Concurrency

Week 4

Inter-Process Communication

Topics

- Inter-Process Communication
- Shared Memory
  - Critical Section
  - Avoiding Race Conditions
- Message Passing

Learning Outcomes

- At the end of this lesson you will be able to:
  - Describe the problems arising from sharing memory between processes/threads.
  - Describe problems arising from communication between processes/threads through message passing
  - Describe some of the solutions of using share memory or message passing between communicating processes/threads

What is the Main Idea of the Course?

- The following four questions summarize this course.
- Questions:
  1. ***How can we run multiple tasks simultaneously***?
     We discuss multitasking and multi-threading to answer this question.
  2. ***How can simultaneous tasks share resources***?
     We discuss resource types and how they can be shared.
  3. ***What problems may we have when we share resources***?
     We discuss race conditions and the inconsistency of data to answer this question.
  4. ***What solution do we have for these problems***?
     We discuss semaphores and mutex locks to answer this question.

Part One!

Part One!

- In this part, we will discuss the problems arising when shared memory is used for communication between processes/threads and how they are resolved. Try to find answers to the following questions:

1. How can arbitrary interleaving cause problems in accessing shared memory?
2. What is a race condition?
3. What is a critical section?
4. What is mutex?
5. What is mutual exclusion? What is starvation?
6. Why should mutex lock/unlock operations be atomic operations?

Inter-Process Communication

Concurrency (recap)

- A *concurrent program* consists of a finite set of (sequential) processes.
- The processes are written using a finite set of *statements*.
- The execution of a concurrent program proceeds by executing a sequence of the statements obtained by *arbitrarily interleaving* the statements from the processes.

Communication

What we have practised so far was only with:

- disjoint (independent) threads: threads that do not communicate with each other

This week we will practice with threads communicating with each other (cooperating threads)

Communication

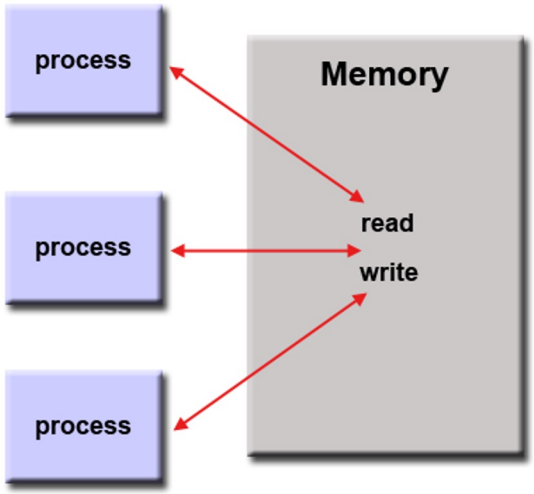
Concurrent processes* need to communicate to :

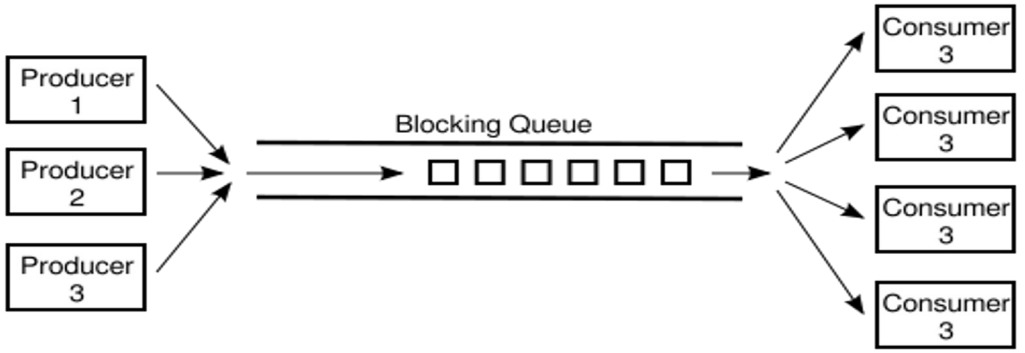
- Share intermediate results
- Convey messages (such as their states)

* A process here is a general concept: we could replace a thread as it is a lightweight process

Communication

- The most common ways of communication:
  - Shared memory: Parts of the memory is shared between processes
    - Main memory, a file, ...
  - Message Passing: Processes inform each other by communicating messages



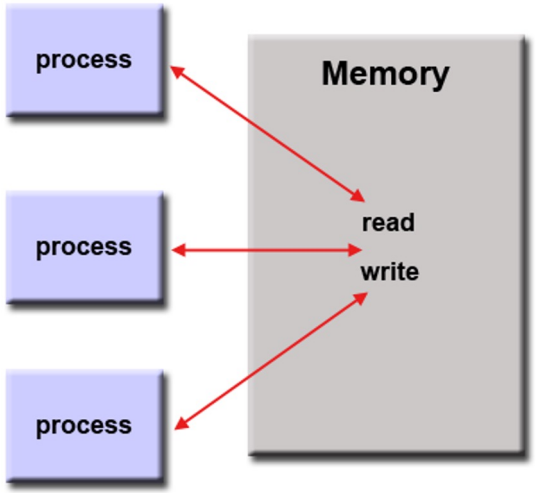


Communication

First, we focus on shared memory and then, we will discuss message passing …

Shared memory:

