CS3.301 Operating Systems and Networks

Concurrency - Condition Variables

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

SERC
Software Engineering Research Centre

Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in three easy pieces by Remzi et al.

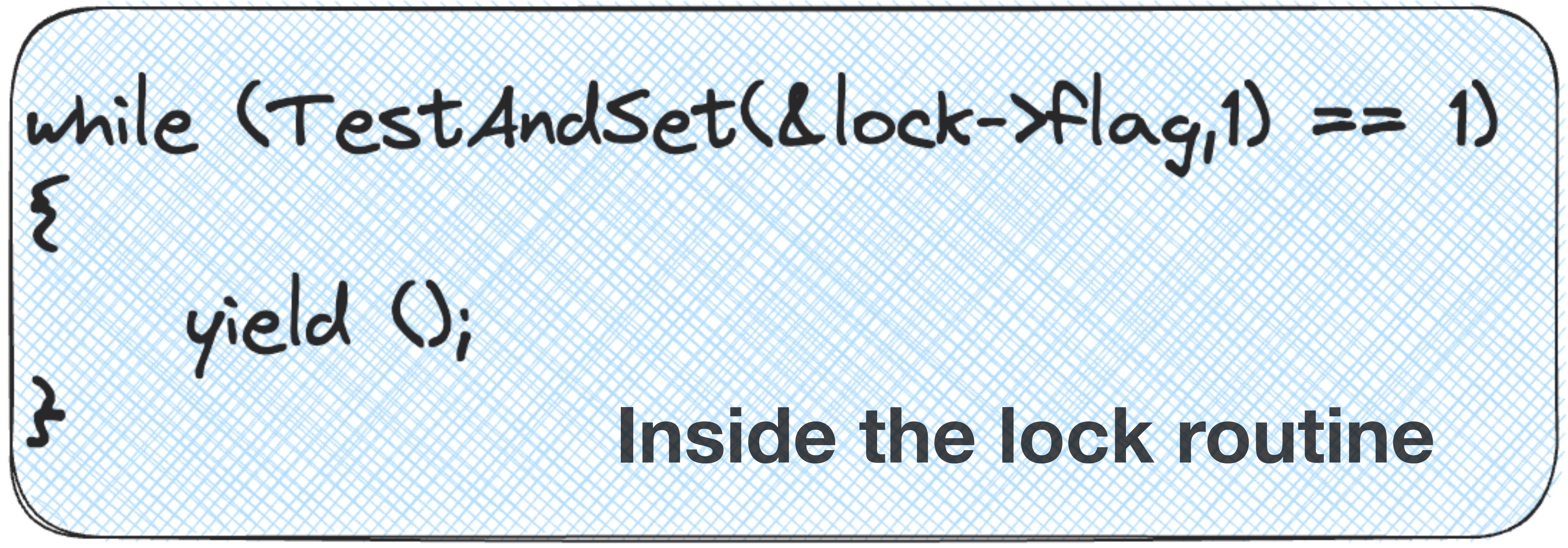
How good is the spin based locks?

- Simple hardware based locks are simple to implement and powerful
- They are also quite inefficient especially when it comes to performance
  - Consider that there are two threads and one thread has the lock
  - When thread has lock, it may get interrupted, the other thread spins for a time slice, waste CPU cycle
  - Think about N threads, N-1 threads might waste CPU cycles in spinning (especially if round robin)
- **Can we come up with something better instead of wasting cycles with spinning?**

OS support can help

The yield call

- If the thread is aware that it is going to spin - Why not give up the CPU to some other thread?
  - Simple OS primitive system call: **yield()**
  - Moves the thread from running state to ready state => another thread can run
  - Does this solution work efficiently?
    - What if there are 100 threads?
    - Still costly! - 99 threads runs to yield
    - Possibility of infinite yields as well - **Starvation! - Why?**



```
while (TestAndSet(&lock->flag,1) == 1)
{
      yield ();
}
```

