Introduction to Operating Systems and SQL for Data Science

Lecture 3 – Threads

Previous lecture

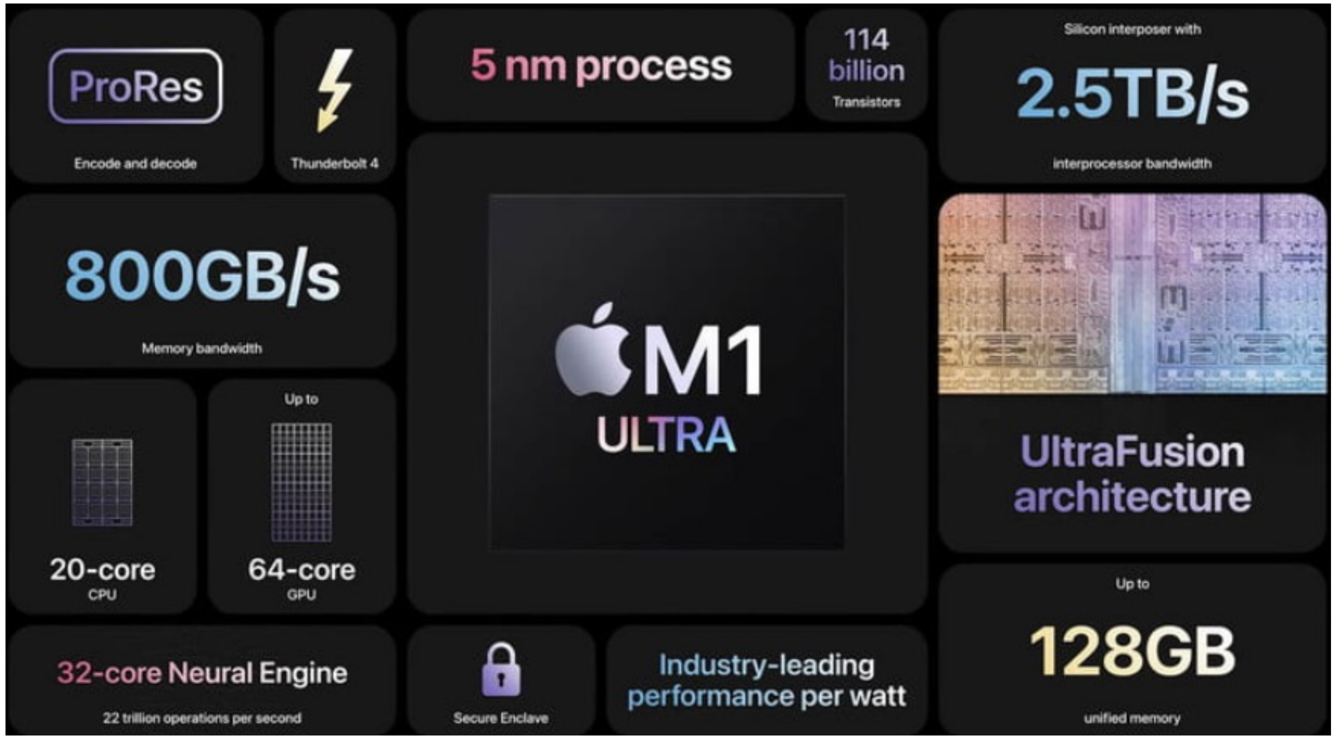
- Scheduling
- Context Switch
- Scheduling algorithm - (non) preemptive
- Batch vs. Interactive systems
- Policies – FCSFS, SJF, RR

Previous lecture

- Scheduling
- Context Switch
- Scheduling algorithm - (non) preemptive
- Batch vs. Interactive systems
- Policies – FCSFS, SJF, RR

Current affairs

The most modern chip is announced:



Has only 20-core CPUs. Why on GPU we always have more cores? (most modern gpu has 10496-cores)

Reichman University

Shared address space – GUI example

Apple/macOS GUI

Shared address space – GUI example

i. Different events are received - click of a button, typing text.

ii. Each event can take processing time. (such as search)

iii. We want the system to be able to receive additional events at the same time. (the system won’t get blocked)

iv. Thus, we want one process that waits for events, and directs them to dispatcher.

Reichman University

Shared address space – GUI example

iv. We want many processes that handle events. (worker threads)

v. We want a process that collects results and displays them in the GUI.

vi. All processes transmit information to each other.

Shared address space – Word example

Another example is Word, we want that a (automatic) saving will done in the background.

