //printf(" %d\n", i);
}
```

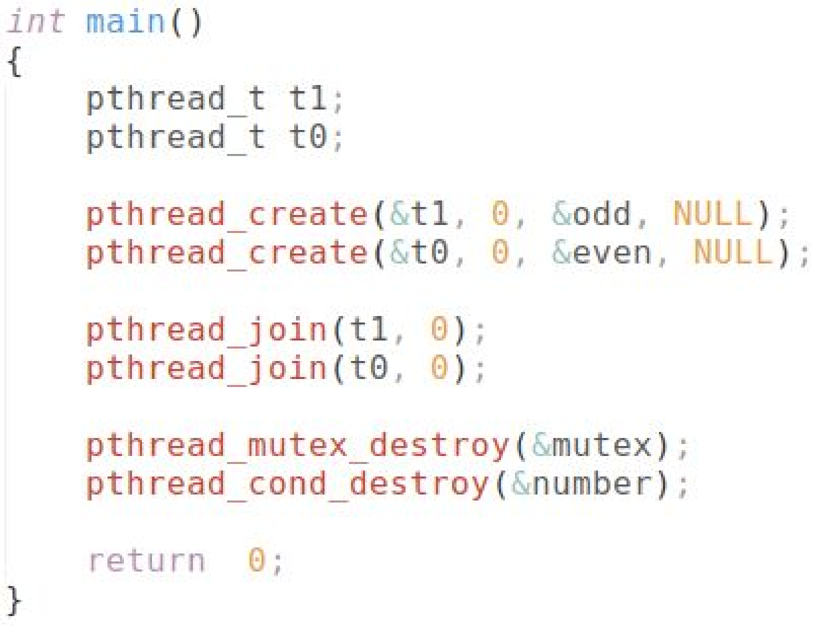


```c
void *even(void *arg)
{
    //printf("This is even thread()\n");
    while(count <= MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 1)
        {
            pthread_cond_wait(&number, &mutex);
            // printf(" even: %d\n", count);
            // count++;
            //printf(" even: %d\n", count++);
        }
        printf("This is even thread()\n");
        print_star(count++);
        pthread_cond_signal(&number);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```

syc_c3.c - right output (2)

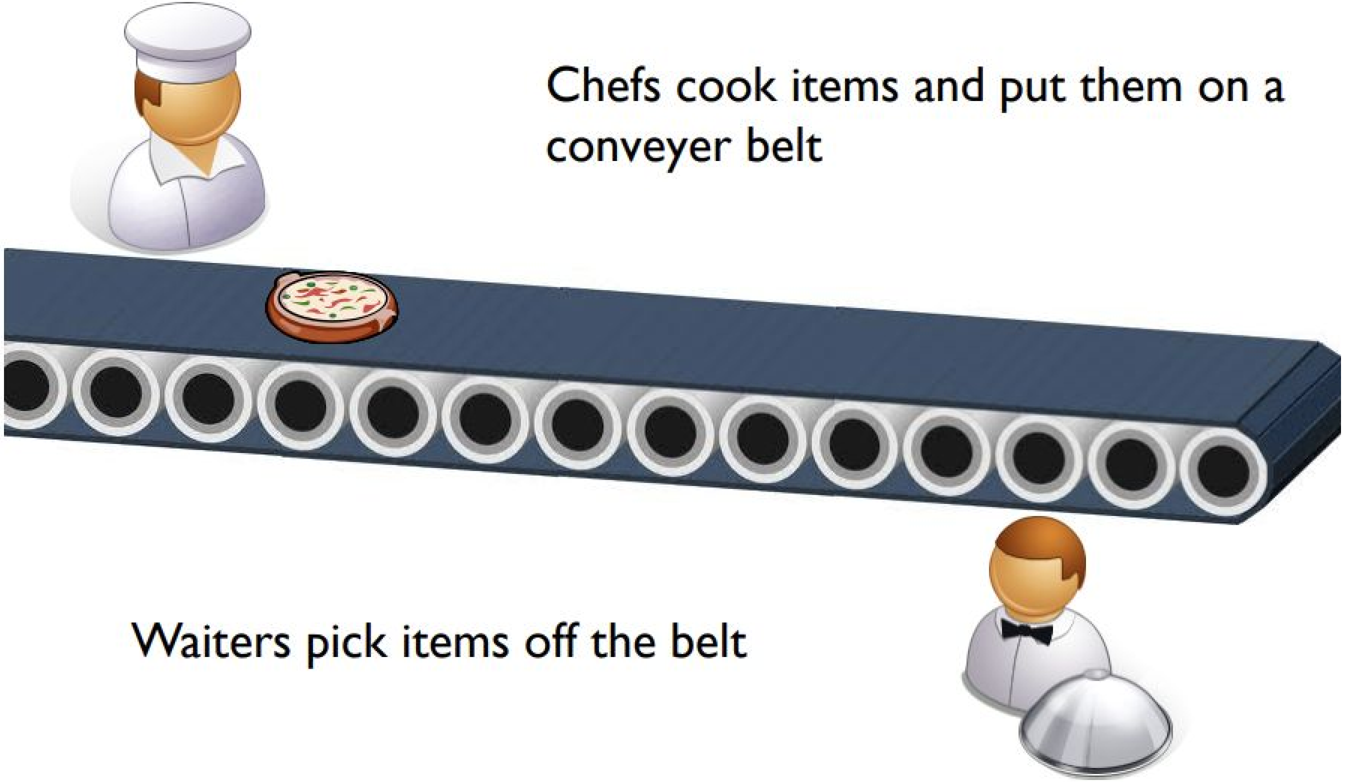


```c
void *odd(void *arg)
{
    //printf("This is odd thread()\n");
    while(count < MAX)
    {
        pthread_mutex_lock(&mutex);
        if(count % 2 == 0)
        {
            pthread_cond_wait(&number, &mutex);
            // printf(" odd: %d\n", count);
            // count++;
            // printf(" odd: %d\n", count++);
        }
        printf("This is odd thread()\n");
        print_star(count++);
        pthread_cond_signal(&number);
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(0);
}
```



```c
int main()
{
    pthread_t t1;
    pthread_t t0;

    pthread_create(&t1, 0, &odd, NULL);
    pthread_create(&t0, 0, &even, NULL);

    pthread_join(t1, 0);
    pthread_join(t0, 0);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&number);

    return  0;
}
```