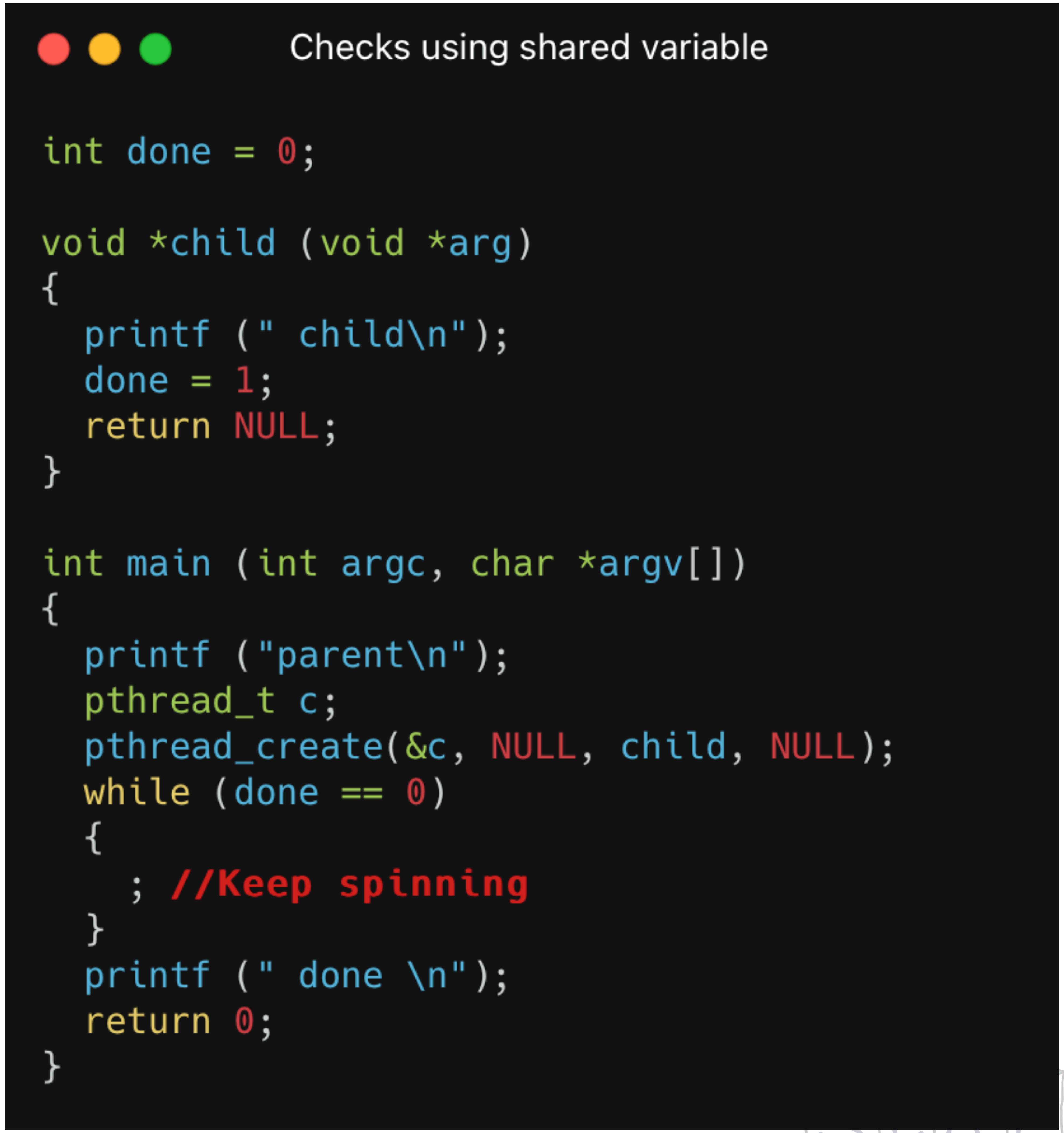
Inside the lock routine

Can we make thread sleep rather than spinning?

- Why don’t we make use of some queue based structures?
- Keep a queue to track which thread needs access to CS
- Syscalls by Solaris: **park()** and **unpark()**
  - park(): puts a thread to sleep
  - unpark(tid): wakeup that particular thread
- If a thread wants to acquire a lock
  - Check if others have the lock, if yes put thread to sleep
  - If lock is free, wake up the thread and give the lock

Locks do help in access to CS! But more challenges

- Locks ensures that thread can get access to CS
    - With help of HW and SW mechanisms efficient locks can be built
- But, thread while executing may want to check for some conditions
    - A parent thread may want to check if the child thread has completed before proceeding
        - Remember join() operation? - How to make it work?
        - **Why don’t we use shared variable?**



Checks using shared variable

```
int done = 0;

void *child (void *arg)
{
  printf (" child\n");
  done = 1;
  return NULL;
}

int main (int argc, char *argv[])
{
  printf ("parent\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  while (done == 0)
  {
    ; //Keep spinning
  }
  printf (" done \n");
  return 0;
}
```