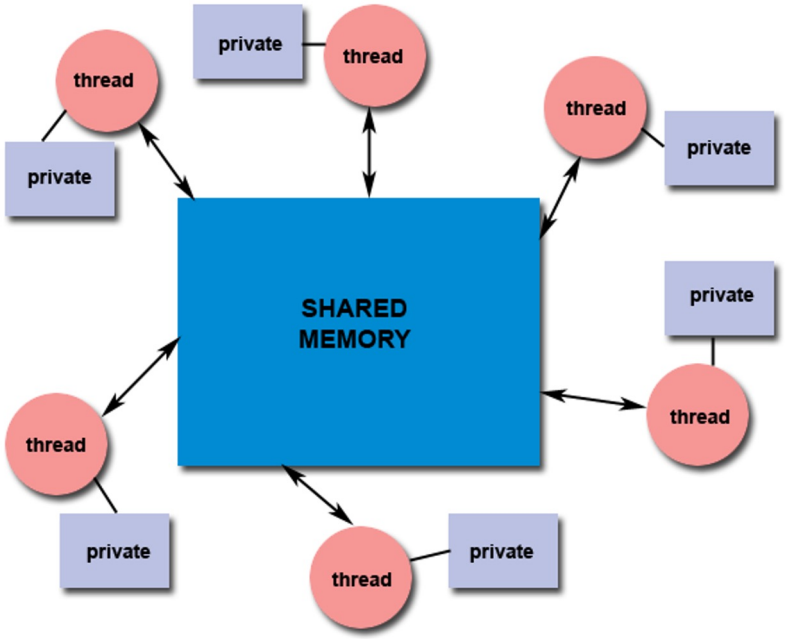
- how to achieve?
- challenges?
- solutions?



Shared Memory

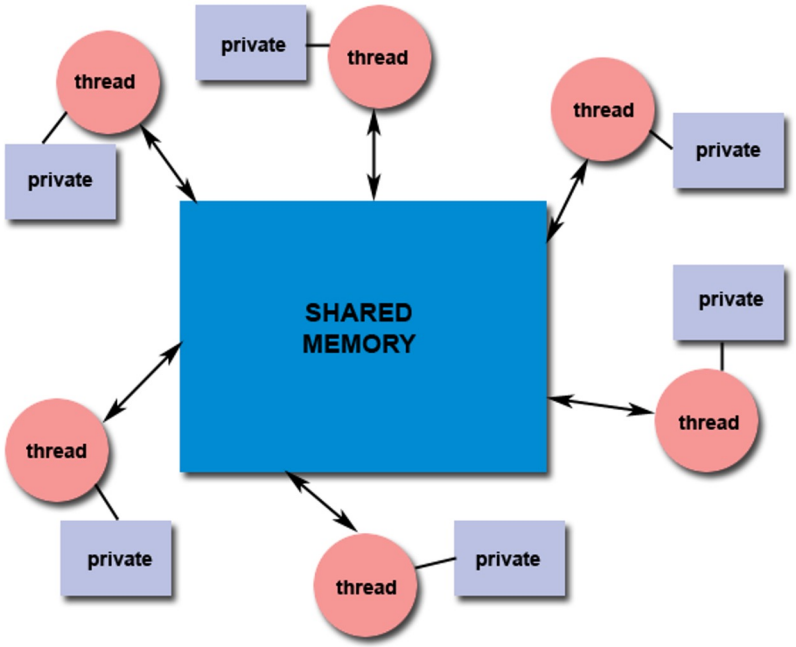
Multiple explicit threads, each thread has access to:

- private memory (pc, stack, local vars):
  - no access to other thread's private memory (local variables)
- shared memory (heap, data):
  - threads can share objects
  - static fields are shared among objects.



Shared Memory

- Multiple threads communicate through the shared memory by:
  - Writing values to a shared location
  - To be read by another thread

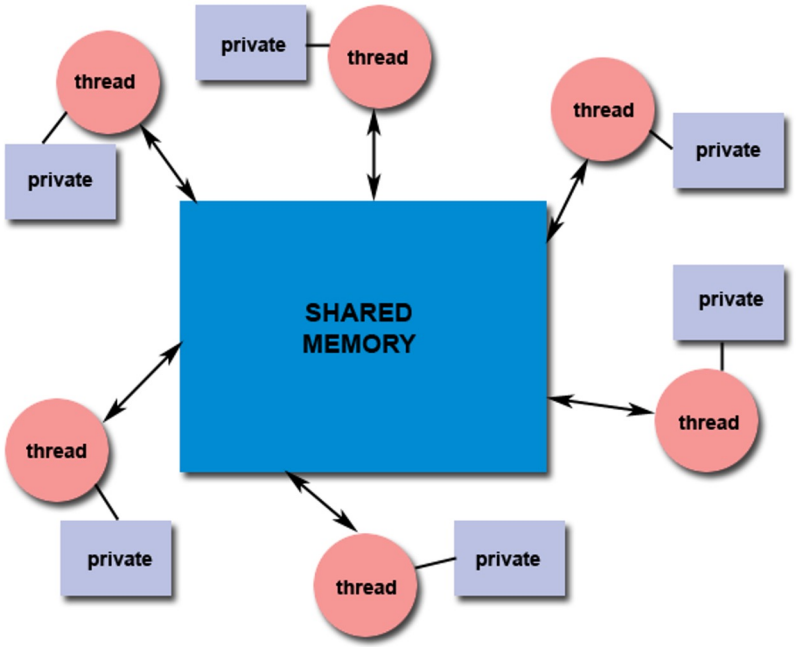


Shared Memory

Q: Two threads are *reading* from shared memory.
Is it safe?

Q: Two threads are *reading* and *writing* from/to shared memory.
Is it safe?

Q: Two threads are *written* to a shared memory.
Is it safe?



Shared resources need protection ...

Concurrency Pitfall

When two or more processes are reading or writing some values from/to shared data,

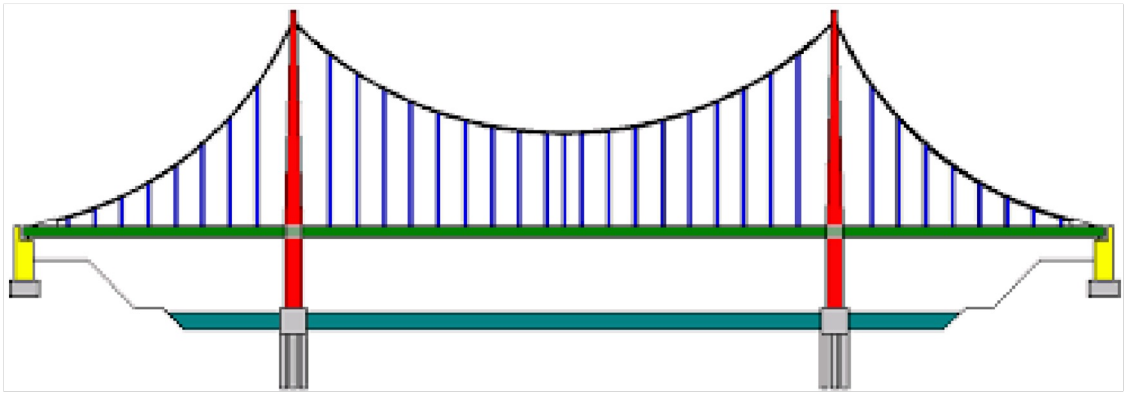
the final result depends on who runs **when**:

- *The order in which different threads access the shared memory is not deterministic.*

The *arbitrary interleaving of* the statements is assumed here.

Example 1

- Assume two observers at two ends of a bridge count the number of passing cars.
- Each observer increments the value of a shared variable whenever it observes a car.



Example 1 (cont.)

Assume the shared variable is named total and is initialized to zero.

Sample code (CPU execution level) run by observers is:

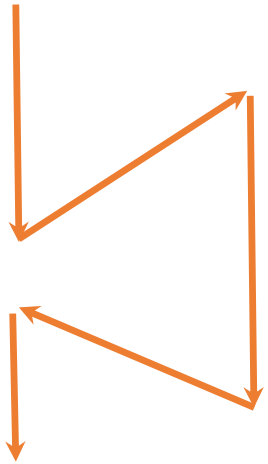
- Temp = read (total)
- Temp = Temp + 1
- Store(Temp, total)

Example 1 (cont.)

Assume the current value of the total is **12:**
run the following execution steps

Process 1

- Temp = read (total)
- Temp = Temp + 1
- Store(Temp, total)



Process 2

- Temp = read (total)
- Temp = Temp + 1
- Store(Temp, total)

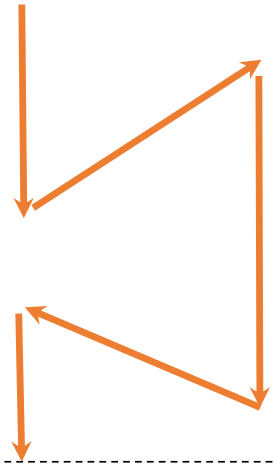
what is the value of total here?

Example 1 (cont.)