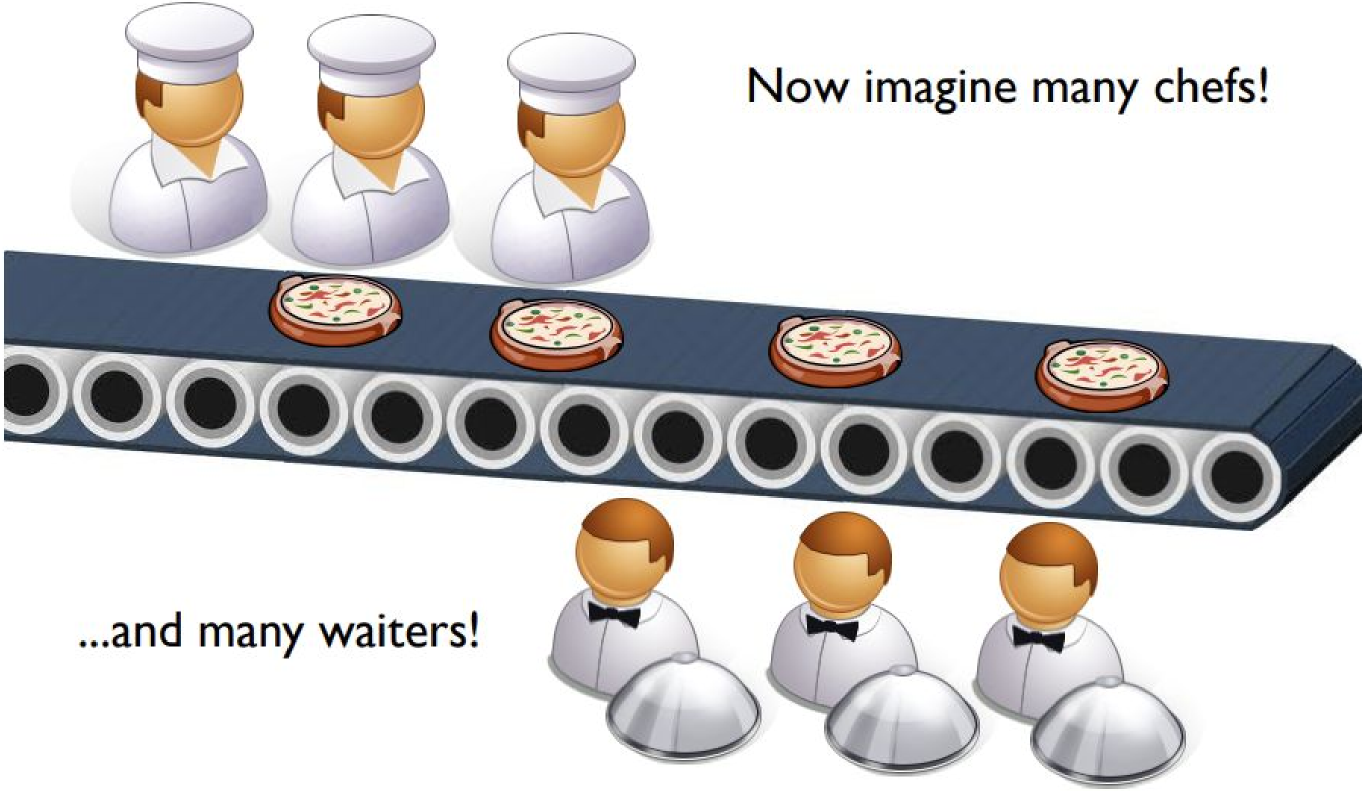
Producer-consumer Problem



Chefs cook items and put them on a conveyer belt

Waiters pick items off the belt

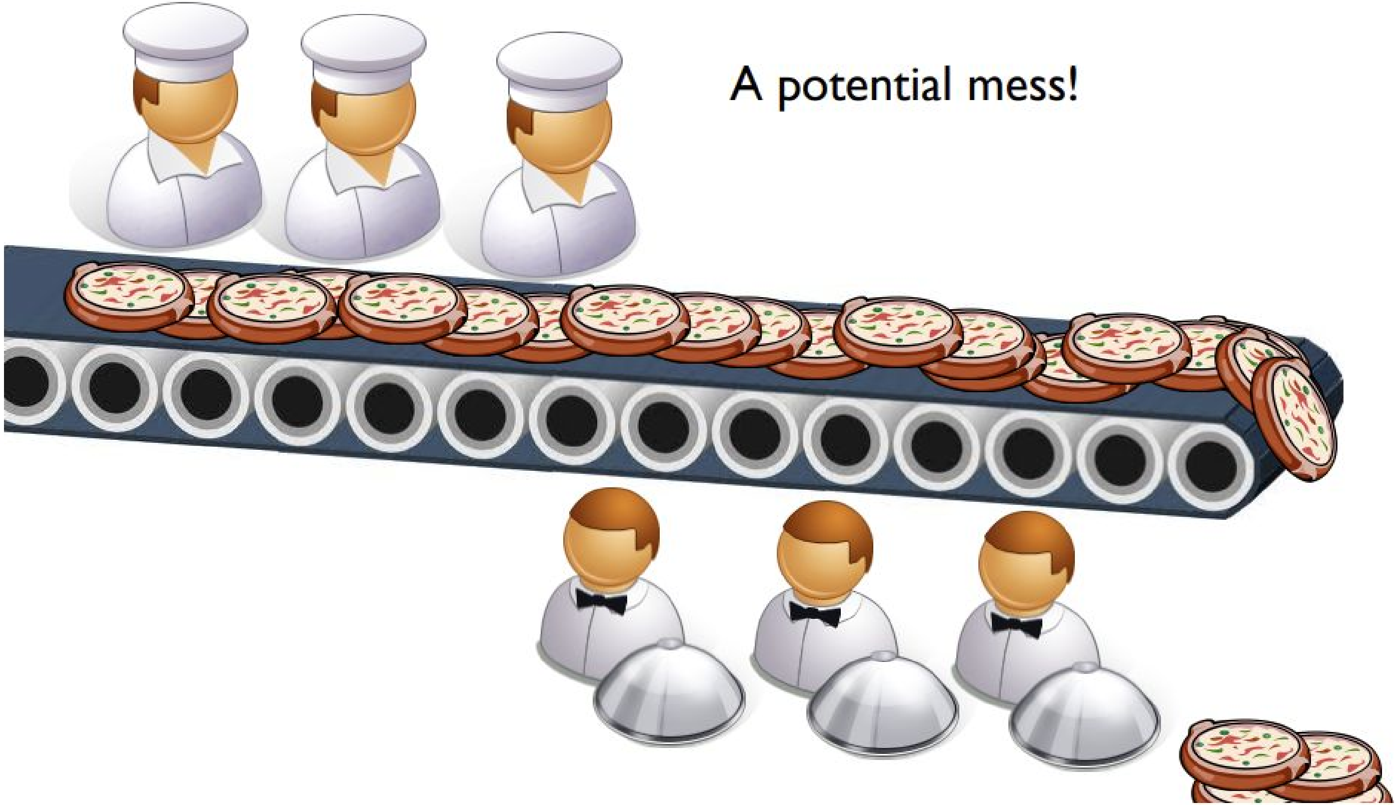
Producer-consumer Problem (Cont.)



Now imagine many chefs!