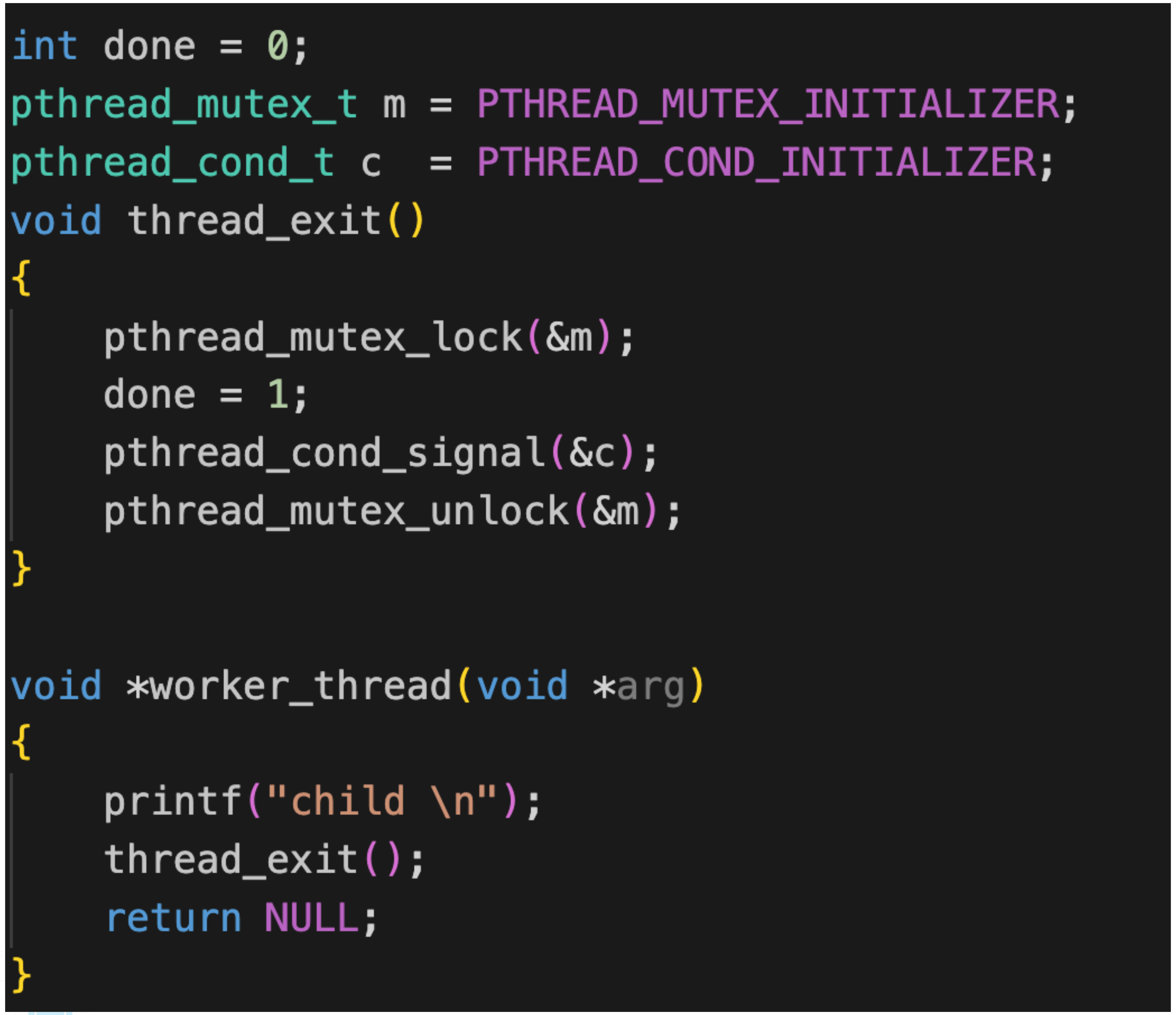
Condition Variables

- **Condition variable:** Explicit queues that the threads can put themselves on when a state of condition is not as desired
    - **Eg:** lock is not available (flag might be 0)
- When condition is met, thread can be woken up to continue

```
pthread_cond_t c;
```

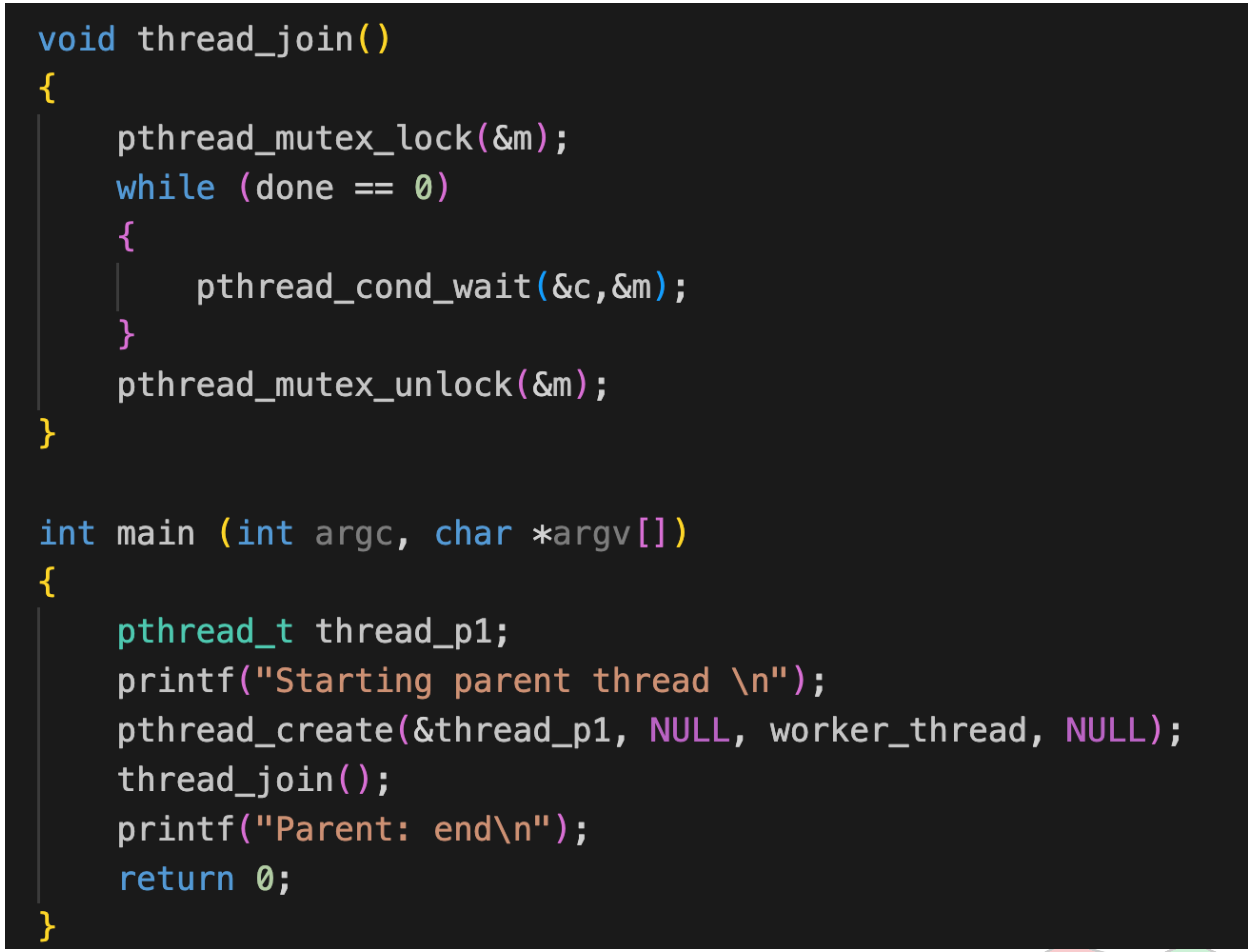
- c is a condition variable with two operations - wait() and signal()
- **wait():** when thread wants to put itself to sleep
- **signal():** there is some change and thread wants to wake up thread waiting on condition