Assume the current value of the total is **12:**
run the following execution steps

Process 1

- Temp = read (total)
- Temp = Temp + 1
- Store(Temp, total)



Process 2

- Temp = read (total)
- Temp = Temp + 1
- Store(Temp, total)

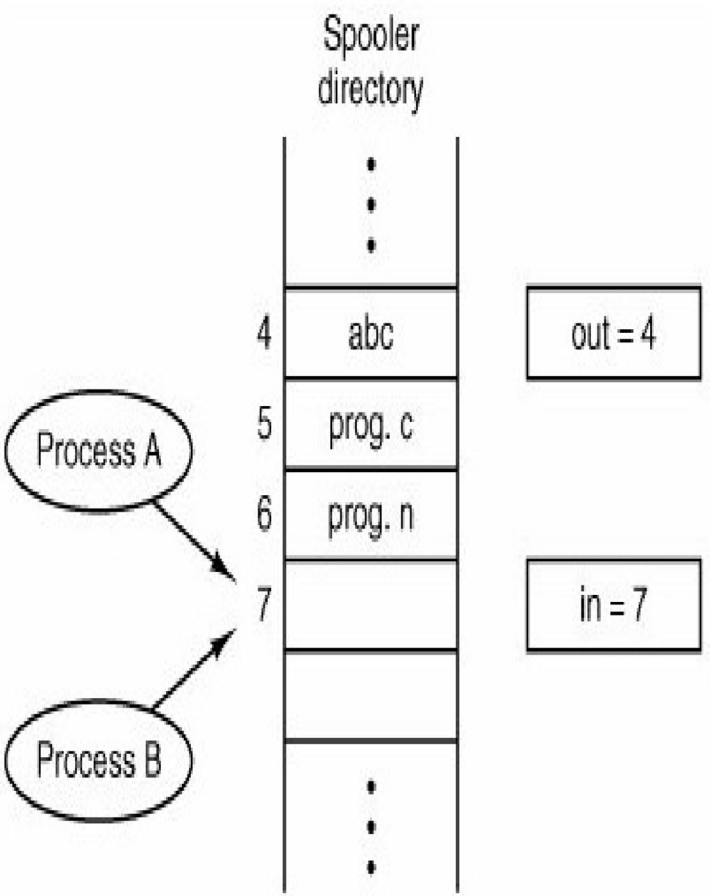
After two observations the value of total becomes **13** instead of **14**

Example 2

- An airline reservation system uses a list to keep track of the seats reserved/sold on each flight.
- Two agents read the list
- Both agents find out that seat 24 on flight ABC123 is available
- The first agent reserves it by updating the list
- The second agent overwrites the updated list reserving the seat for another passenger

Example 3

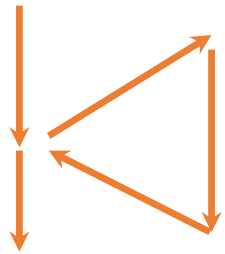
- A shared printer is used in a computer system.
- The data sent by each process are stored at a shared location named a print spool.
- The printer gets the next data item using the index “out”
- The processes use the index “in” to store their data in the print spool
- Each process increments the value of the index “in” after adding its data



Example 3 (cont.)

Process 1

- Index = Read (in)
- Store (data, Spool[index])
- Store (in, index + 1)



Process 2

- Index = Read (in)
- Store (data, Spool[index])
- Store (in, index + 1)

As a result, process 2 overwrites the data stored by process 1

Race Condition

- A **race condition** occurs when some processes or threads can access (read or write) a *shared data* variable *simultaneously* and *at least one of the accesses is a write* (data manipulation).

- The result of the access in presence of *race conditions* may change each time that the code is run, as it *depends on the order of access*.

Critical Section

- Each process has *a segment of code*, called a **critical section**, in which the process may be changing shared data.

Critical Section (2)

Processes access shared resources only at a specific part of their codes.

- Typical sequences are
    - Data = Read( )
    - Results = Process(data)
    - Write(Results, shared_locations)

  Or

    - Data = Access(shared_locations)
    - Results = Process(Data)
    - Print(Results)

Avoiding race conditions...

To avoid race conditions we need to either

- **Avoid** shared resources whenever possible
- **Avoid** global variables

Or

- **Protect** them: Apply *locking* and *mutual exclusion* principles

Avoiding Race Condition (cont.)

In order to avoid race conditions:

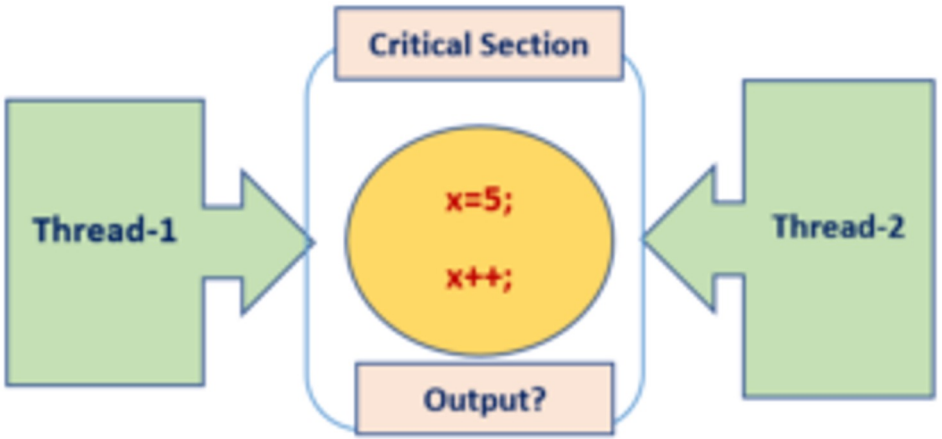
- If a process is in its critical section (using the shared resource), other processes should not be in its critical section (mutual exclusion).