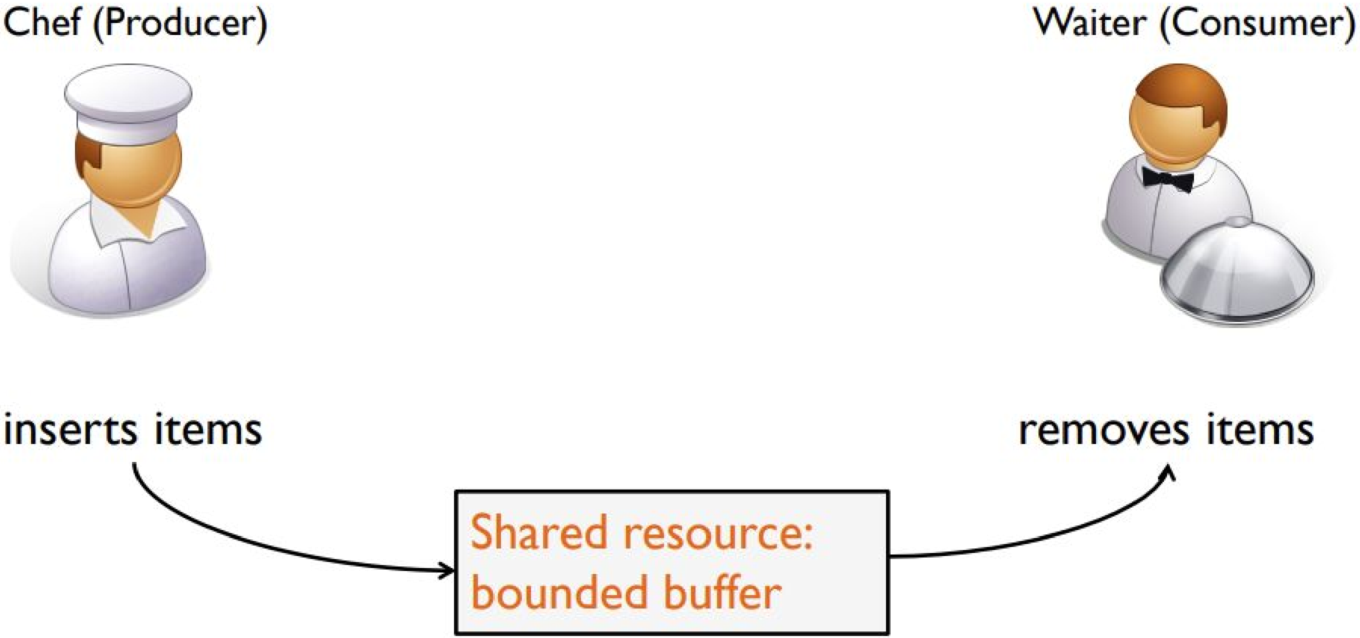
...and many waiters!

Producer-consumer Problem (Cont.)



A potential mess!

Producer-consumer Problem (Cont.)



Chef (Producer)

Waiter (Consumer)

