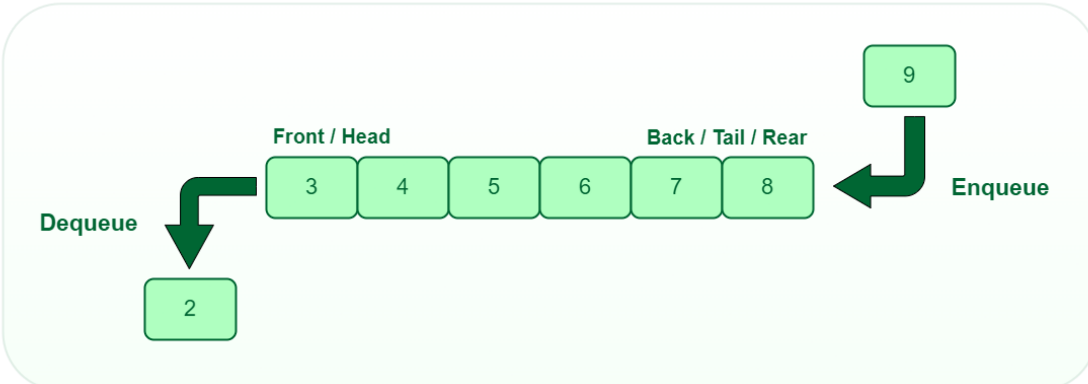
Introduction to Parallel Computing

Shared-memory programming with OpenMP

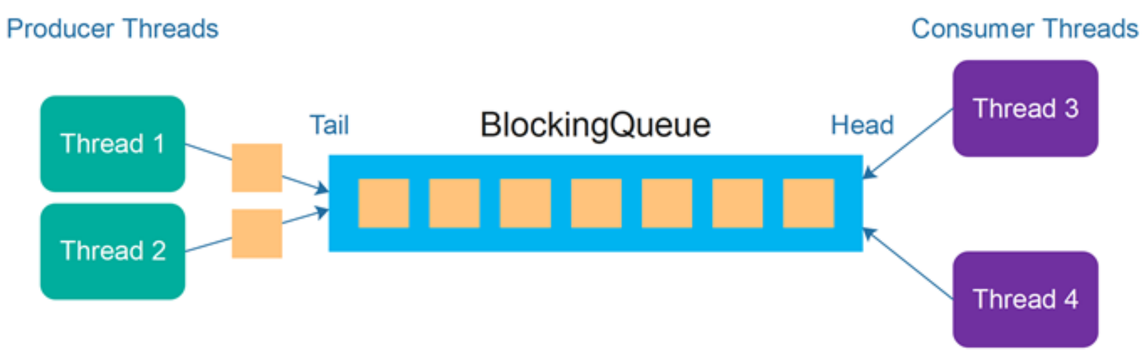
Table of Contents

Producers and consumers

- A **queue** is a data structure in which elements are inserted at the rear of the queue and removed from the front of the queue.
    - Adding elements → enqueue
    - Removing elements → dequeue



- Queues appear in many **applications**:
    - Print spooling
    - Operating systems scheduling/task scheduling
    - Data buffers in networking
    - Load balancing in web servers
- Queues are fundamental in **producer-consumer** applications:
    - Part the of the application is **producing** data, and another part is **consuming** the data



- Let's implement a **message-passing** application that is based on the **producer-consumer** pattern.
    - Each thread has a **shared-message queue**.
    - Each thread generates random (integer values) messages and random destination for the messages.
    - A **producer** thread **enqueues** the message in the queue.
    - A **consumer** thread checks if the queue received a message, and then **dequeues** it.
    - Each thread **alternates** between sending and receiving messages.

- The user will specify the **number of messages** each thread should send.
- When a thread is **done sending messages**, it receives messages until all the threads are done, at which point all the threads **quit**.

- Pseudocode for each thread might look something like this:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

- Core segments/functions of the program:
  - Sending messages → $Send\_msg()$
  - Receiving messages → $Try\_receive()$
  - Terminating the program → $Done()$

Sending messages

- Accessing a **message queue** to **enqueue** a message is a **critical section**.
  - We need to regulate how threads access the queues to avoid dropping messages.
- To successfully **enqueue** a message, we need a pointer to the **rear** of the queue.
  - When we enqueue a message, we'll update the rear pointer.

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

  - The $Enqueue$ functions handles updating the rear pointer.
  - Note that this allows a thread to send a message to itself.