Condition Variables in Action



```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
void thread_exit()
{
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void *worker_thread(void *arg)
{
    printf("child \n");
    thread_exit();
    return NULL;
}
```



```
void thread_join()
{
    pthread_mutex_lock(&m);
    while (done == 0)
    {
        pthread_cond_wait(&c,&m);
    }
    pthread_mutex_unlock(&m);
}

int main (int argc, char *argv[])
{
    pthread_t thread_p1;
    printf("Starting parent thread \n");
    pthread_create(&thread_p1, NULL, worker_thread, NULL);
    thread_join();
    printf("Parent: end\n");
    return 0;
}
```

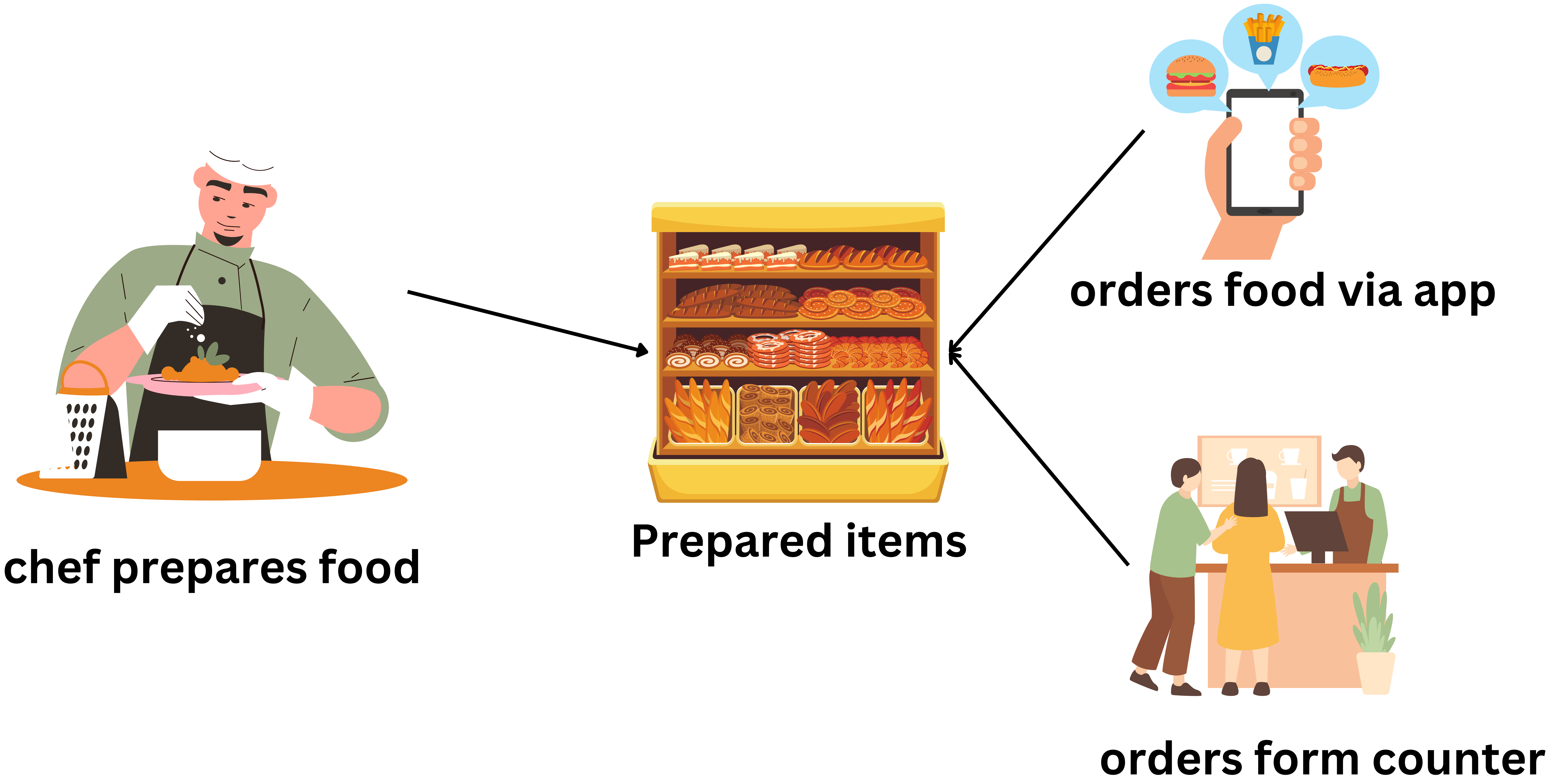
Two cases to consider as it works

- **Parent creates the Child and continues running**
    - Goes into the join call
    - Checks the state variable since child is not done, puts itself to sleep
    - Child runs and invokes exit -> updates state variable and wakes up parent thread
    - Parent will run returning from wait and prints done
- **Child runs immidiately upon creation**
    - Sets done to 1, wakes up sleeping thread (none available) so returns
    - Parent runs join, the done variable is 1 so returns
- **Do we need while loop for checking state and do we need locks?**

State variable and Locks