inserts items

removes items

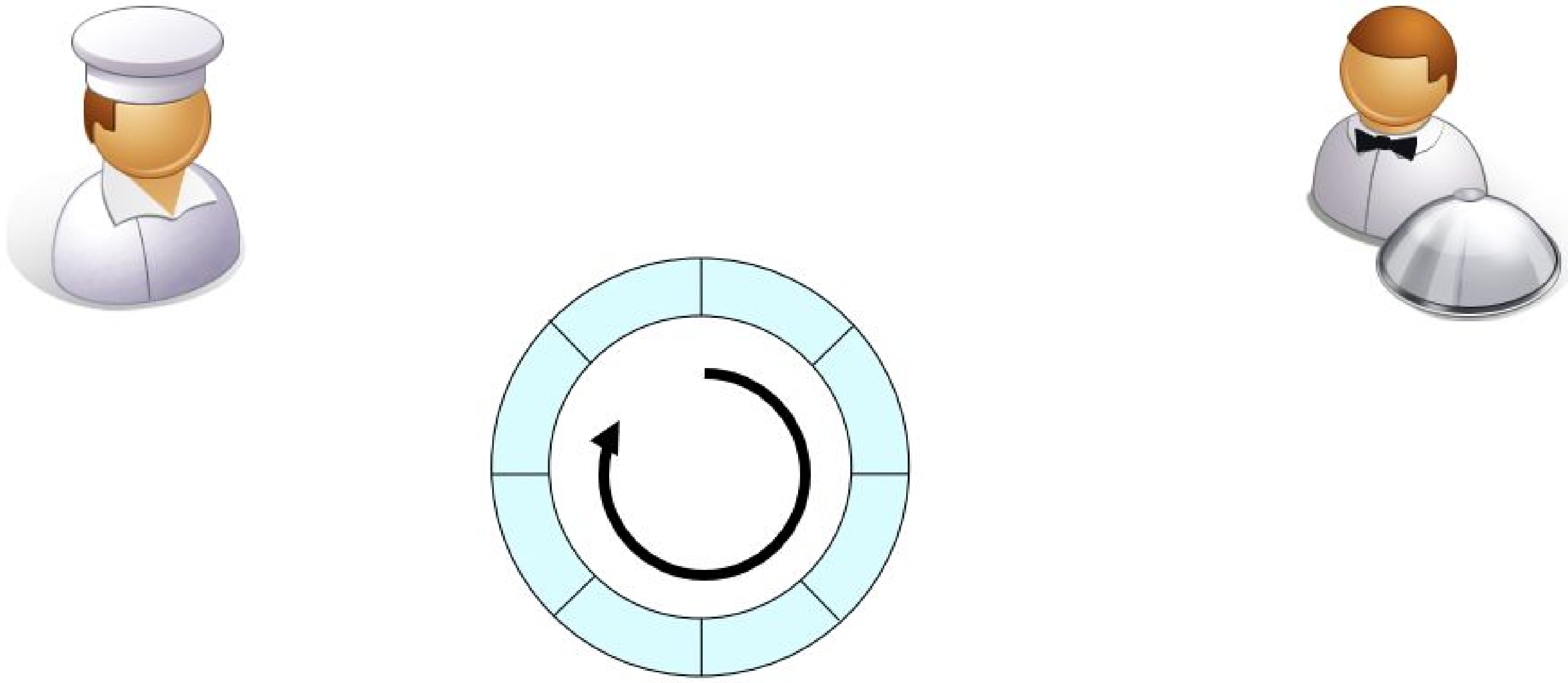
Shared resource: bounded buffer

Efficient implementation:
circular fixed-size buffer

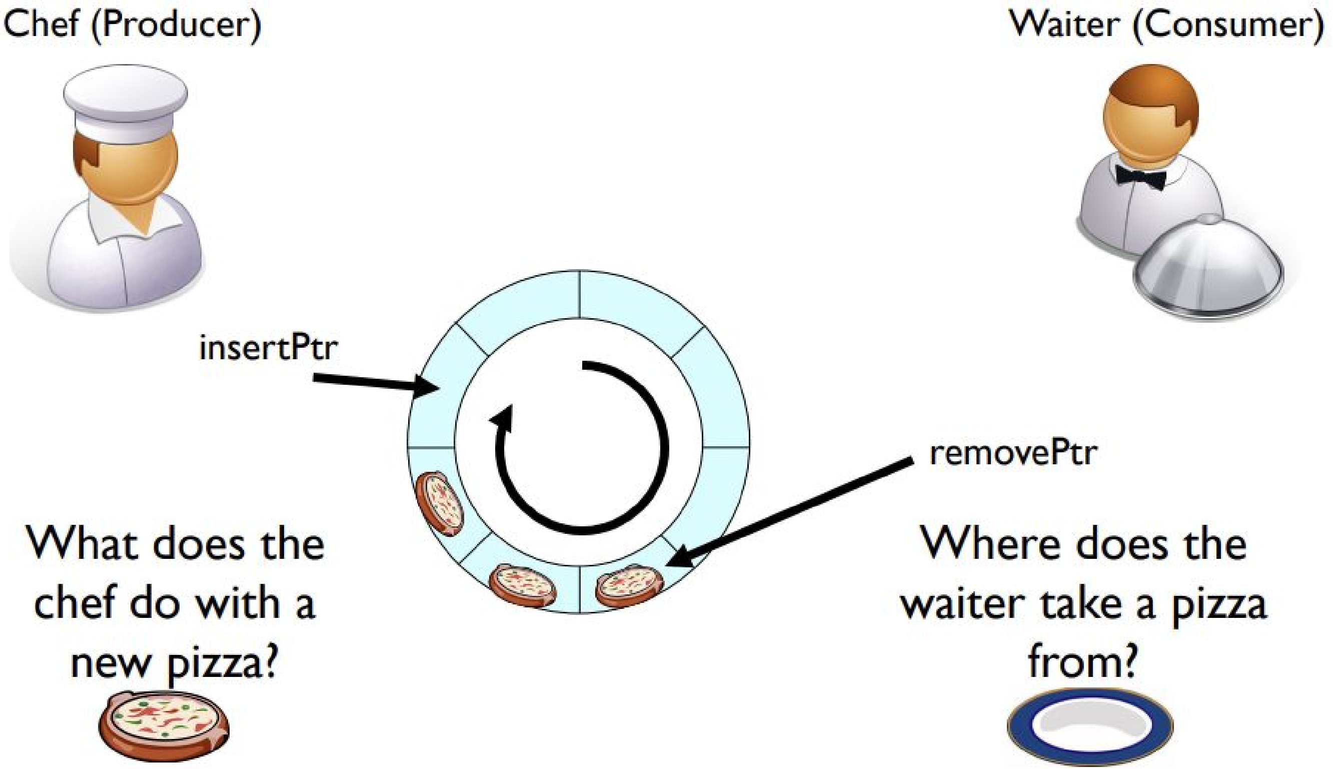
Producer-consumer Problem (Cont.)

Chef (Producer)

Waiter (Consumer)



Producer-consumer Problem (Cont.)



Chef (Producer)

Waiter (Consumer)

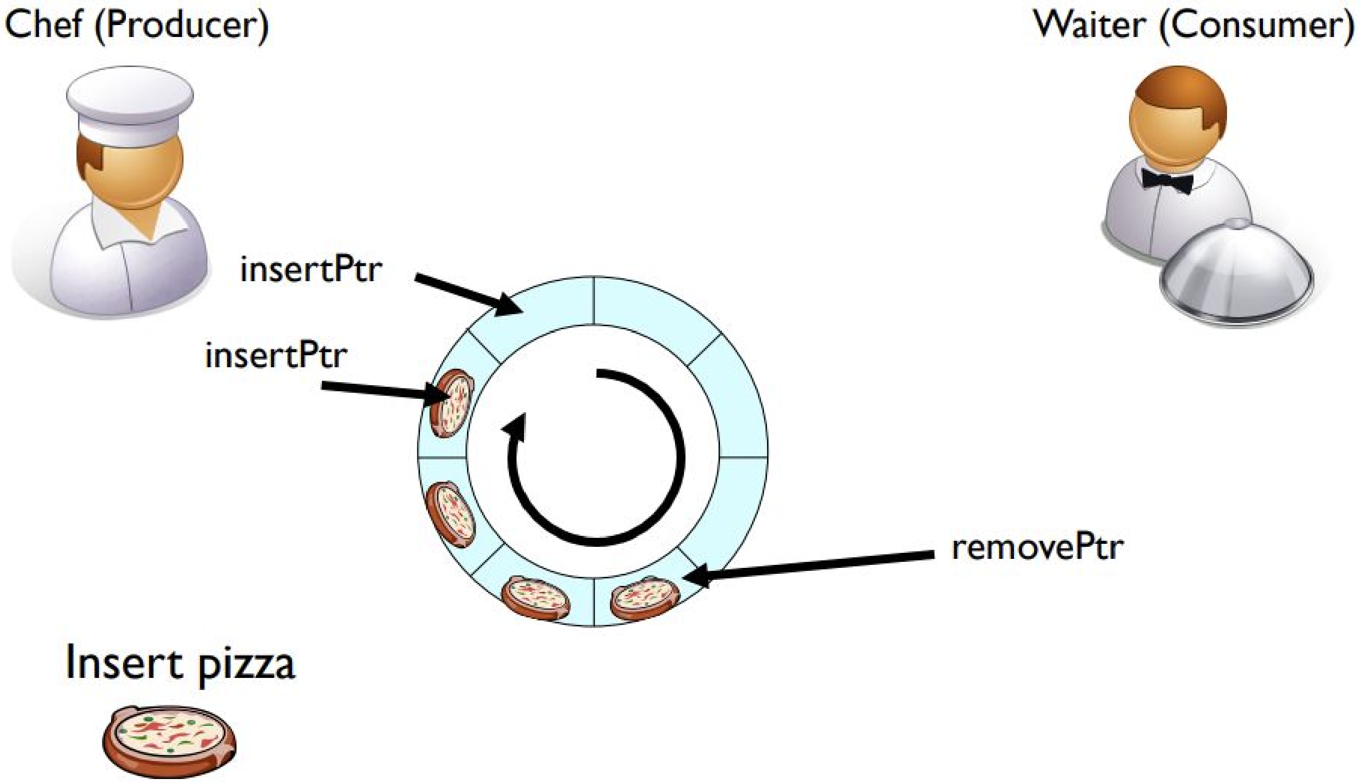
insertPtr

removePtr

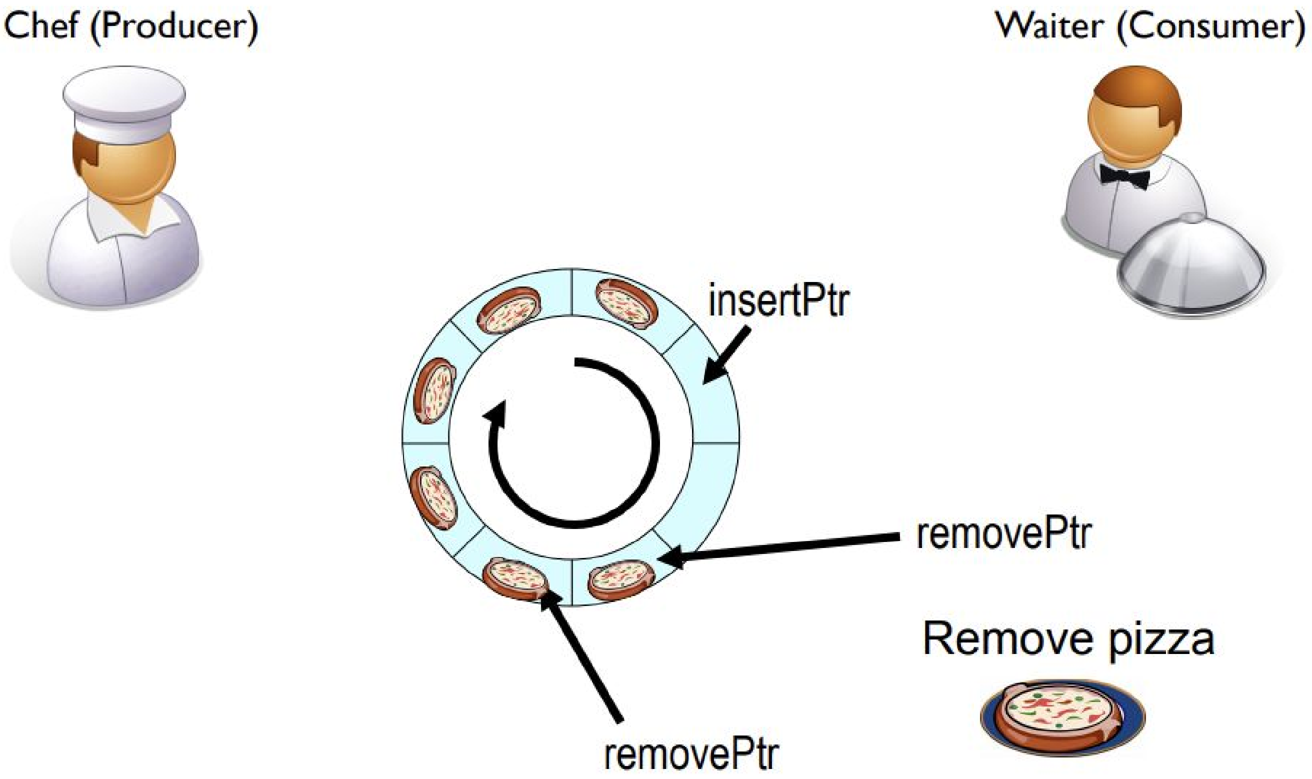
What does the chef do with a new pizza?