Receiving messages

- **Only the owner of the queue (that is, the destination thread) will dequeue from a given message queue.**
- If there are at least two messages in the queue, a call to $Dequeue$ **can't conflict** with any calls to $Enqueue$.
- Before $Dequeue$ing the queue, **we must check its size**; if 0, return none. If the size is >=1, then return the element at the front.
  - We store two variables: $enqueued$, which is the number of added elements, and $dequeued$, which is the number of removed elements.

```
    queue_size = enqueued - dequeued;
    if (queue_size == 0) return;
    else if (queue_size == 1)
#      pragma omp critical
       Dequeue(queue, &src, &mesg);
    else
       Dequeue(queue, &src, &mesg);
    Print_message(src, mesg);
```

- The $critical\ section$ is added to prevent other threads from updating the queue when the owner thread is $Dequeue$ing it when it has only one element.

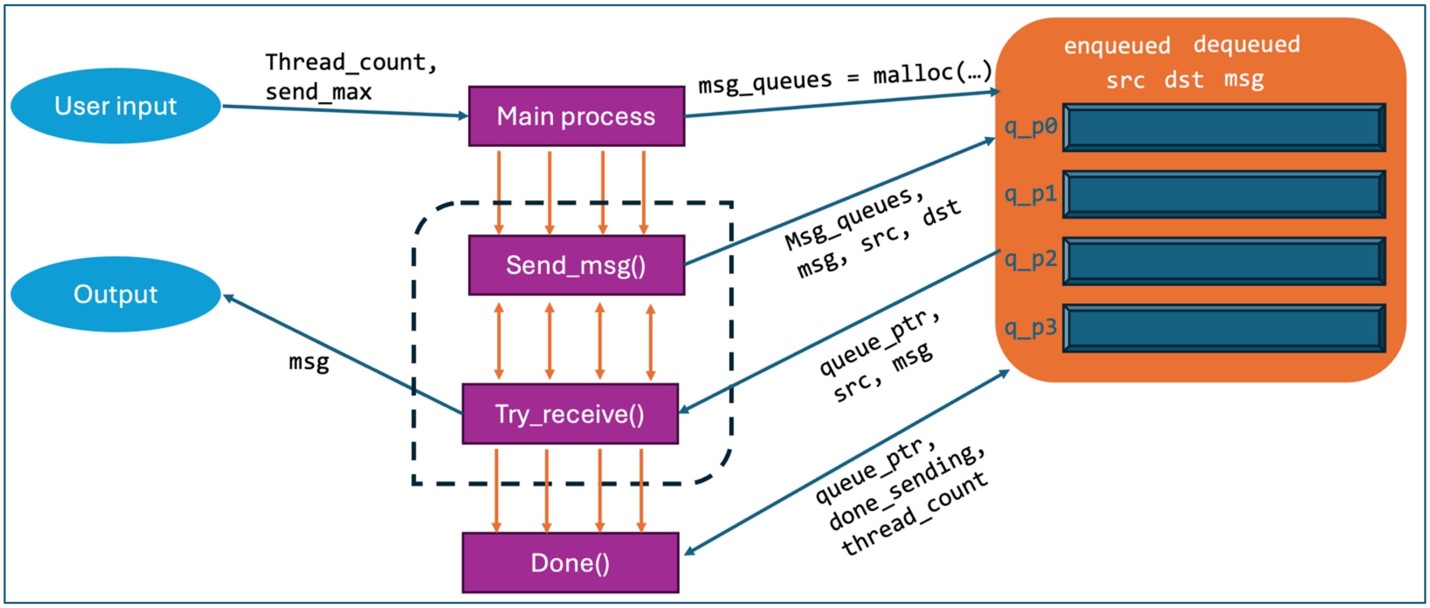
Terminating the program

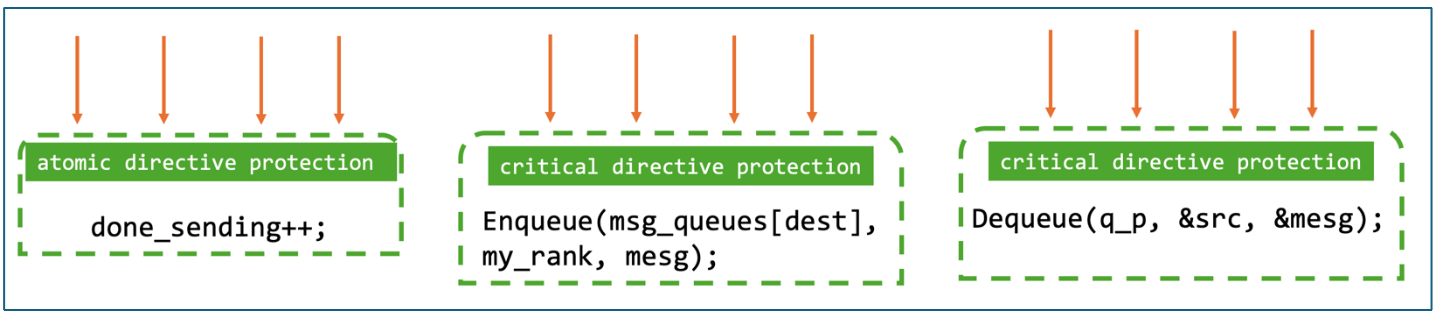
- The **naïve way** to terminate the program (implementing $Done()$) is as follows:

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

- The problem with this way is that it is possible that the owner thread compute $queue\_size = 0$, while another thread is adding an element into the queue.
    - Thus, the added message will never be received by the owner thread.
- Instead, we can include $done\_sending$ as a counter variable, so it serves as a flag when all threads have no more messages to send.

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```






```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include "queue.h"
// gcc-14 -o prog -fopenmp -DDEBUG omp_msgps.c queue.c

/*-------------------------------------------------------------------*/
void Send_msg(struct queue_s *msg_queues[], int my_rank, int
thread_count, int msg_number) {
    int mesg = -msg_number;
    int dest = random() % thread_count;
#  pragma omp critical
    Enqueue(msg_queues[dest], my_rank, mesg);
    printf("Thread %d > sent %d to %d\n", my_rank, mesg, dest);
}
/*-------------------------------------------------------------------*/
void Try_receive(struct queue_s *q_p, int my_rank) {
    int src, mesg;
    int queue_size = q_p->enqueued - q_p->dequeued;

    if (queue_size == 0) return;
    else if (queue_size == 1)
#      pragma omp critical
        Dequeue(q_p, &src, &mesg);
    else
        Dequeue(q_p, &src, &mesg);
    //printf("Thread %d > received %d from %d\n", my_rank, mesg, src);

}
```

```c
/*-----------------------------------------------------------------*/