- mutual exclusion:
  - only one process can execute its critical section at a time.

Avoiding Race Condition (cont.)

- A process should be able to enter its critical section if no process is in its critical section.

- No process should wait indefinitely to enter its critical section (starvation).



Mutex

- Mutexes are used to put a lock on shared resources.

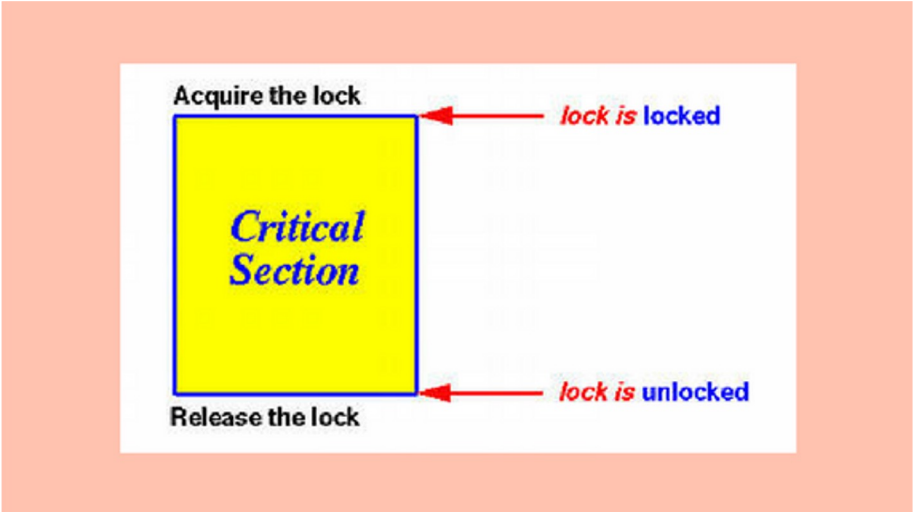
- When a resource is locked, any process trying to access it ***will be blocked***

Mutex etimology: ***mut****ually* ***ex****clusive*

Mutex

- Locking and unlocking should be implemented as *atomic operations* (there will be no context switching before the operation is complete).

- Function lock(mutex) and unlock(mutex) are used to lock and unlock a shared resource.



Mutex: Example

- Assume a library client wants to borrow a book.

  The steps are:
  - *Mutex library*
  - *BookList = GetList( )*
  - *Item = Examine(BookList)*
  - *If available (Item)*
    - *Borrow (Item)*

- What is the shared resource here?
- what is the critical section?

Mutex: Example

- Assume a library client wants to borrow a book.

  The steps are:
  - *Mutex library* ←**Read** access
  - *BookList = GetList( )*
  - *Item = Examine(BookList)*
  - ***If available (Item)*** ←**Write** access
    - ***Borrow (Item)***

- How can we protect the critical section?

Mutex: Example

- Assume a library client wants to borrow a book.

  The steps are:
  - *Mutex library*
  - ***Lock (library)***
  - *BookList = GetList( )* ←**Read** access
  - *Item = Examine(BookList)*
  - *If available (Item)*
    - *Borrow (Item)* ←**Write** access
  - ***Unlock (library)***

- How can we protect the critical section?

Mutex: Example

- Assume a library client wants to borrow a book.
  The steps are:
  - *Mutex library*
  - ***Lock (library)***
  - *BookList = GetList( )* ←Read access
  - *Item = Examine(BookList)*
  - *If available (Item)*
    - *Borrow (Item)* ←Write access
  - ***Unlock (library)***

- How can we protect the critical section?

If we need to
JUST **read** the list…
we will ***not need any lock***
at all!

The example with only reads:

- *Mutex library*
- *bookList = GetList( )*
- *item = Examine(BookList)*
- *Print(item)*

Part One!

- In this part, we will discuss the problems arising when shared memory is used for communication between processes/threads and how they are resolved. Try to find the answer to the following questions:

1. How can arbitrary interleaving cause problems in accessing shared memory?
2. What is a race condition?
3. What is a critical section?
4. What is mutex?
5. What is mutual exclusion? What is starvation?
6. Why should mutex lock/unlock operations be atomic operations?

Part Two!

Part Two!

- In this part, we will discuss the problems arising when message passing is used for communication between processes/threads and how they are resolved. Try to find answers to the following questions:

- Why message passing is more suitable for communication between processes rather than threads?
- When do we use direct naming for channel specification?
- When do we use port naming for channel specification?
- When do we use global naming for channel specification?
- Why should the manager, defined in the message-passing system, use two ports?
- What happens if the read is blocking the ports? What solution exists?