Where does the waiter take a pizza from?

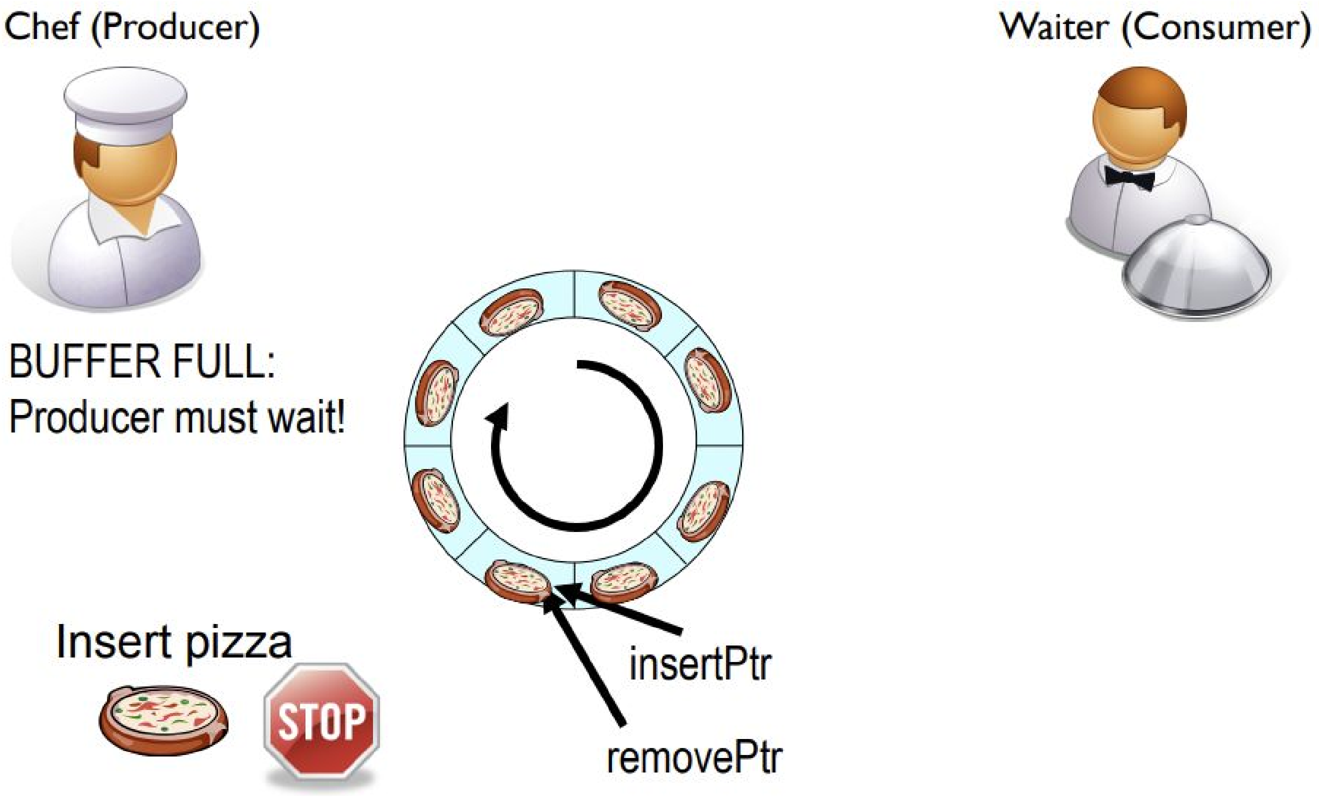
Producer-consumer Problem (Cont.)



Producer-consumer Problem (Cont.)



Producer-consumer Problem (Cont.)



Chef (Producer)

Waiter (Consumer)

BUFFER FULL:
Producer must wait!