So, we could still use the GUI of the word and doing other operation. (such as writing)

So how can we do it?

Threads

Threads

A mini processes that sharing the address space.

Its faster to create new threads, because they have no separate memory. (Sharing the same AS)

Let's see how we run threads…

Thread of execution

Process is a union of resources (stack, data)

Threads are entities that runs the code on a CPU.

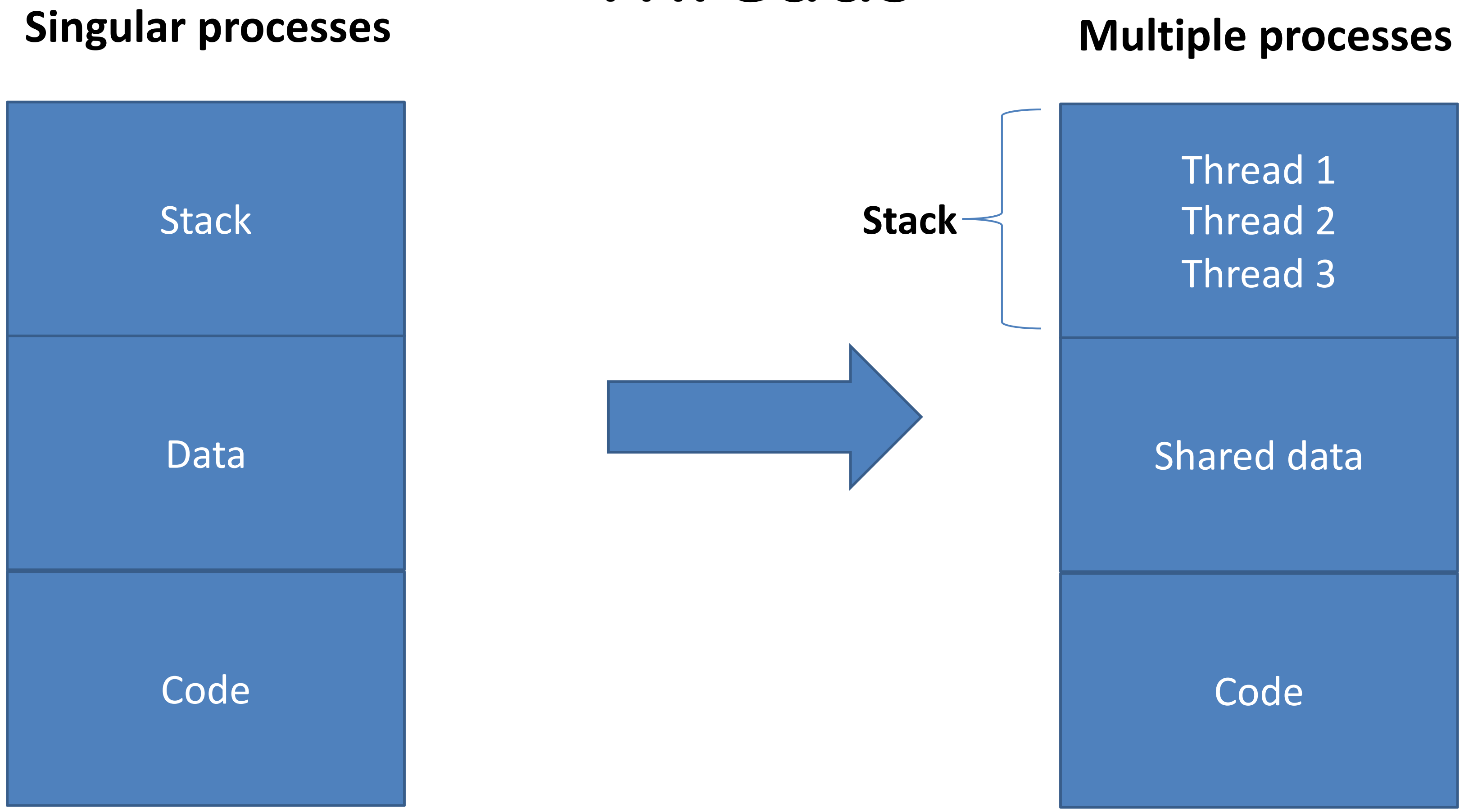
We can run many threads in a single process.

Every threads has its own stack of function, but the memory (heap) is combined.

Each thread can overwrite other thread stack.

Processes are staring with one thread and can start more.