int Done(struct queue_s *q_p, int done_sending, int thread_count) {
    int queue_size = q_p->enqueued - q_p->dequeued;
    if (queue_size == 0 && done_sending == thread_count)
        return 1;
    else
        return 0;
} /* Done */
/*-----------------------------------------------------------------*/
int main(int argc, char *argv[]) {
    int thread_count;
    int send_max;
    struct queue_s **msg_queues;
    int done_sending = 0;

    thread_count = 4;
    send_max = 5;

    msg_queues = malloc(thread_count * sizeof(struct queue_node_s *));

#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(thread_count, send_max, msg_queues,
done_sending)
    {
        int my_rank = omp_get_thread_num();
        int msg_number;
        srandom(my_rank);
        msg_queues[my_rank] = Allocate_queue();

#     pragma omp barrier /* Don't let any threads send messages  */
        /* until all queues are constructed     */

        for (msg_number = 0; msg_number < send_max; msg_number++) {
            Send_msg(msg_queues, my_rank, thread_count, msg_number);
            Try_receive(msg_queues[my_rank], my_rank);
        }

   // done_sending is a critical section
  // using atomic instead of critical section has a better performance
#     pragma omp atomic
        done_sending++;

        while (!Done(msg_queues[my_rank], done_sending, thread_count))
            Try_receive(msg_queues[my_rank], my_rank);

      /* My queue is empty, and everyone is done sending             */
      /* So my queue won't be accessed again, and it's OK to free it */
        Free_queue(msg_queues[my_rank]);
        free(msg_queues[my_rank]);
    } /* omp parallel */

    free(msg_queues);
    return 0;
} /* main */
```

- When the program begins execution, the master thread will get command-line arguments and allocate an array of message queues, one for each thread.
- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.
- We use $omp\ barrier$ to add an explicit barrier so that no thread starts enqueuing or dequeuing until all threads have constructed their queues to avoid errors.
- After completing its sends, each thread increments $done\_sending$ before proceeding to its final loop of receives.
    - Incrementing $done\_sending$ is a **critical section**, and we could protect it with a $\boldsymbol{critical}$ **directive**.
    - Instead, we use **a higher performance directive**, $omp\ atomic$
- $omp\ atomic$ can only **protect** statements with **a single assignment**.
- For example, $\boldsymbol{omp\ atomic}$ protect statements of the following form:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

    - $< expression >$ **must not reference** x
    - $< op >$ can be + , $*$,−, / , &, ^ , | , <<, or > >
- $atomic$ directive has special for $\boldsymbol{load - modify - store}$ instructions in modern processors, which is **more efficient** than $critical\ section$

Experiments

1. Run the program as is – with **4 threads and 5 messages** to send and check the output.
    a. You will see that each thread must **send exactly 5 messages**.
    b. Threads **may or may not** receive 5 messages.
2. **Remove the critical section** from the $Try\_receive$ function and run the program **with 4 threads and 5 messages**. The $Try\_receive$ function should look like this:

```
/*------------------------------------------------------------------*/
void Try_receive(struct queue_s *q_p, int my_rank) {
    int src, mesg;
    int queue_size = q_p->enqueued - q_p->dequeued;
    if (queue_size == 0) return;
        Dequeue(q_p, &src, &mesg);
    //printf("Thread %d > received %d from %d\n", my_rank, mesg, src);

} /* Try_receive */
```

    a. **The program works fine without any issues.**

3. **Revert the *Try_receive* function** and run the program with **10 threads and 10000** messages.
    a. You notice that the program may **take a little longer, but it finishes successfully.**
4. **Remove the critical section** again from *Try_receive* and run the program **with 10 threads and 10000 messages.**
    a. It will either run for ever or it will crash because of heap errors.

```
prog(11076,0x16b79b000) malloc: Heap corruption detected, free list is damaged at 0x60000004ebf0
*** Incorrect guard value: 0
Thread 5 > sent -1063 to 4
Thread 4 > sent -1012 to 5
Thread 3 > sent -1097 to 1
prog(11076,0x16bdbf000) malloc: *** error for object 0x60000004c3e0: pointer being freed was not allocated
prog(11076,0x16bdbf000) malloc: *** set a breakpoint in malloc_error_break to debug
Thread 0 > sent -1149 to 5
prog(11076,0x1eb99c840) malloc: *** error for object 0x60000004c350: pointer being freed was not allocated
przsh: abort      ./prog
```

5. Run the **same program** but with **100 threads and 100,000 messages**.
    a. The program is **very likely** to run for ever.
6. **Revert the *Try_recieve* function** and run with **100 threads and 100000** messages.
    a. The program **takes a few minutes** and **finishes successfully**.

- *These experiments demonstrate the importance of handling **special edge-cases**, in which the termination of a thread/program is dependent on state of the object (whether the queue is empty or not). With small inputs (e.g., 4 threads and 5 messages) errors **may not** occur. But when the problem size's increases (10 threads and 100000 messages) problems occur and hard to detect.*
- Go through the same program but with the textbook version.

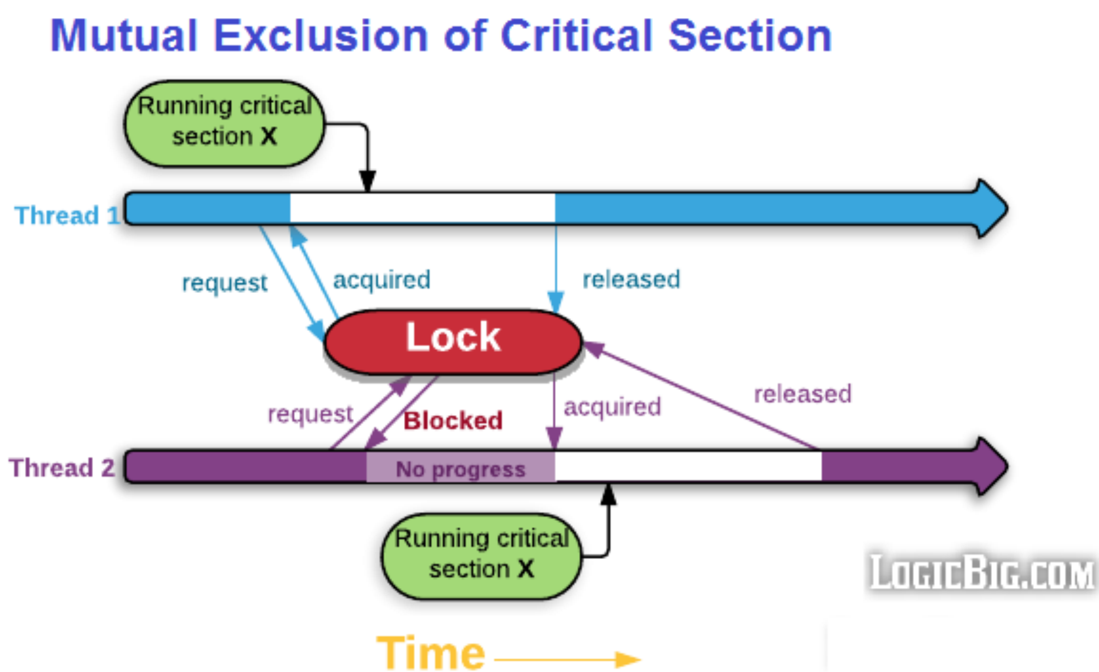
Locks

- As an alternative to *critical* sections, we can use ***Locks***.
- A *lock* consists of a **data structure/functions** enforce mutual exclusion to data or resources.
- Defining locks follows this pseudocode:

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

- The lock data structure is **shared among the threads** that will execute the critical section.
- **One** of the threads will **initialize** the lock.
- **One** of the threads will **destroy** the lock.
- A thread entering the critical section will ***set*** the lock.
- When the thread finishes the critical section, it will ***releases*** or ***unset*** the lock.

- How locks achieve mutual exclusion:



- Locks in OpenMP:

Action	Function
Initialize a lock	`void omp_init_lock (omp_lock_t* lock_p);`
Acquire a lock	`void omp_set_lock (omp_lock_t* lock_p);`
Release a lock	`void omp_unset_lock (omp_lock_t* lock_p);`
Remove a lock	`void omp_destroy_lock (omp_lock_t* lock_p);`

Using locks in the message-passing program

- In the previous program, we used ***critical*** **directive** to enforce a mutually exclusive access to a **shared resource**.
  - `done_sending++;`
  - `Enqueue(q_p, my_rank, mesg);`
  - `Dequeue(q_p, &src, &mesg);`
- There is one efficiency issue with this critical section: the *critical* directive **blindly** allows only one thread to **Enqueue** or **Dequeue**.

- But we don’t need to block threads that **don’t conflict with each other**: for instance, it’s safe for **thread 0** to enqueue a message in **thread 1’s queue** <u>at the same time</u> that **thread 1** is enqueuing a message in **thread 2’s queue**.
    - **So, the *critical* directive allows only one thread do *enqueue*ing at a time regardless the process causes conflicts or not.**
- To **overcome** this, we use *locks* and instead of making the **whole function** of *enqueue* or *dequeue* a **critical section**, we will make the **corresponding queues only critical sections.**
- So, we can do the following:

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

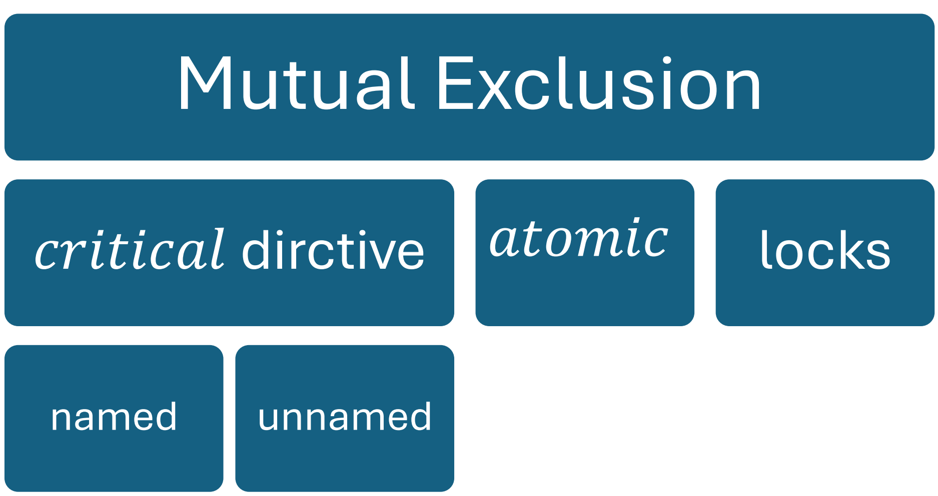
```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

- **Check the source code** in $queue\_lk.h$, $queue\_lk.c$, and $omp\_msglk.c$.
    - **The *lock* is defined as a data member in the *queue* struct.**
- Now when a thread tries to send or receive a message, it can only be **blocked** by a thread accessing the **same message queue**, since **different message queues have different locks**.
- In previous implementation, **only one thread** could **send** at a time, regardless of the destination.

Critical directives, *atomic* directives, or locks?

- So far, we introduced **three** basic mutual exclusion mechanisms:



Mechanism	Properties	Syntax	Group regions under one directive?

atomic	Simple and the fastest	`#pragma omp atomic`
		`x <op>= <expression>;`
		`x++;`
		`--x;`
			Yes

***critical* (unnamed)**	Easier than locks	`#pragma omp critical`
		`{`
		…
		`}`
			Yes

***critical* (named)**	-	`#pragma omp critical(name)`
		`{`
		…
		`}`
			No

Locks	Better for data structures	`omp_set_lock(lock);`
		…
		`omp_unset_lock(lock);`
			No

- Critical regions specified by *atomic* or *critcal* (unnamed) directives **<u>may</u>** treat all their regions as one block.
- For example:

```
#pragma omp parallel
{
    #pragma omp atomic
    var1 = var1 + 1;   // Atomic operation on var1

    #pragma omp atomic
    var2 = var2 + 1;   // Atomic operation on var2
}
```

  - **In one implementation**, the runtime might enforce exclusive access, meaning that the operations on $var1$ and $var2$ cannot happen at the same time, even though they operate on different variables.
  - **In another implementation**, the runtime might allow these operations to run concurrently if they access different variables, as there's no dependency or conflict between the two operations.
- The same applies if using (**unnamed**) *critical* directive:

```
#pragma omp parallel
{
    #pragma omp critical
    {
        var1 = var1 + 1;
    }

    #pragma omp critical
    {
        var2 = var2 + 1;
    }
}
```

- **Named** *critical* directive and **locks** avoid that.

```
#pragma omp parallel
{
    #pragma omp critical(section1)
    {
        var1 = var1 + 1;  // Atomic operation on var1
    }

    #pragma omp critical(section2)
    {
        var2 = var2 + 1;  // Atomic operation on var2
    }
}
```

Some caveats

1. **Don't mix** the different types of mutual exclusion for a **single** critical section.
   - The code below uses two mutual exclusive mechanisms for the variable x.

```
#  pragma omp atomic          #  pragma omp critical
   x += f(y);                    x = g(x);
```

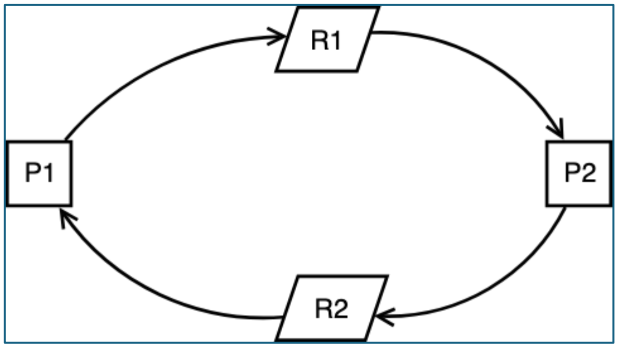
   - It's possible that the two sections get executed **concurrently**, leading to **incorrect** results.
   - You can either: **use** *critical* directive **for both sections** or **rewrite** $g()$ to have the **form** required by the *atomic* directive.
2. There is no **guarantee of fairness** in mutual exclusion constructs.
   - For example, the code below may allow one thread to **always** be executing the function g, thus preventing other threads from accessing it.

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

   - This won't happen if the *while* loop **terminates**.
3. It can be dangerous to "**nest**" mutual exclusion constructs.

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

- This will cause **deadlock**: a situation in which a group of threads are waiting for each other thread to finish.



- If a thread is executing the first block, it won't be able to **enter the second block**. At the same time, it **will not leave the first block** until it proceeds to the second block.
- One possible solution is use **named** *critical* sections

```
#   pragma omp critical(one)
    y = f(x);
    . . .
    double f(double x) {
#       pragma omp critical(two)
        z = g(x);  /* z is global */
        . . .
    }
```

- But this is not the **ultimate** solution, as deadlocks can occur as following:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block