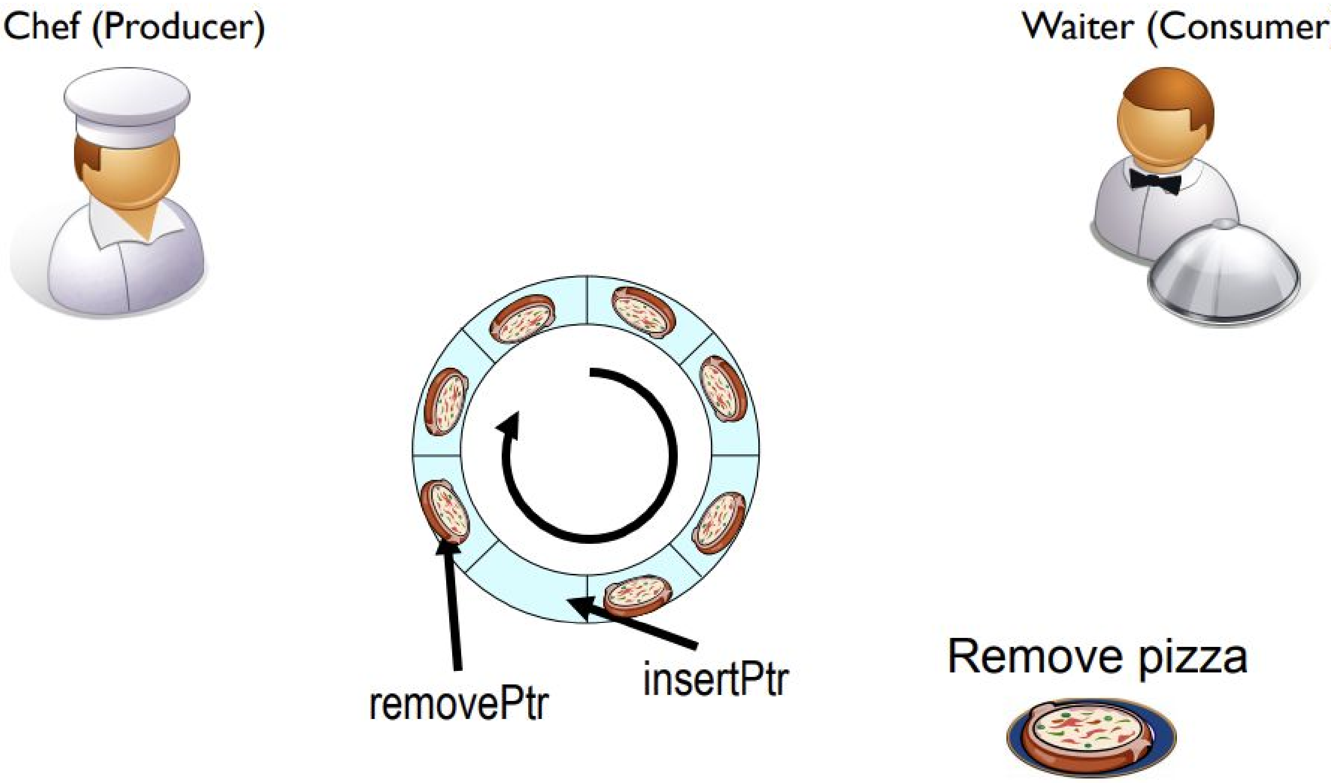
Insert pizza

STOP

insertPtr

removePtr

Producer-consumer Problem (Cont.)



Chef (Producer)

Waiter (Consumer)

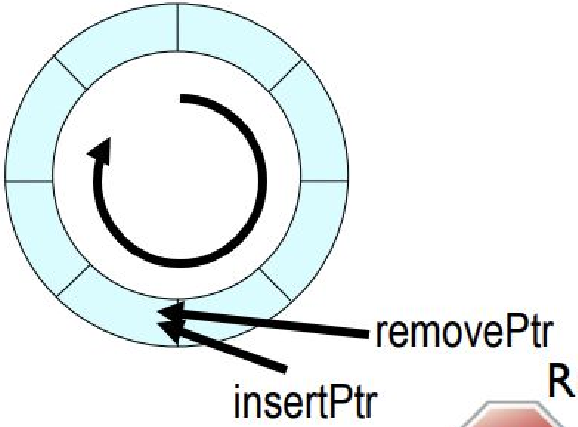
removePtr

insertPtr

Remove pizza

Producer-consumer Problem (Cont.)

Buffer empty:
Consumer must be blocked!



Producer-consumer Problem (Cont.)

Chef (Producer)

Waiter (Consumer)

Wait for empty slot
Insert item
Signal item arrival

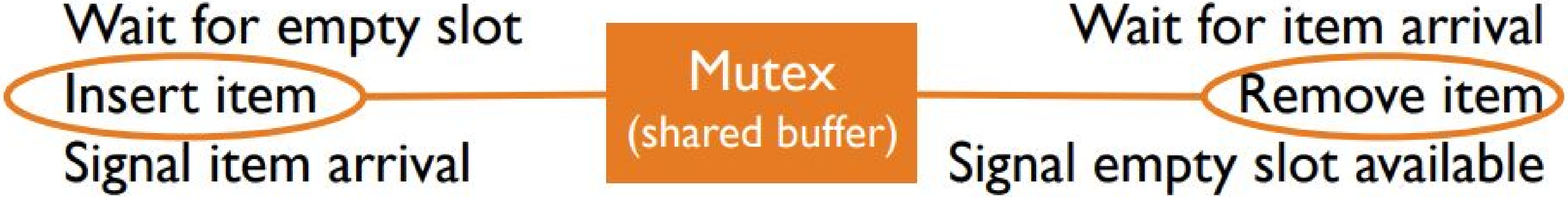
Wait for item arrival
Remove item
Signal empty slot available

What synchronization do we need?

Producer-consumer Problem (Cont.)

Chef (Producer)

Waiter (Consumer)



Wait for empty slot

Insert item

Signal item arrival

Mutex
(shared buffer)

Wait for item arrival

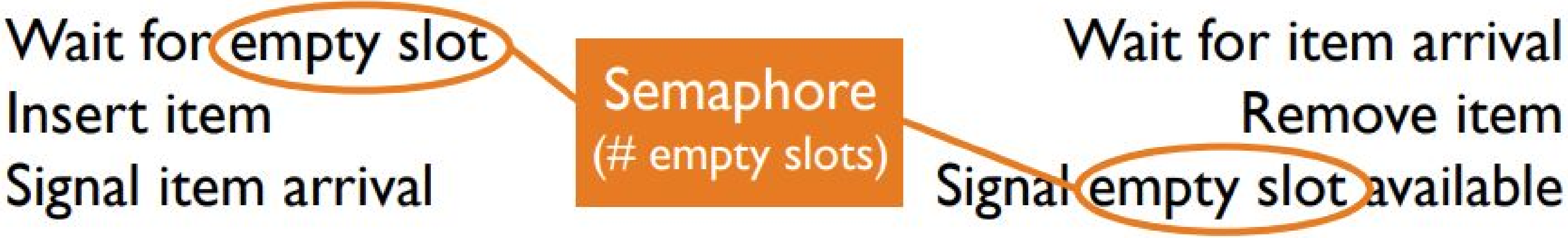
Remove item

Signal empty slot available

Producer-consumer Problem (Cont.)

Chef (Producer)

Waiter (Consumer)



Wait for empty slot

Insert item

Signal item arrival

Semaphore

(# empty slots)

Wait for item arrival

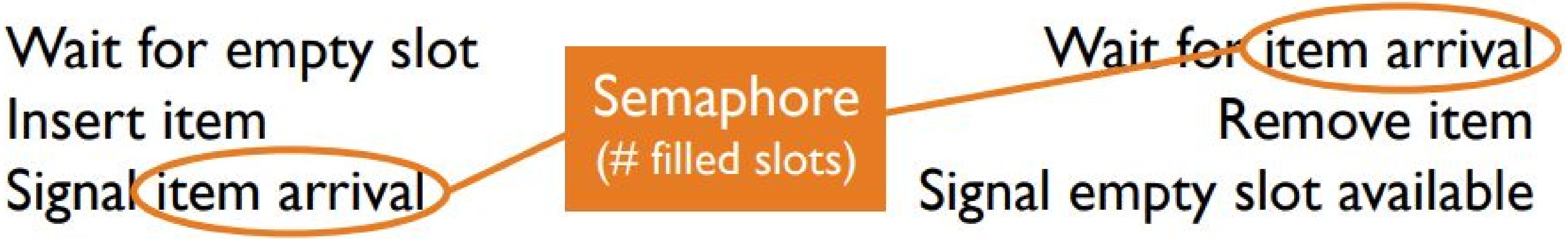
Remove item

Signal empty slot available

Producer-consumer Problem (Cont.)

Chef (Producer)

Waiter (Consumer)



Wait for empty slot

Insert item

Signal item arrival

Semaphore

(# filled slots)

Wait for item arrival

Remove item

Signal empty slot available

Producer-consumer Problem (Cont.)