Communication: Message Passing

In some concurrent environments, Shared-memory communication is not very effective.

- For example, the standard programming model of Unix (Linux, etc.) was based on **processes** rather than threads.
- Compared to Windows-based systems, creating and running processes was cheaper

Communication: Message Passing

- The major difference between threads and processes is that processes do not share any state.

- Instead, we can take the other major approach to managing concurrency, called **message-passing**.

Communication: Message Passing

- In Week 2 we discussed general message-passing techniques between processes.
  - Can you name some?

- Threads also can use message-passing techniques.
  - Programmers may need to implement and manage this technique.

Message Passing: How?

- Two primary mechanisms are needed in a message-passing system:
  - A method of creating and specifying communication channels
  - A method of sending and receiving messages

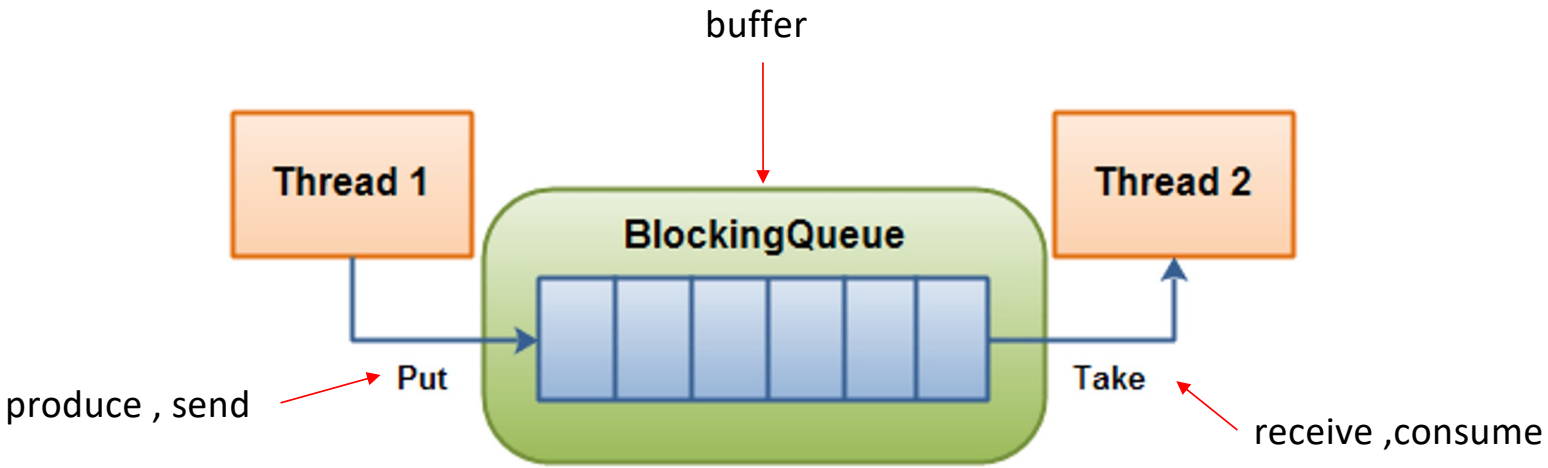
Channel Specification

- *Direct naming:* Sender and receiver directly name each other
  - Simple but less powerful
  - Not suitable for the specification of client-server systems
  - The destination of messages is known in advance.
  - Example: **Pipes**
- *Port naming*
  - The server uses a single port to receive client requests
  - The sender and receiver do not need to know each other in advance
  - Simple, suitable for a single server, multiple client systems
  - Example: **Sockets**
- Global naming
  - Suitable for multiple servers, multiple client systems
  - Example: **Mailbox**