- **What if we don't have the state variable done**
  - Exit and join functions simply calls wait and join
  - What if child runs first and calls exit, child will signal but no parent thread
  - When parent runs, it will simply wait and never come out of it
- **What if there are no locks around statements in exit and join?**
  - Parent calls join, checks that done is 0, sleeps
  - Just before sleep call, interrupt, child runs and sets done to 1 and signals
  - No thread is waiting, parent runs goes into sleep and forever sleeps - **Race condition**

An Analogy



chef prepares food

Prepared items

orders food via app

orders form counter

One cannot get food items that are not yet ready!

The Producer/Consumer Problem

AKA Bounded Buffer Problem

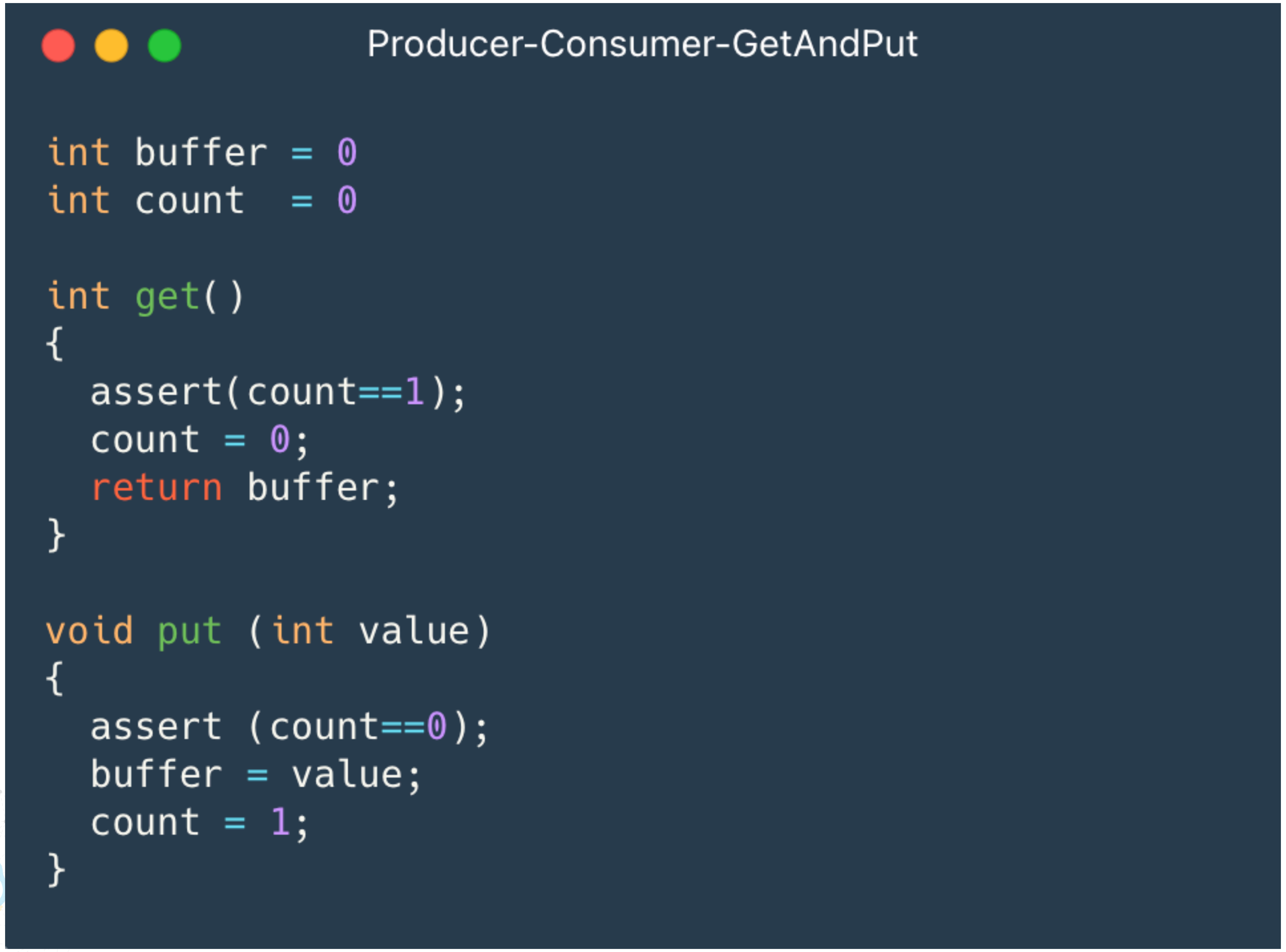
- Think about web servers
  - **Producer:** Produces HTTP requests into a queue
  - **Consumer:** The threads that process the HTTP requests from the queue
  - **Bounded buffer:** The work queue
- Piped calls in unix: grep linux os.txt | wc -l
  - **Producer:** grep gets text that contains “linux” from os.txt and puts them to standard output
  - **Consumer:** Shell redirects them to pipe call, where wc as another process counts and prints the number of lines
  - **Buffer:** Shared resource