Counting semaphore – check and decrement the number of free slots

```
sem_wait(&slots);

mutex_lock(&mutex);

buffer[ insertPtr ] =
data;

insertPtr = (insertPtr +
1) % N;

mutex_unlock(&mutex);

sem_post(&items);
```

Block if there are no free slots

Done – increment the number of available items

Counting semaphore – check and decrement the number of available items

```
sem_wait(&items);

mutex_lock(&mutex);

result =
buffer[removePtr];

removePtr = (removePtr
+1) % N;

mutex_unlock(&mutex);

sem_post(&slots);
```

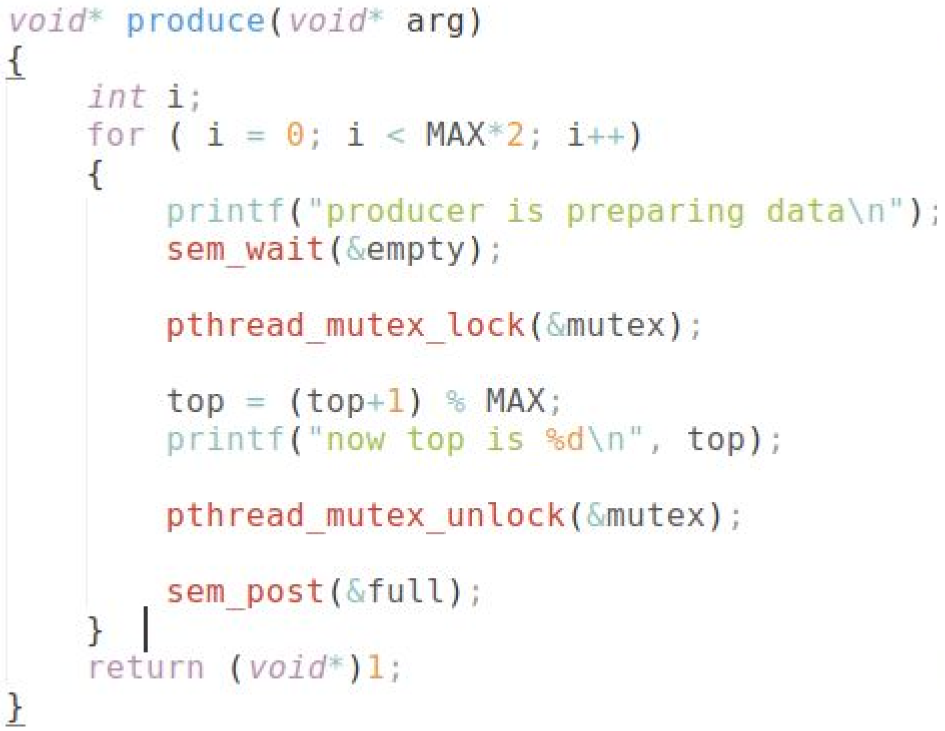
Block if there are no items to take

Done – increment the number of free slots

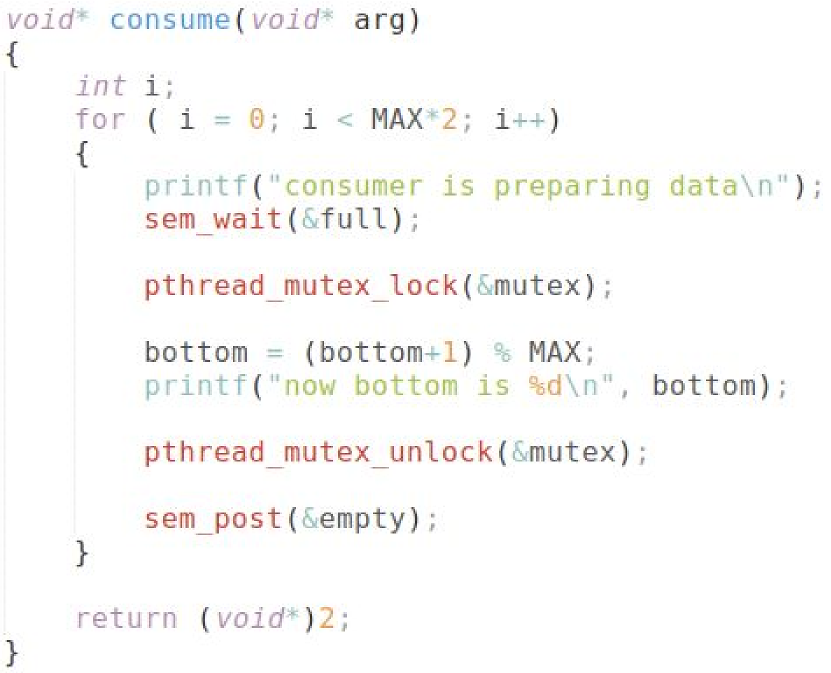
PCP - Two ways to solve

- By *Semaphore* with *Mutex*
- By *Condition Variable* with *Mutex*

PCP - By Semaphore with Mutex

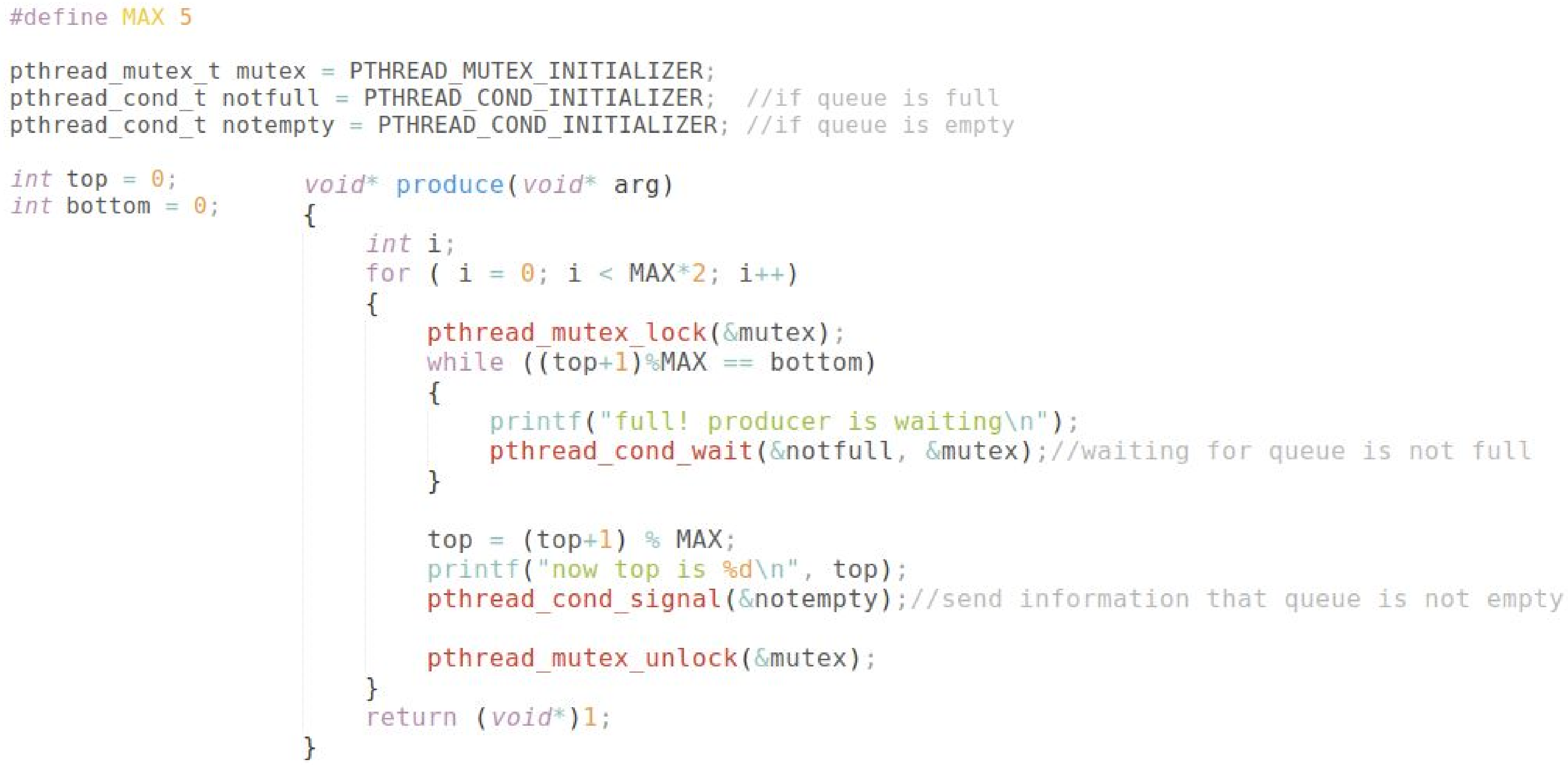


```
void* produce(void* arg)
{
    int i;
    for ( i = 0; i < MAX*2; i++)
    {
        printf("producer is preparing data\n");
        sem_wait(&empty);

        pthread_mutex_lock(&mutex);

        top = (top+1) % MAX;
        printf("now top is %d\n", top);

        pthread_mutex_unlock(&mutex);

        sem_post(&full);
    }
    return (void*)1;
}
```



```
void* consume(void* arg)
{
    int i;
    for ( i = 0; i < MAX*2; i++)
    {
        printf("consumer is preparing data\n");
        sem_wait(&full);

        pthread_mutex_lock(&mutex);

        bottom = (bottom+1) % MAX;
        printf("now bottom is %d\n", bottom);

        pthread_mutex_unlock(&mutex);

        sem_post(&empty);
    }

    return (void*)2;
}
```

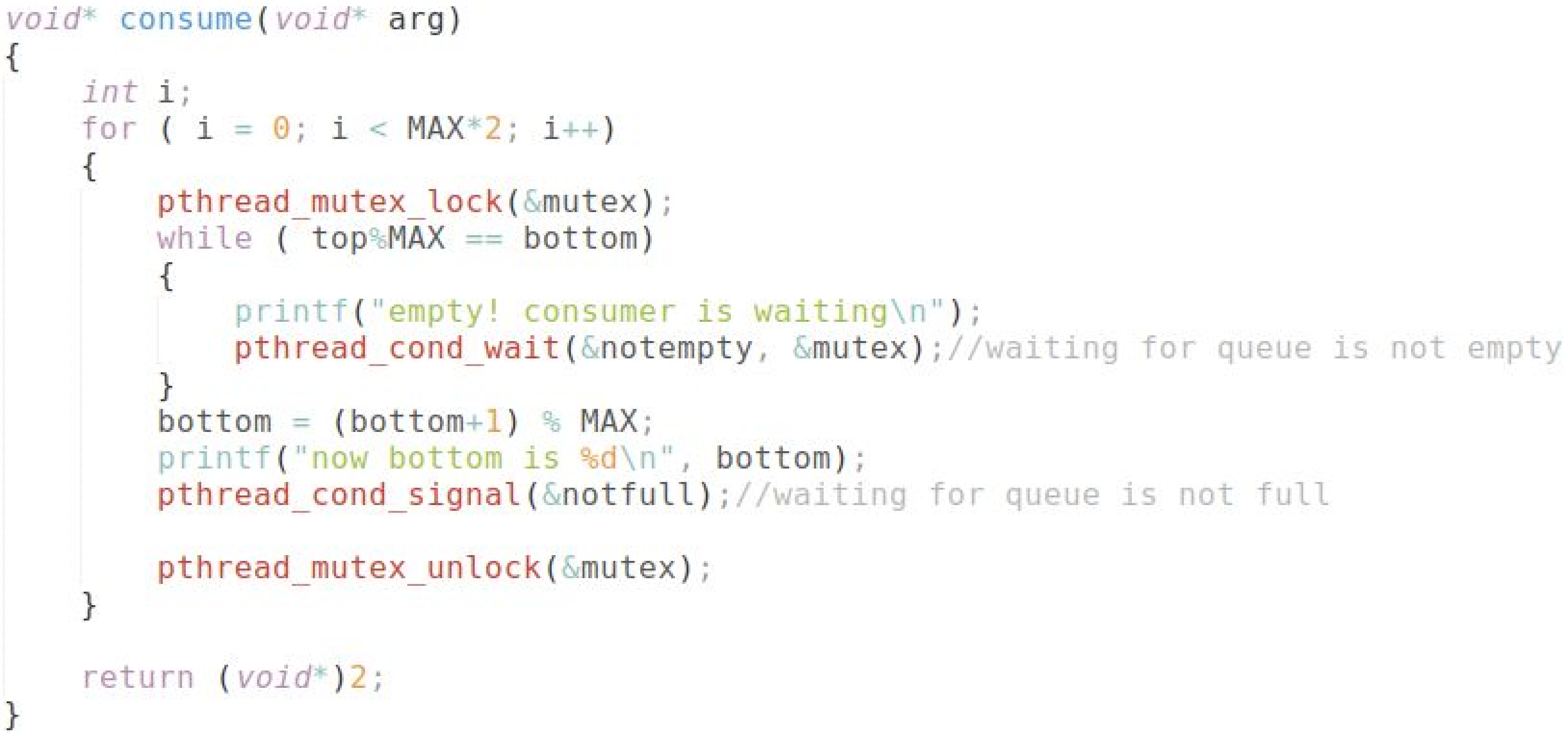
PCP - By Condition Variable with Mutex 1



```c
#define MAX 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t notfull = PTHREAD_COND_INITIALIZER;  //if queue is full
pthread_cond_t notempty = PTHREAD_COND_INITIALIZER; //if queue is empty

int top = 0;
int bottom = 0;
```

```c
void* produce(void* arg)
{
    int i;
    for ( i = 0; i < MAX*2; i++)
    {
        pthread_mutex_lock(&mutex);
        while ((top+1)%MAX == bottom)
        {
            printf("full! producer is waiting\n");
            pthread_cond_wait(&notfull, &mutex);//waiting for queue is not full
        }

        top = (top+1) % MAX;
        printf("now top is %d\n", top);
        pthread_cond_signal(&notempty);//send information that queue is not empty

        pthread_mutex_unlock(&mutex);
    }
    return (void*)1;
}
```

PCP - By Condition Variable with Mutex 2



```c
void* consume(void* arg)
{
    int i;
    for ( i = 0; i < MAX*2; i++)
    {
        pthread_mutex_lock(&mutex);
        while ( top%MAX == bottom)
        {
            printf("empty! consumer is waiting\n");
            pthread_cond_wait(&notempty, &mutex);//waiting for queue is not empty
        }
        bottom = (bottom+1) % MAX;
        printf("now bottom is %d\n", bottom);
        pthread_cond_signal(&notfull);//waiting for queue is not full

        pthread_mutex_unlock(&mutex);
    }

    return (void*)2;
}
```