Message Passing: Producer-Consumer

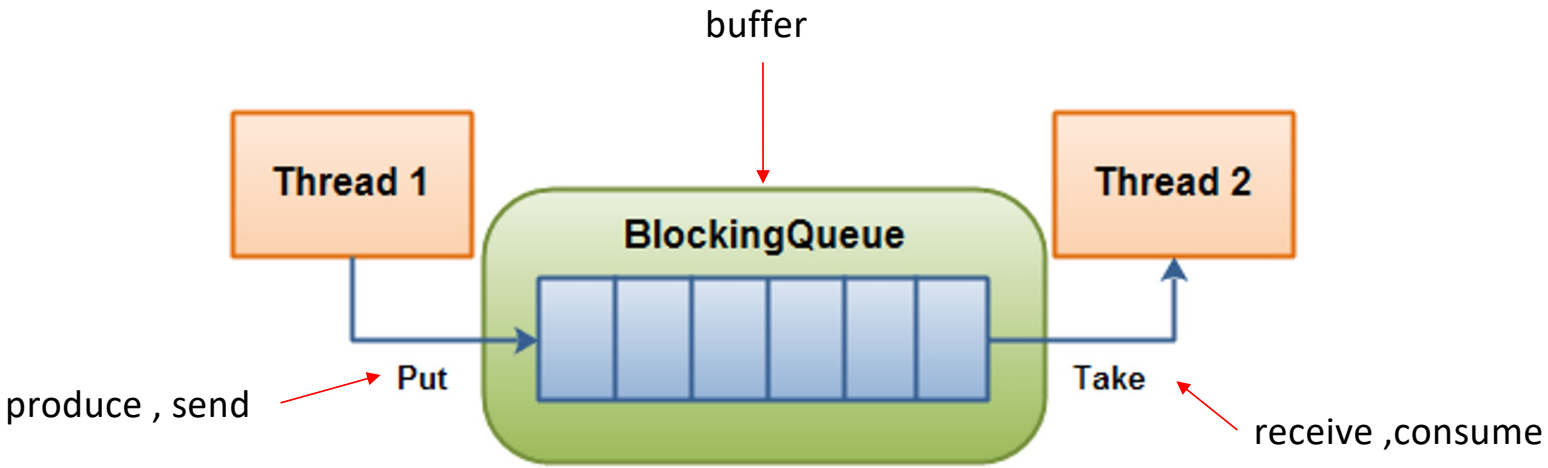
Producer - Consumer: is a well-known model for message passing between threads.

- There is a channel: usually a buffer (a data structure)
- Operations for send/receive



Message Passing: Producer-Consumer

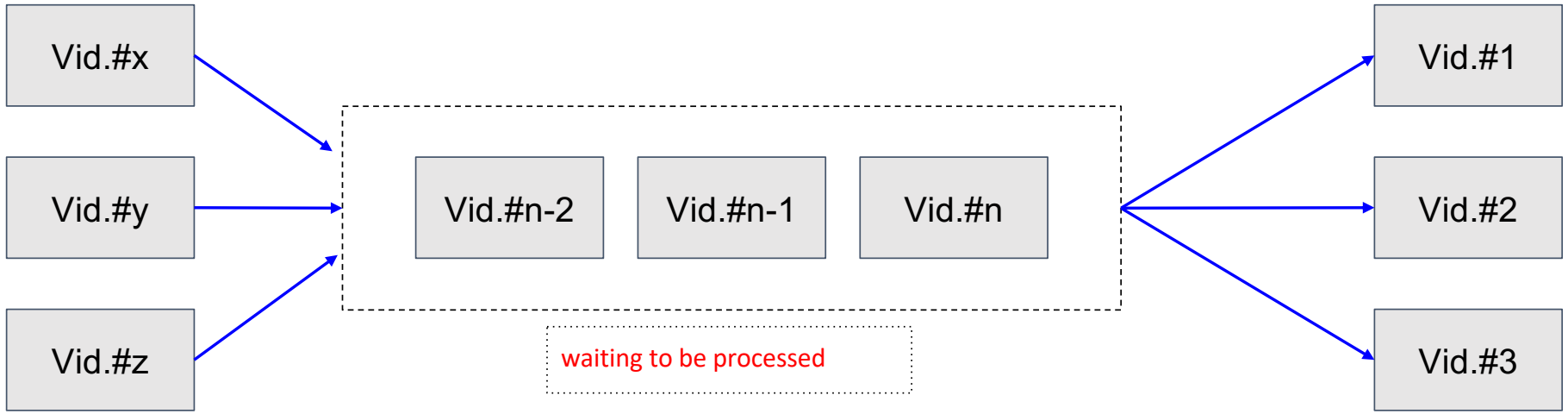
Producer thread inserts (send/put/add) the data in the buffer.
Consumer thread takes (receive/take/remove) the data to use



Producer-Consumer: Example

Transcoding video (what Netflix does every day):

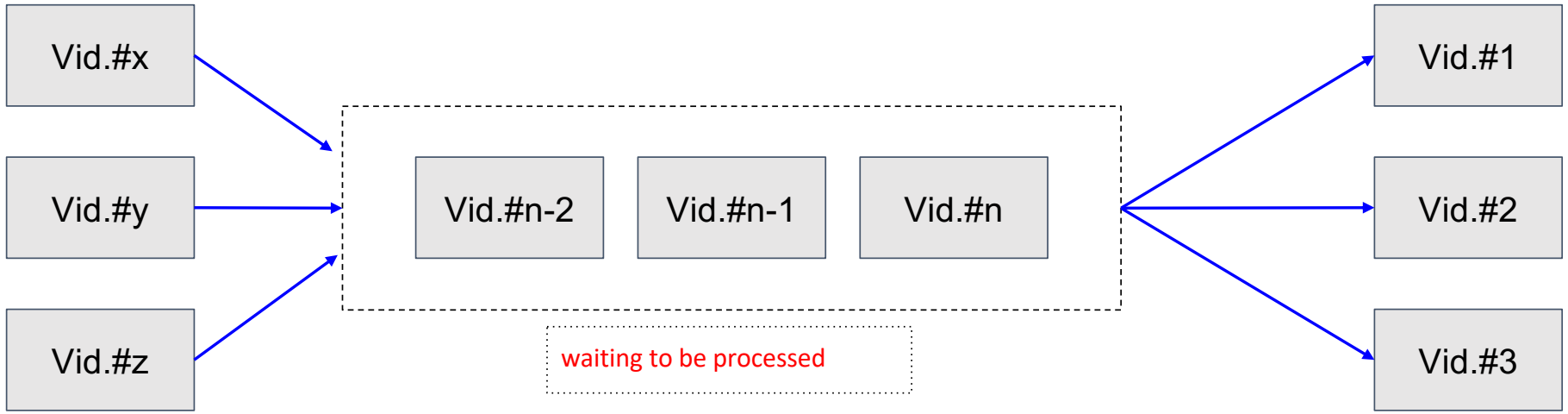
- A set of video to be transcoded in various formats (and resolutions) they are divided into chunks so you can have a complete video encoded faster.
- Functionalities: collecting, reading, transcoding, storing.



Producer-Consumer: Example

Depending on the requirements and specifications, we may consider more than just one solution:

- Multiple-Videos, Multiple-Encoders
- Multiple-Videos, One-Encoder



Producer-Consumer Problem

- Consider multiple producers and multiple consumers
  - Any producer's output can be processed by any consumer

How do producers and consumers communicate?

Solution 1:

- Use port naming
  - Assign a port to each process
- Problem:
  - NOT GOOD: Each producer needs to decide to which consumer the item should be sent … more problems.

(Remember: Any consumer can receive an item produced by any producer)

Producer-Consumer Problem

- Consider multiple producers and multiple consumers
  - Any producer's output can be processed by any consumer

How do producers and consumers communicate?

Solution 2:

- Use a buffer manager
  - Assume only port naming is available
  - A manager manages a shared buffer
  - Similar to a mailbox, but implements a shared queue

Producer-Consumer Problem

The solution based on a buffer manager and port naming poses some challenges:

Challenge 1: **Synchronization**

- How to synchronize the producers and consumers with the bounded buffer
  - *Buffer **full***: Stop putting messages from producers
  - *Buffer **empty***: Stop receiving messages from consumers

Producer-Consumer Problem

Challenge 2: **Managing the buffer**

- The manager needs two ports, one for consumers, and one for producers (why?)
  - Otherwise, it cannot block only consumers or only producers
  (when the buffer is full producers are blocked but consumers are not, and when the buffer is empty consumers are blocked but producers are not)
  - When the buffer is neither full nor empty, the manager should accept request from both ports:
    - Q: How can the manger read from two ports?

Producer-Consumer Problem

- Solution 1: Use **blocking** *receive()* and two threads
  - The manager cannot be blocked on both ports. Why?

- Solution 2: Use **non-blocking** *receive()*
  - Use blocking send, and non-blocking receive *(BSNR)*
  - Poll two ports

Producer-Consumer Problem

- Manager process ($P_{manager}$)
  - *Req_port*: manager receives requests from consumers
  - *Data_port*: manager receives data items from producers
  - BSNR
    - *Receive* returns immediately, it returns *true* if a message is ready in the port, otherwise, *false* *(polling)*
    - *Send* only returns after the message is delivered
  - *count*: # data items in $P_{manager}$'s buffer (updated by the manager)

Producer-Consumer: A Sample algorithm

Producer Process:

```
repeat
  produce item;
  send (dataport, item);
until false;
```

Consumer Process:

```
repeat
  send (reqport, localport);
  while not receive (localport,
     item) do nothing;
  consume item;
until false;
```

Producer Process: is light blue
Consumer Process: is dark red
Manager Process: is green

Producer-Consumer Problem

Manager Process (blocking send, and non-blocking receive, BSNR):

```
repeat forever
  if count = 0 then
  {  while not receive (dataport, item) do nothing; //receive from producer
     put item in the queue;  count := count + 1;
  }
  else if count = N then
  {  while not receive (reqport, consumer_port) do nothing; //receive request from consumer
     take item from queue;  count := count - 1;
     send (consumer_port, item);
  }
  else
  {
        if receive (dataport, item) then
        {  put item in the queue;  count := count + 1; }
        if receive (reqport, consumer_port) then
        {  take item from queue;  count := count - 1;
             send (consumer_port, item);
        }
  }
```

Producer Process: is light blue
Consumer Process: is dark red
Manager Process: is green

Part Two!

- In this part, we will discuss the problems arising when message passing is used for communication between processes/threads and how they are resolved. Try to find answers to the following questions:

- Why message passing is more suitable for communication between processes rather than threads?
- When do we use direct naming for channel specification?
- When do we use port naming for channel specification?
- When do we use global naming for channel specification?
- Why should the manager, defined in the message-passing system, use two ports?
- What happens if the read is blocking the ports? What solution exists?

Quiz?

Yes, now...

Time to apply