Wait There is a challenge

- Bounded buffer is a shared resource
- Producer puts data to empty buffer
- Consumer can only consume from full buffer
- We need synchronisation mechanisms to access it
    - Else it may result in **race conditions**
- How to solve the problem?
- What kind of synchronisation mechanisms can be developed?

Lets start simple

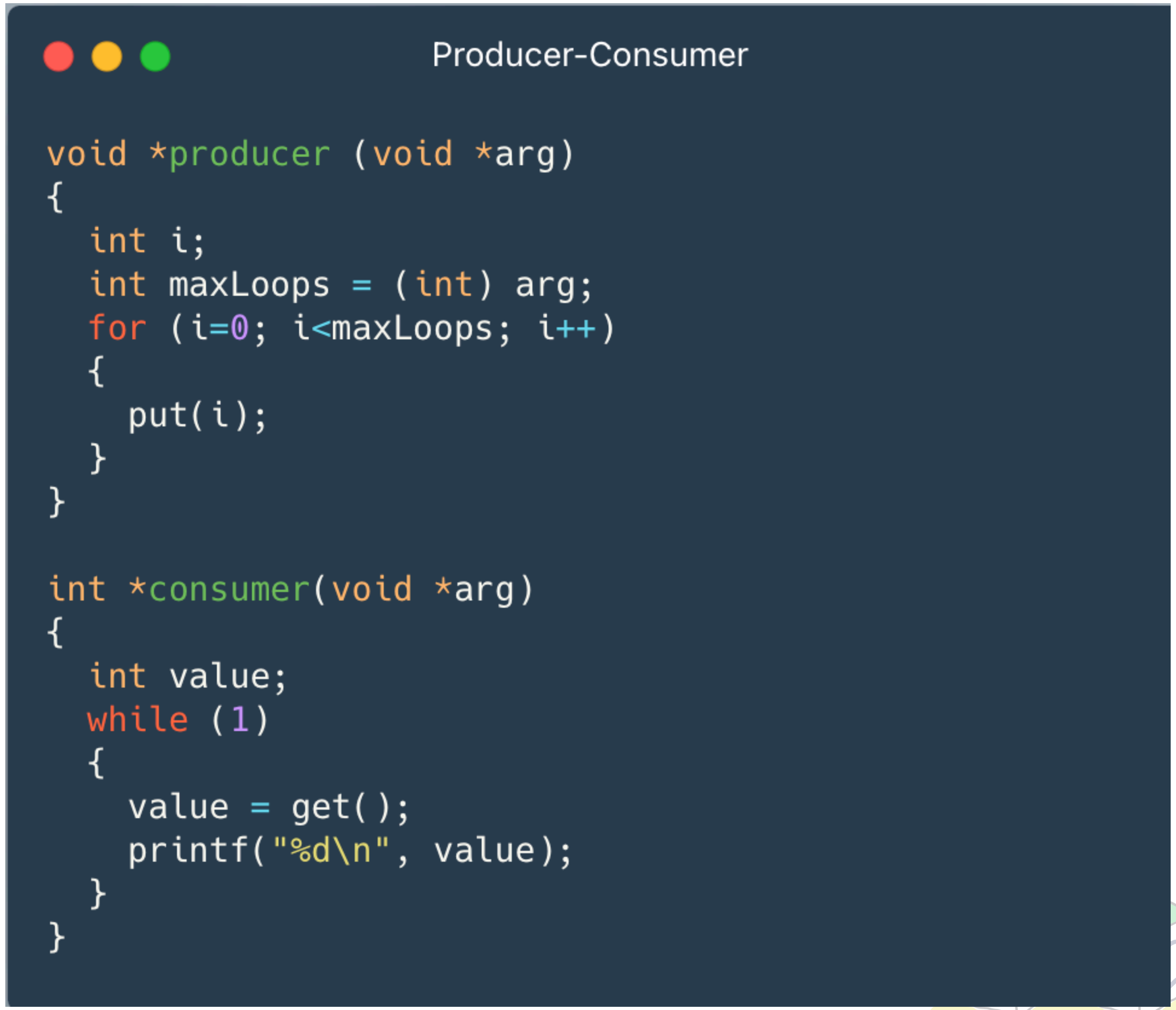
- Consider buffer can hold only one item, a single integer - **How to solve?**



```
int buffer = 0
int count  = 0

int get()
{
  assert(count==1);
  count = 0;
  return buffer;
}

void put (int value)
{
  assert (count==0);
  buffer = value;
  count = 1;
}
```