Threads



The stack is the only component which the threads are separated (while they can overwrite each other stack)

Every thread is pointed on some location in the code

The communication of the different threads is through the data

Threads

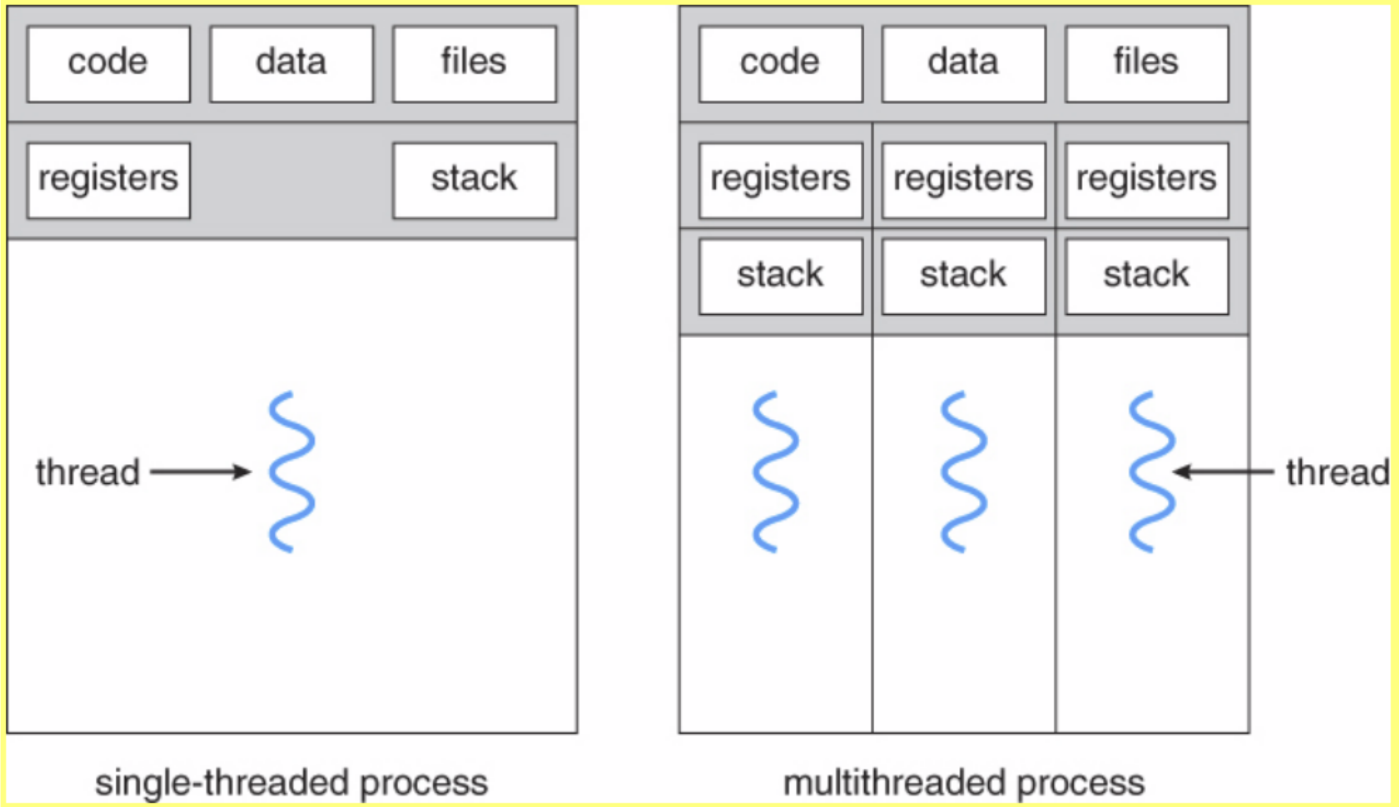


Figure 4.1 - Single-threaded and multithreaded processes

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Threads creation

A process is starting with one thread.

The process can create another thread.

In order to create another thread, we need a starting point – Function.

Threads creation

The new thread is starting to run while the first thread is running.

If we look on the following code:

```
Thread t = new Thread(f);
t.start()
f();
t.join();
```

```
Thread t = new Thread(f);
t.start()
f();
t.join();
```

Threads creation

We can see that the function f will run twice.

One time on the first thread, the second time on the new thread.

The join function order the first thread to wait for the new thread.

Race conditions

When a Threads sharing memory, there is a chance that two threads are accessing into a same location simultaneously.

Let's see an example of adding an element into a linked list.

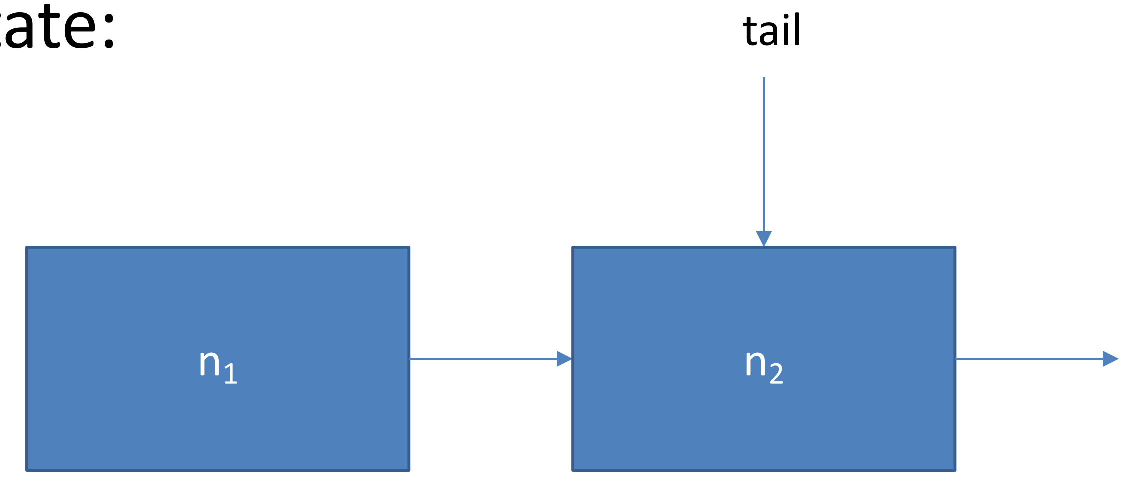
Adding a new node to an end of the list:

```
a. Node newTail = newNode(data);
b. tail.next = newTail;
c. tail = newTail;
```

Race - linked list

```
a. Node newTail = newNode(data);
b. tail.next = newTail;
c. tail = newTail;
```

Initial state:

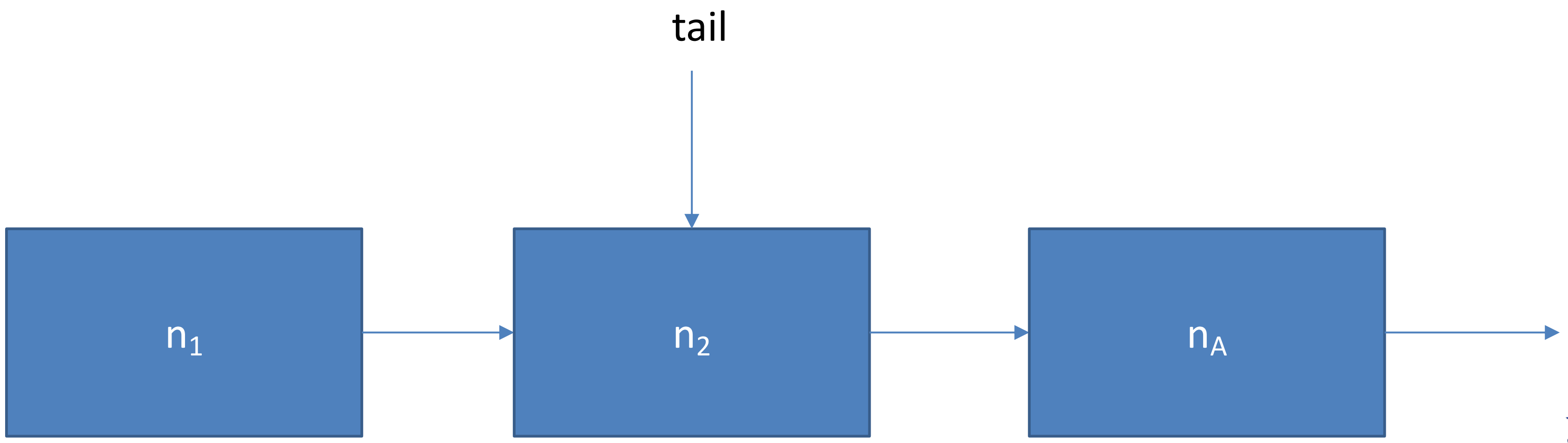


Race - linked list

```
a. Node newTail = newNode(data);
b. tail.next = newTail;
c. tail = newTail;
```

Thread A is finish line b and blocked (added n_A to the list).

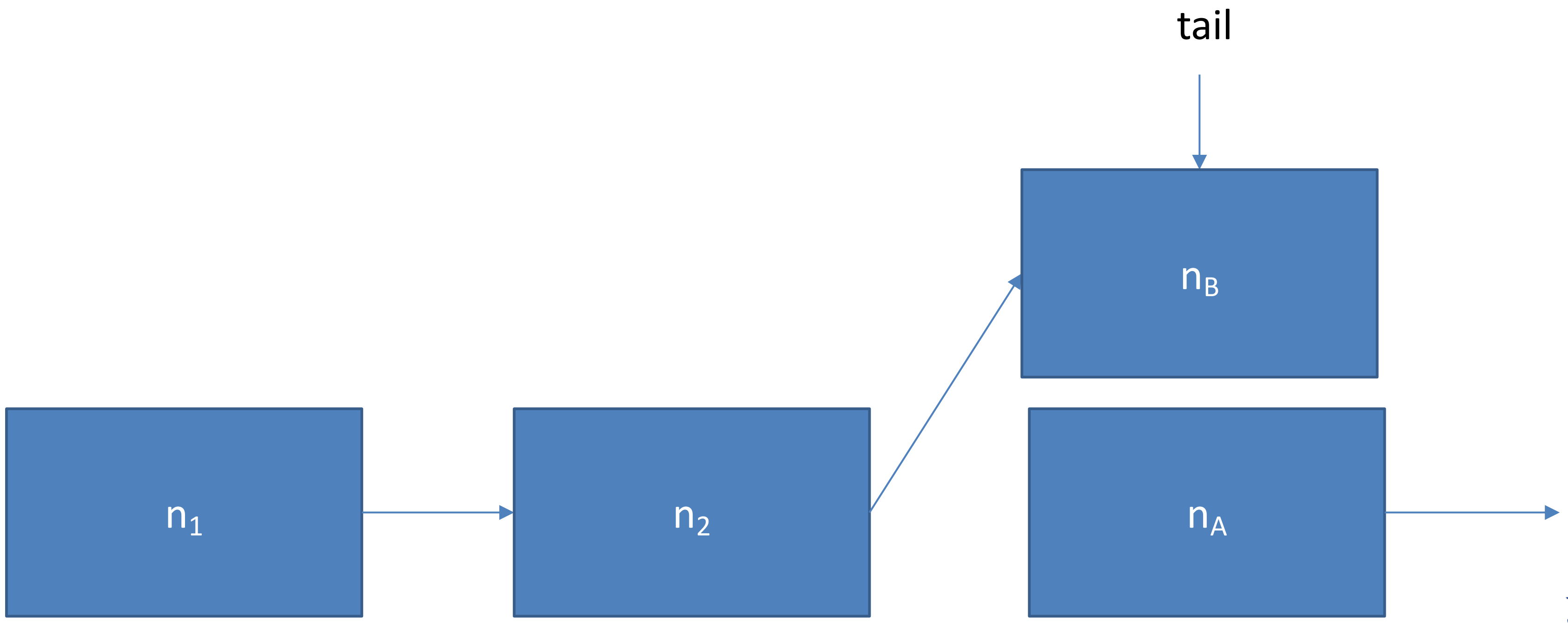
Pay attention, tail is not set to n_A



Race - linked list

```
a. Node newTail = newNode(data);
b. tail.next = newTail;
c. tail = newTail;
```

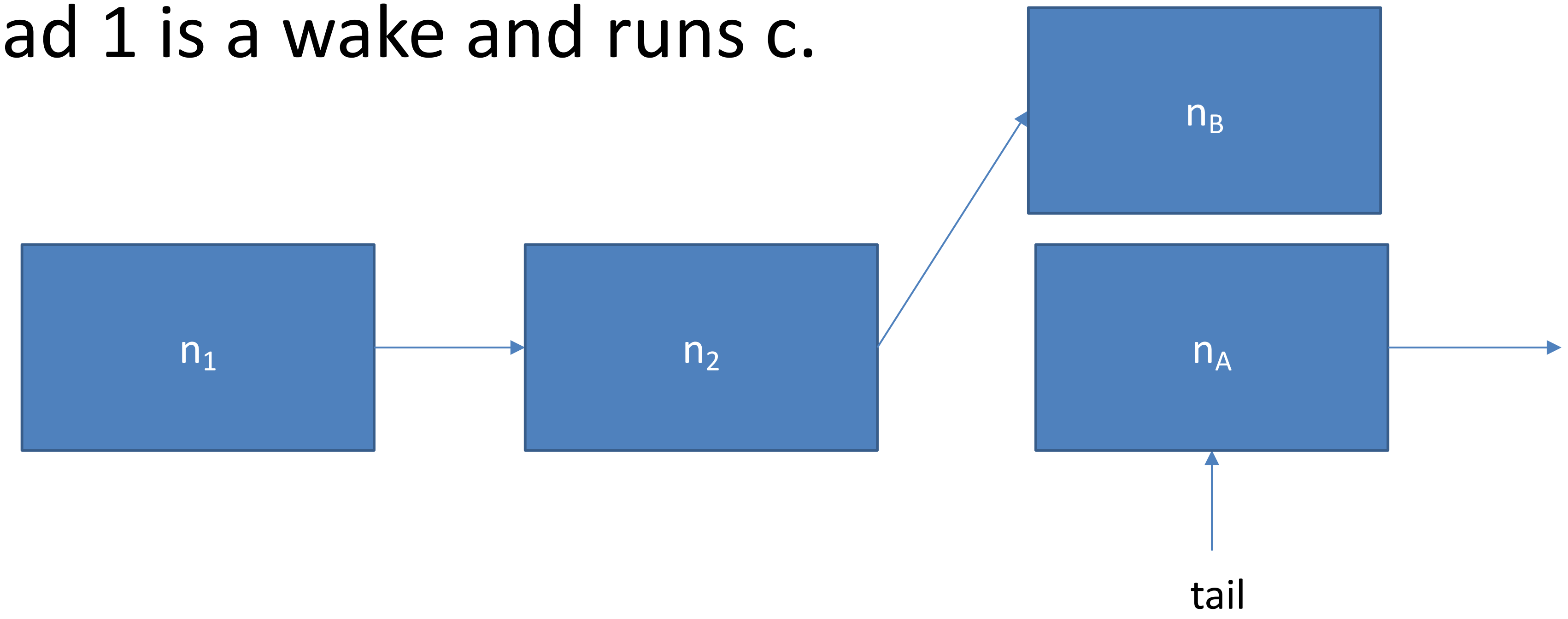
Thread B finish all lines a-c (added a n_B to the list).



Race - linked list

```
a. Node newTail = newNode(data);
b. tail.next = newTail;
c. tail = newTail;
```

Thread 1 is a wake and runs c.



At the end of the execution, a non legit list created.

Race formal definition

When two threads access the same resource for reading or writing and the result depends on the running order.

Critical section (cs) – a code section that could be a race, very hard to detect in running time. Can’t be reproduce each code execution.

We need to detect minimal CS line.

```
b. tail.next = newTail;
c. tail = newTail;
```

CS

Race Vs. bug

We need to distinguish between a bug and a race.

A race is happening in a specific order, but in other orders the running will be fine.

There could be a parallel bug where in all possible ordering there will be a bug.

It is hard to detect races. Since only in rare cases there is a bug, and it will be hard to detect.

Mutual exclusion

Mutual exclusion – avoid a situation where 2 threads are in critical section.

A (failed) attempt to solve this issue:

1. Declare a lock variable.

```
bool m_bUpdating = false;
```

2. In the start of the adding function, we add a locking mechanism:

```
while(!m_bUpdating);
 m_bUpdating = true;
```

3. In the end of the function, we add a line:

```
 m_bUpdating = false;
```

Still not good, what happened if the thread would block right after 2.while

Mutual exclusion – Boolean example

We need a scenario that show a race:

We must provide:

1. Initial state.
2. Action series (in a specific order)
3. Final non-legitimate state

```
bool updating = false;
While (!Updating){
        updating = true;
        cs
}
updating = false;
```

Handle races

Mutex

Mutex – an object with 2 operations:

1. Lock – locking the mutex, use before CS.
2. Unlock – unlock the mutex, use after CS.

When a thread is locking a mutex:

If the mutex is open, the mutex is locked and the thread continue.

If the mutex is locked, the thread is blocked.

When the thread is awake it need to lock the mutex again.

Mutex

When a thread is unlocking a mutex:

The mutex is opened and if there are any blocked threads, there are two option of releases:

1. We wake the first in the list.
2. We wake all in the list. (the first one, we bloked all others)

Hence, the Mutex is saving a list of threads that was blocked when they try to do lock.

Mutex – how to use

```
Mutex m = new Mutex();
m.lock();
<CS>
m.unlock();
```

Producer-Consumer

Producer:

```
While(true){
        obj = produce();
        i++;               ⎫
        buffer[i] = obj;   ⎭
```

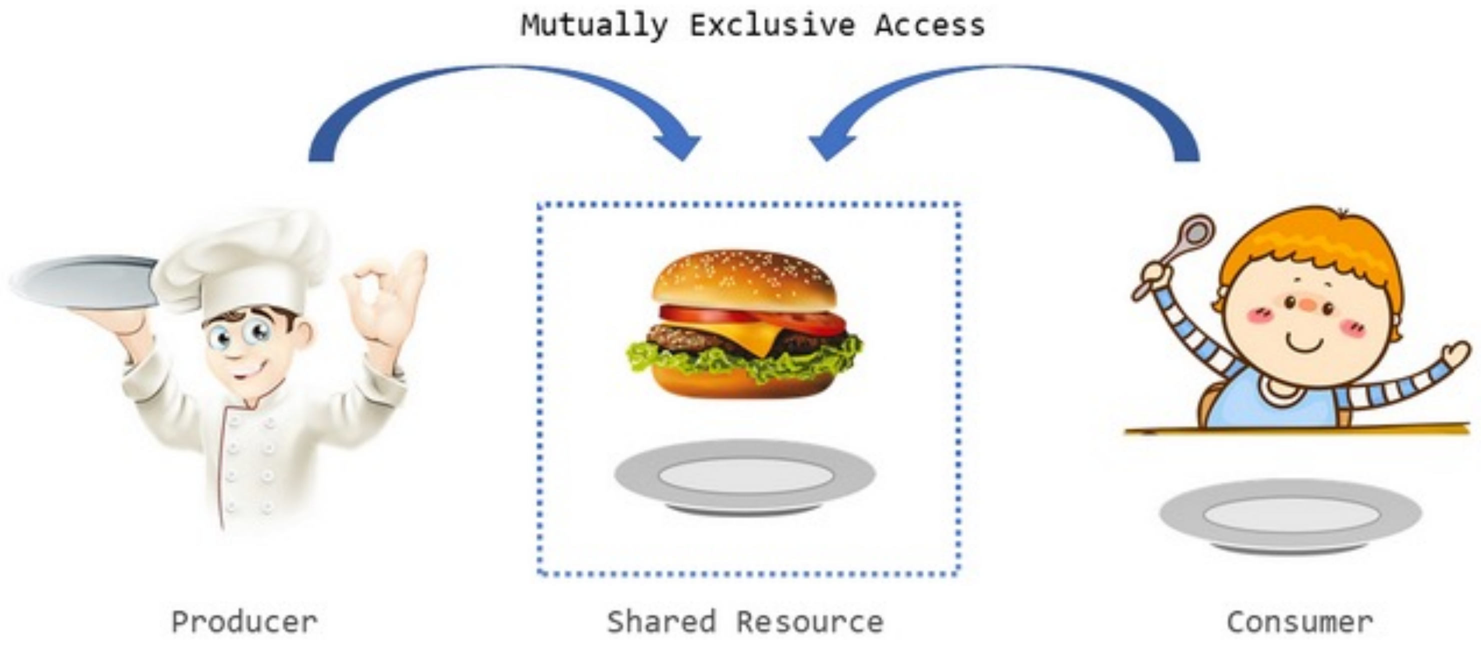
Mutual CS

Consumer:

```
While(true){
      ⎧ obj = buffer[i];
      ⎩ i--;
        Consume(obj);
```

Let’s declare when there is a race

Producer-Consumer



https://levelup.gitconnected.com/producer-consumer-problem-using-mutex-in-c-764865c47483

Producer-Consumer race

Producer:

```
While(true){
        obj = produce();
        i++;
        buffer[i] = obj;
```

Consumer:

```
While(true){
        obj = buffer[i];
        i--;
        Consume(obj);
```

Initial state:

Buffer -> []
i = -1

Producer-Consumer race

```
Producer:

While(true){
      1.obj = produce();
      2.i++;
      3.buffer[i] = obj;
```

```
Consumer:

While(true){
      4.obj = buffer[i];
      5.i--;
      6.Consume(obj);
```

Actions:

Prod is start and blocked between 2-3. (i=0)
Cons is start and finish line 6.
Final non-legitimate state: Cons consume a null object.

Producer-Consumer

Producer:

```
While(true){
    obj = produce();
    i++;
    buffer[i] = obj;
```

Mutual CS

Consumer:

```
While(true){
    obj = buffer[i];
    i--;
    Consume(obj);
```

To solve this race, we need to set mutex as a combined resource.
We can’t use two separate mutexes since, the CS is shared.
This solution is still not covered all edge cases.

Semaphore

We saw that the mutex has solved the producer-consumer problem, but only one producer and consumer can run at parallel. What if we want more than one?

We can use Semaphore. It act the same as the Mutex just is not a binary(lock/unlock) it can count!

Semaphore

The Semaphore has two operations (up, down) and a counter.

If a thread is executed down command and the counter is 0. The thread blocked, and when it wake up, he wakes other threads that are blocked.

If the counter will be set to 1, the semaphore will act as Mutex. The only difference is in mutex only the thread that lock the mutex can open the lock, while in semaphore it’s the opposite.

Semaphore – producer/consumer

We need to set two semaphores.

1. Full – counts the full places.
2. Empty – counts the empty places.
3. Producer:
   a) Down(empty) at start.
   b) Up(full) in the end.

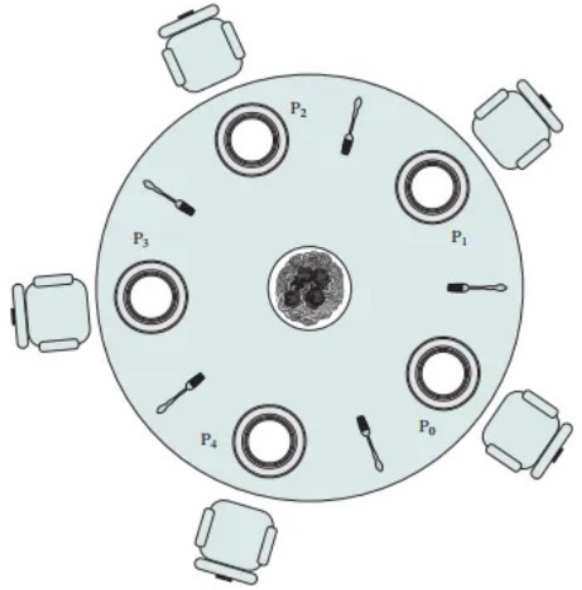
4. Consumer:
   a) Down(full) at start.
   b) Up(empty) in the end.

Deadlock

Dining philosophers' problem

There are n philosophers that are eating and thinking all the time. A philosopher needs exactly two forks.

We can define this problem as a synchronization problem where the philosophers are threads, and the forks are the shared resources.



https://medium.com/science-journal/the-dining-philosophers-problem-fded861c37ed

Dining philosophers' problem

Initial state:

All the philosophers are thinking (not eating) and all of the forks are available.

Steps:

P_1 is finish thinking and is raising frok1 (and then blocked in the middle).

P_2 is finish thinking and is raising fork_4 and fork_1.

Final non-legitimate state:

Two philosophers are holding the same fork

Dining philosophers' problem

How can we solve this problem?

We can use an array of mutex, as such each fork will have a mutex.

Before a philosopher is holding a fork, he had to lock the mutex. And when he finish with the fork, he unlock the mutex.

This could lead into deadlock.

Dining philosophers' problem

Suppose we have 5 philosophers and 5 forks.

Let’s assume that each of the philosophers are raising a fork.

All the philosophers now lock the forks mutex, but they could not eat. Since each needs 2 forks.

Deadlock - definition

Minimal deadlock:

There are two threads t_1, t_2 and two resources r_1, r_2.

We need at least two threads and resources to create a deadlock.

t_1 access r_1

t_2 access r_2

$t_1$access r_2 and get blocked.

t_2 access $r_1$and get blocked.

Deadlock – all the threads in the system are blocked and there is no one who can release its resources.

Deadlock - Necessary conditions

1. Mutual exclusion – an exclusive access to resources.

2. hold & wait – the thread that access a resource can ask for another resource.

3. No preemption – we can’t switch a resource in the middle.

4. Cyclic waiting – there is a cycle of waiting, every thread is waiting for the other one to release its resources.

Reichman University

Deadlock detection algorithm

- Ostrich algorithm – we are ignoring deadlock. Sounds funny but all modern OS are implementing this algorithm.
- Detect and recover: https://www.geeksforgeeks.org/deadlock-detection-recovery/
- Banking algorithm:

https://sajid576.medium.com/bankers-algorithm-1eb83dabaff1

We won’t cover the other algorithms in the course. But you can read it as an extra.