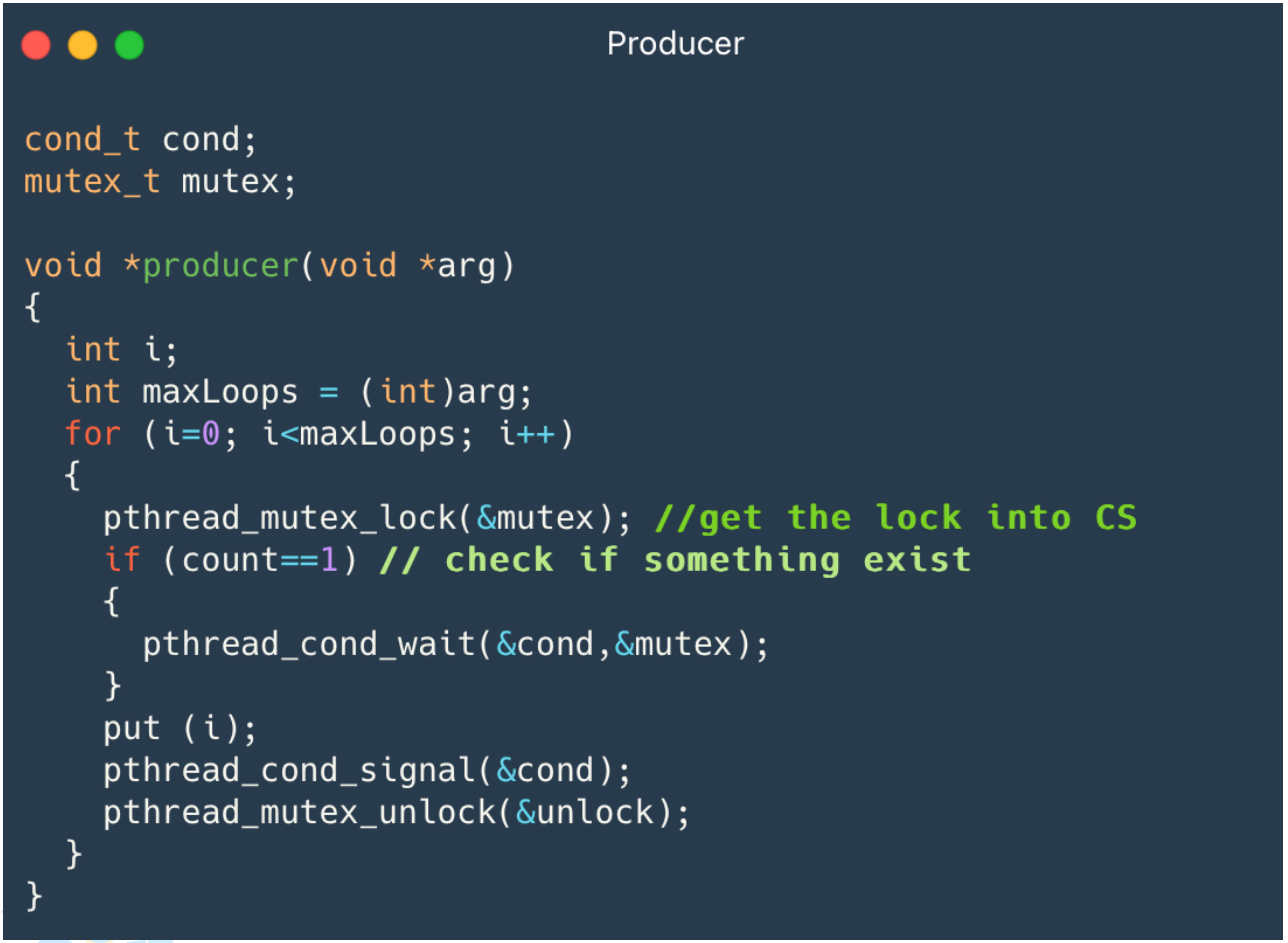


```
void *producer (void *arg)
{
  int i;
  int maxLoops = (int) arg;
  for (i=0; i<maxLoops; i++)
  {
    put(i);
  }
}

int *consumer(void *arg)
{
  int value;
  while (1)
  {
    value = get();
    printf("%d\n", value);
  }
}
```

Surround with Locks and Condition Variables

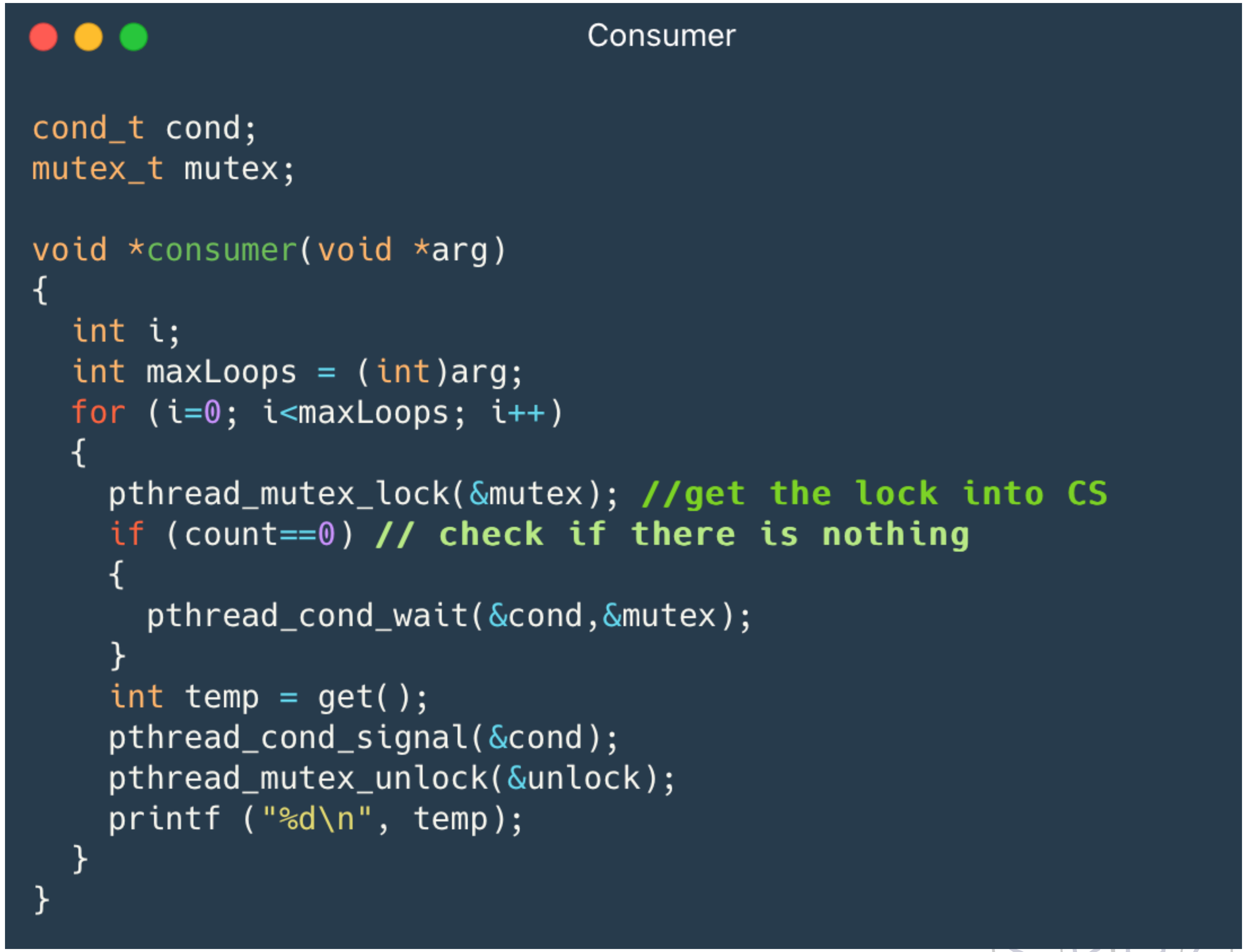
Only one producer and one consumer



Producer

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg)
{
  int i;
  int maxLoops = (int)arg;
  for (i=0; i<maxLoops; i++)
  {
    pthread_mutex_lock(&mutex); //get the lock into CS
    if (count==1) // check if something exist
    {
      pthread_cond_wait(&cond,&mutex);
    }
    put (i);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&unlock);
  }
}
```



Consumer

```
cond_t cond;
mutex_t mutex;

void *consumer(void *arg)
{
  int i;
  int maxLoops = (int)arg;
  for (i=0; i<maxLoops; i++)
  {
    pthread_mutex_lock(&mutex); //get the lock into CS
    if (count==0) // check if there is nothing
    {
      pthread_cond_wait(&cond,&mutex);
    }
    int temp = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&unlock);
    printf ("%d\n", temp);
  }
}
```

What if there are multiple consumers?

SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere