DCS216 Operating Systems

Lecture 07
Inter-process Communication (1)

Mar 18th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

Content

Content

- **Cooperating Processes**
  - Processes within a system can be **independent** or **cooperating**.
    - **Independent:** An independent process (独立进程) does not share data with any other processes executing in the system.
    - **Cooperating:** A cooperating process (协作进程) can affect or be affected by other processes executing in the system.

Inter-process Communication

Cooperating Processes

- Processes within a system can be **independent** or **cooperating**.
  - **Independent:** An independent process (独立进程) does not share data with any other processes executing in the system.
  - **Cooperating:** A cooperating process (协作进程) can affect or be affected by other processes executing in the system.
- Reasons for cooperating processes:
  - Information Sharing
  - Computation Speedup
  - Modularity
  - Convenience

Cooperating Processes

- Processes within a system can be **independent** or **cooperating**.
  - **Independent:** An independent process (独立进程) does not share data with any other processes executing in the system.
  - **Cooperating:** A cooperating process (协作进程) can affect or be affected by other processes executing in the system.
- Reasons for cooperating processes:
  - **Information Sharing**
    - concurrent access to information by several applications
  - **Computation Speedup**
    - break into subtasks, execute in parallel to speed up computation
  - **Modularity**
    - construct the system in a modular fashion, dividing the system functions into separate processes or threads
  - **Convenience**
    - Users may be multitasking, e.g., a programmer might be editing source files, listening to music and compiling at the same time.

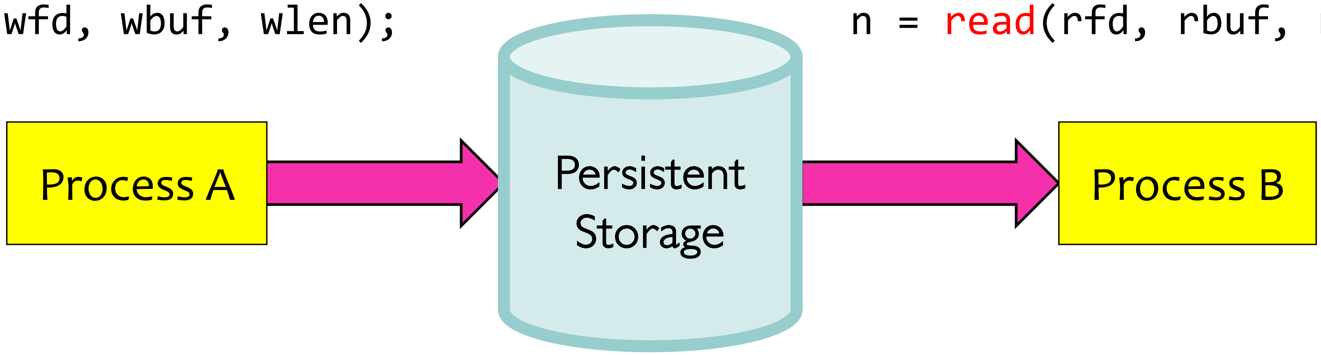
Inter-process Communication

Communication Between Processes

- The simple and naive approach: **Use a FILE!**

```
write(wfd, wbuf, wlen);
```

```
n = read(rfd, rbuf, rmax);
```



L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- Downsides:
  - Very expensive (disk I/O requests)
  - Most communications do not require persistent storage on disk

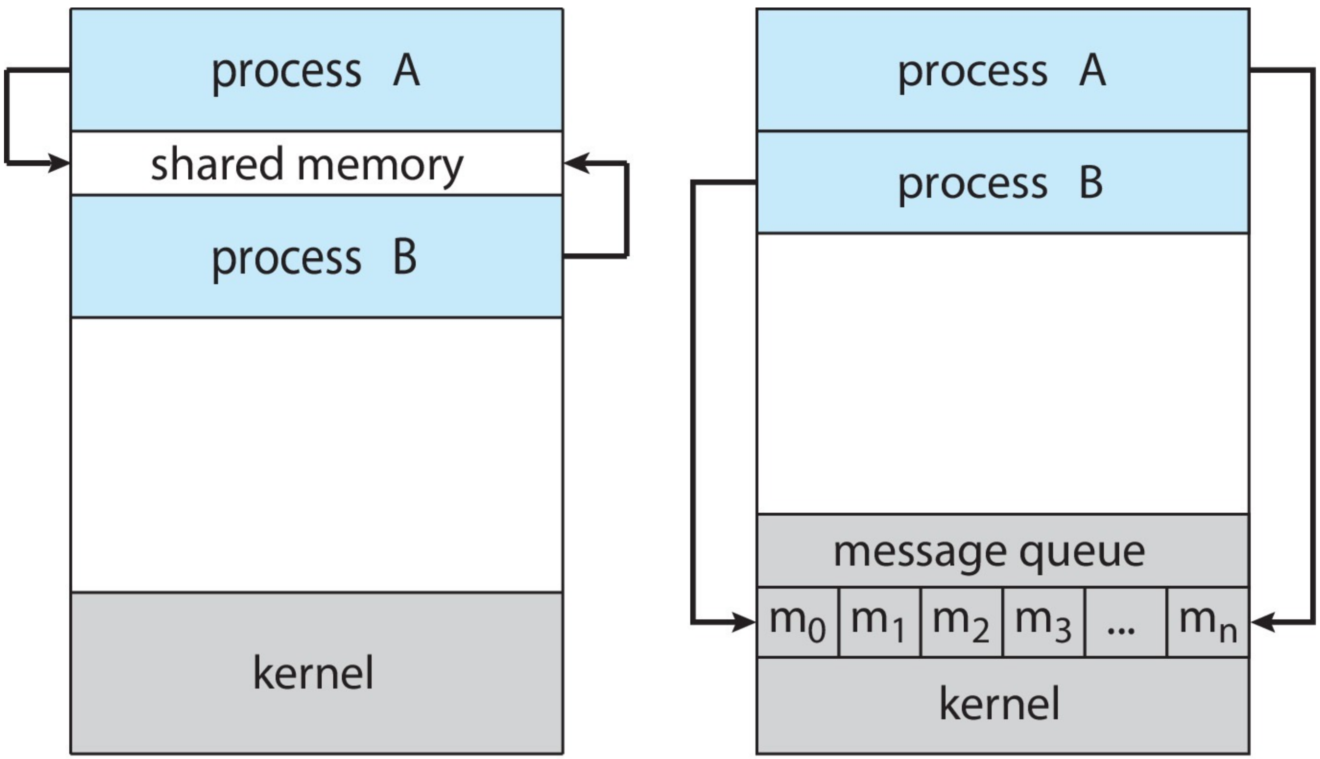
Cooperating Processes

- Processes within a system can be **independent** or **cooperating.**
  - **Independent:** An independent process (独立进程) does not share data with any other processes executing in the system.
  - **Cooperating:** A cooperating process (协作进程) can affect or be affected by other processes executing in the system.
- Reasons for cooperating processes:
  - Information Sharing
  - Computation Speedup
  - Modularity
  - Convenience
- Cooperating processes need **inter-process communication** (**IPC**)
- Two fundamental models or mechanisms of IPC:
  - **Shared Memory**
  - **Message Passing**

IPC Models

- Two fundamental models of IPC

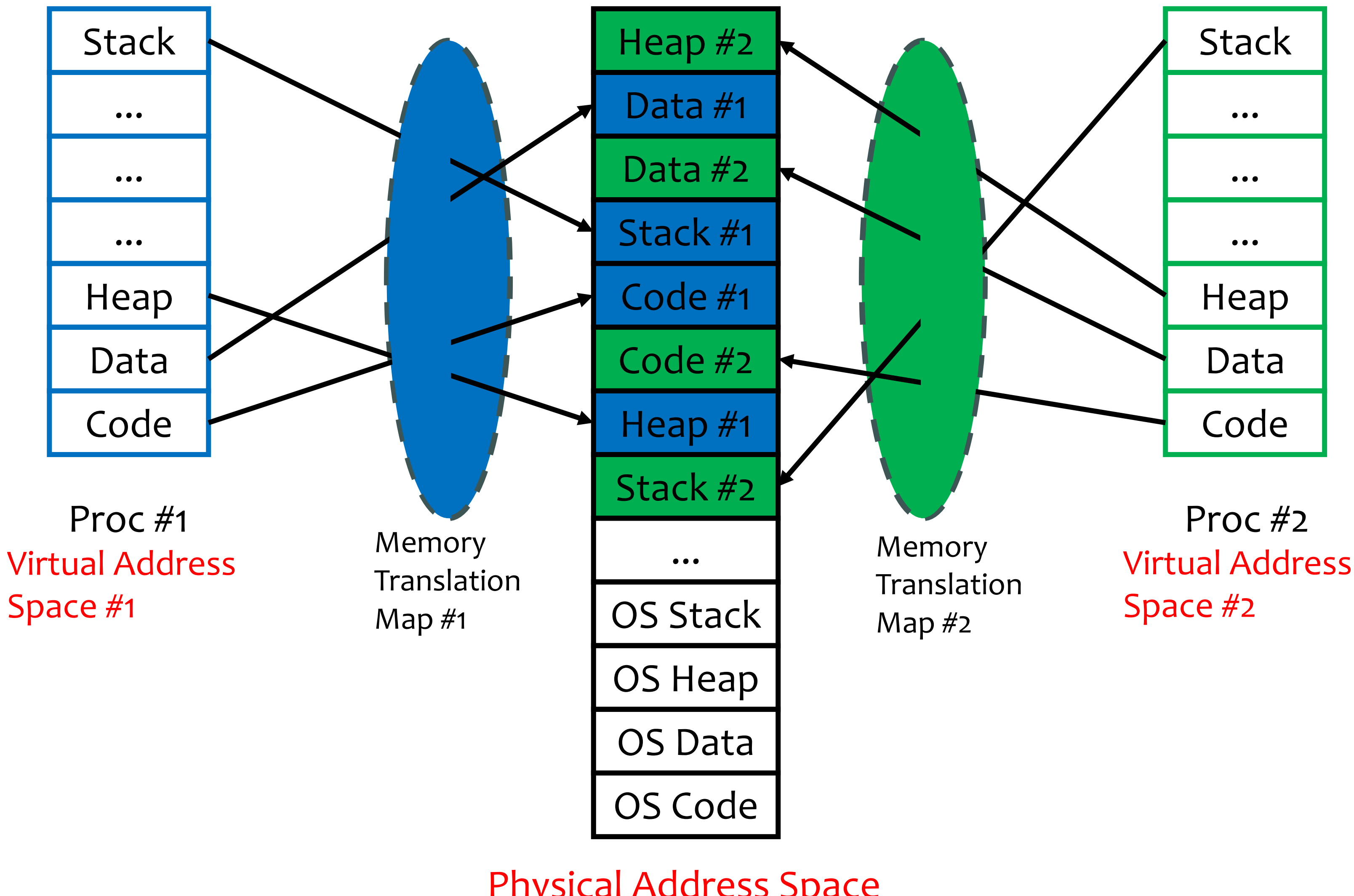
(a) Shared Memory (b) Message Passing



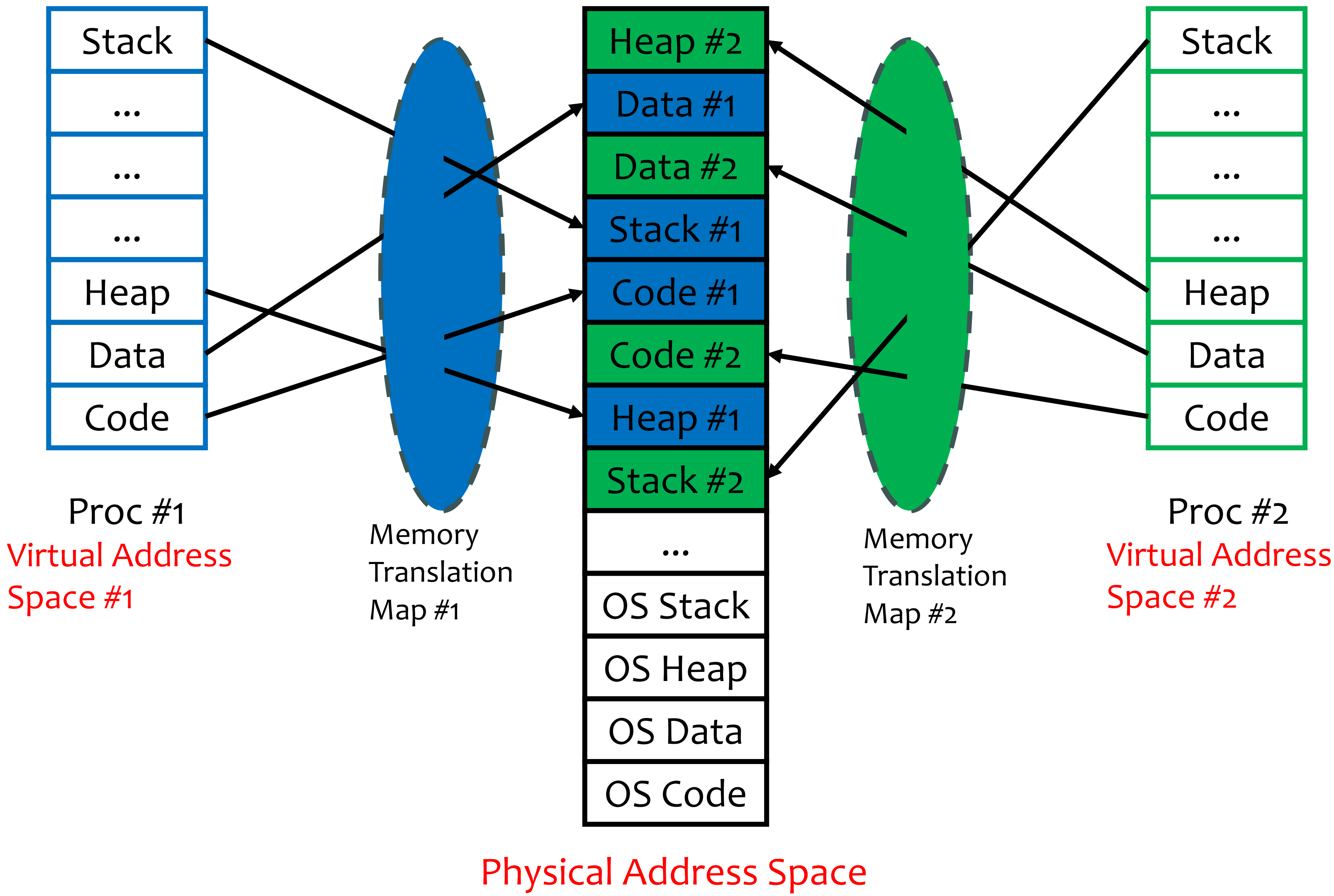
(a) (b)

(Recall) Processes Protected from Each Other

(Recall) Processes Protected from Each Other



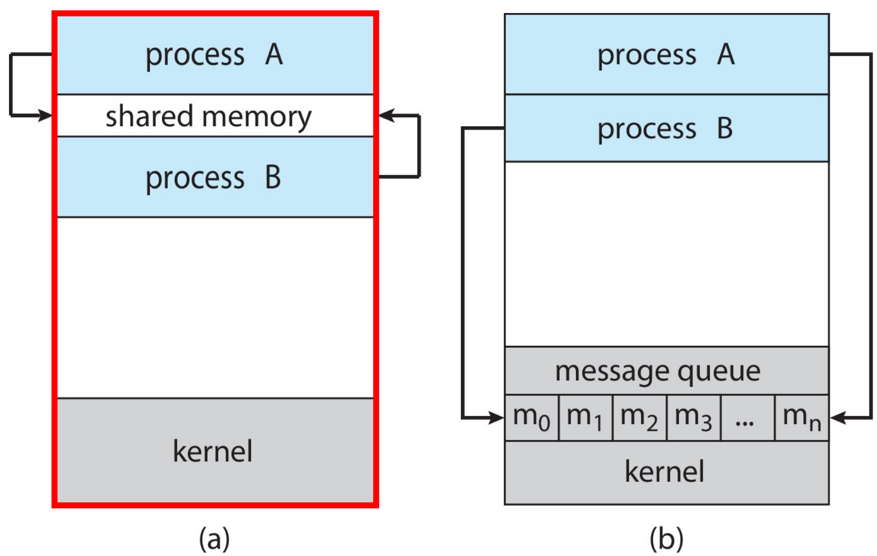
(Recall) Processes Protected from Each Other



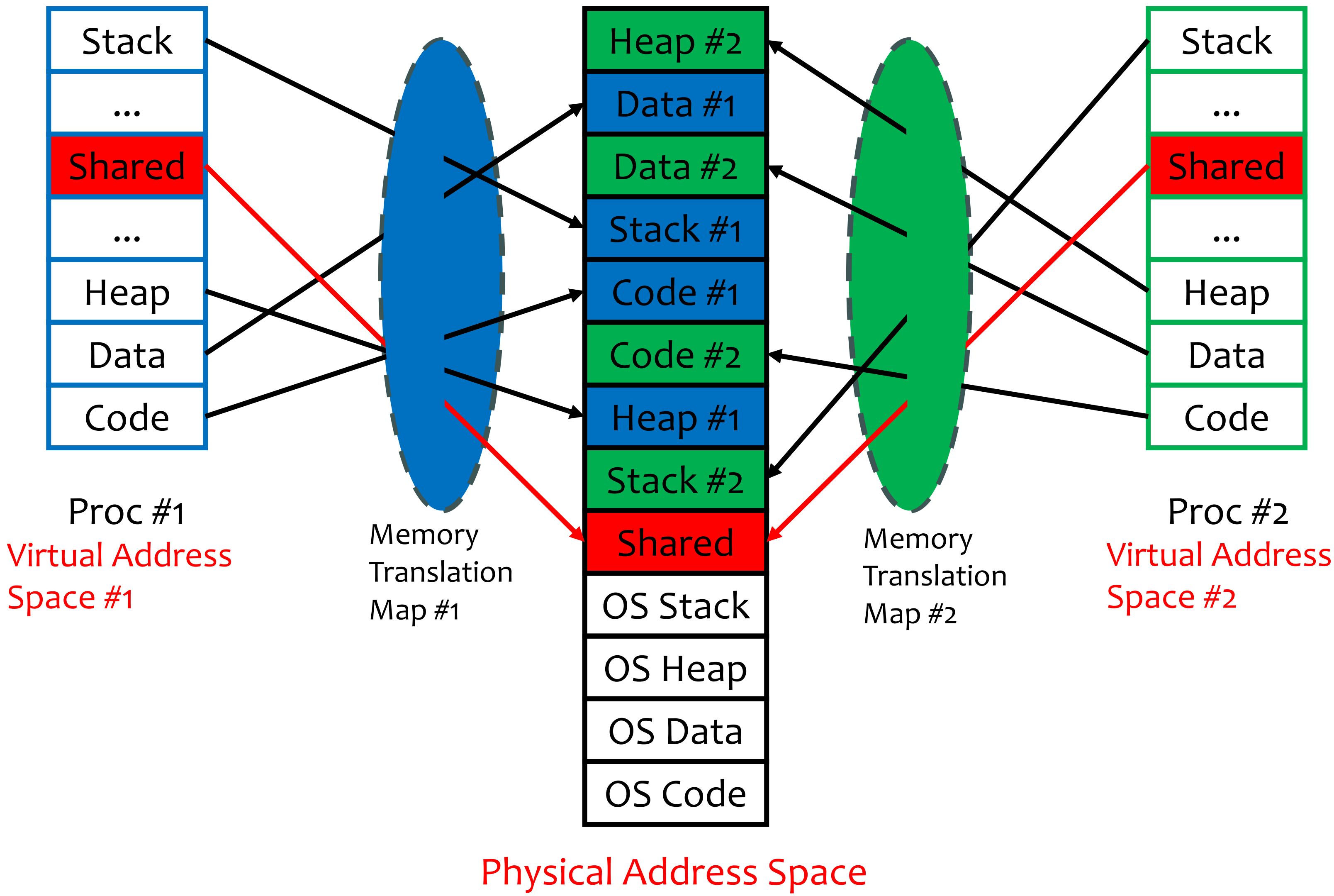
IPC Models

Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes, not the operating system.
- System calls are **required only to establish** shared memory
- Once established, all accesses to the shared memory are treated as routine memory accesses, no assistance from the kernel is required



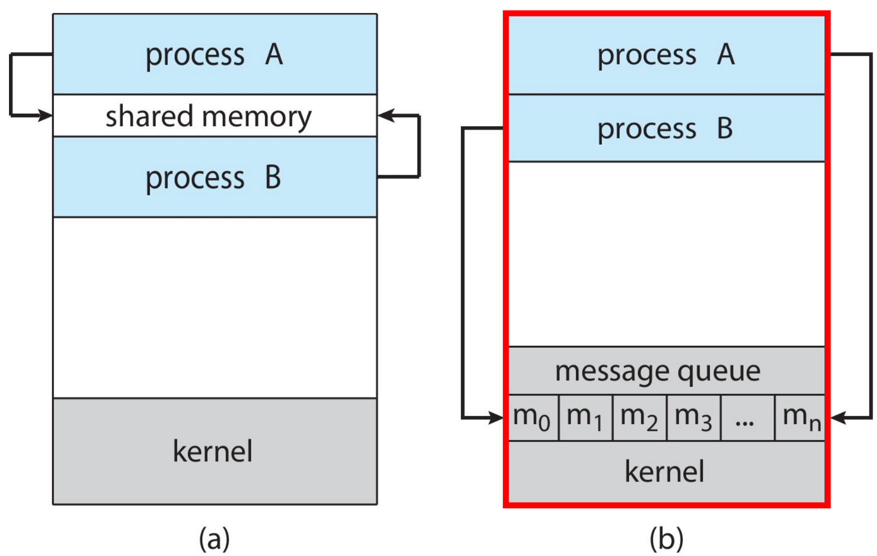
Shared Memory



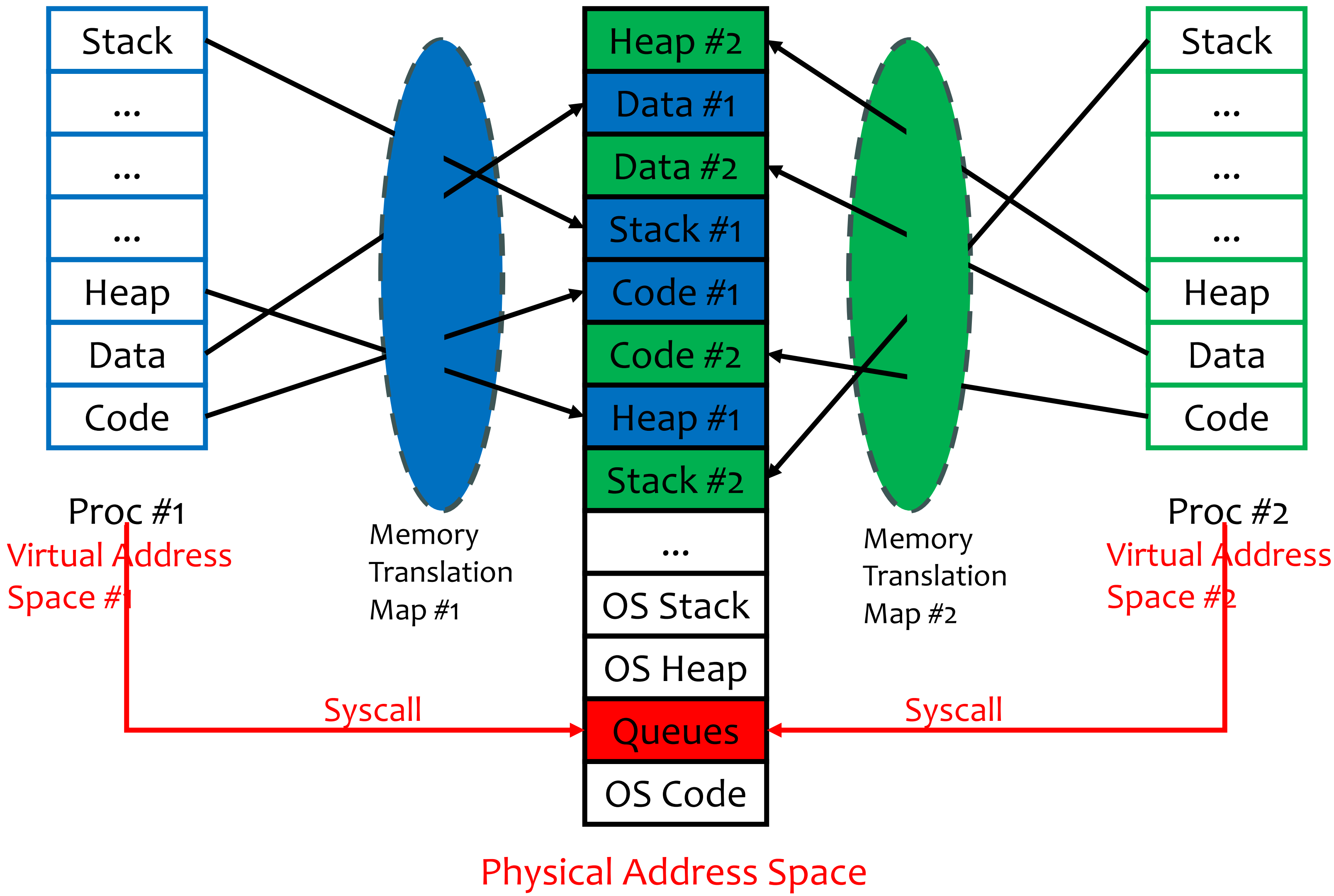
IPC Models

Message Passing

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- Message passing provides (at least) two basic operations:
  - **send**(message)
  - **receive**(message)
- Message size can be either fixed or variable



Message Passing



Shared Memory vs. Message Passing

	Shared Memory	Message Passing
Transfer Data Amount	Large	Small
Implementation Difficulty	More Difficult	Easier
Communication Speed	Faster	Slower (via Syscalls)
Synchronization	Explicit (required)	Implicit (not required)
Flexibility	Less Flexible	More Flexible
Security	Less Secure	More Secure

Inter-process Communication

- **IPC Models**
  - **Shared Memory**
    - Direct Sharing (**System V Standard**) System Calls:
      - `shmget()`, `shmat()`, `shmdt()`, `shmctl()`
    - Indirect Sharing (**POSIX Standard**) Library Calls:
      - `shm_open()`, `shm_unlink()`, `ftruncate()`, `mmap()`
  - **Message Passing**
    - **Pipes**
      - Unnamed Pipe: `pipe()`
      - Named Pipe: `mkfifo()`
    - **Message Queues**
      - System V: `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`
      - POSIX: `mq_open()`, `mq_close()`, `mq_send()`, `mq_receive()`
    - **Sockets**: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`
    - **Signals**: `signal()`, `sigaction()`

Shared Memory

- IPC using shared memory requires communicating processes to establish a region of shared memory
- Typically, a shared-memory region resides in the address space of the process **creating** the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must **attach** it to their address space.
  - OS prevents one process from accessing another process's memory
  - It requires that two processes agree to remove such restriction
- Two processes can then **exchange** information by reading or writing data in the shared area of memory region.
- The processes are **responsible** for ensuring that they are not writing to the same location **simultaneously.**
  - E.g., mutex, semaphore...
  - Topics for "**Synchronization**", more on this in upcoming lectures.

Producer-Consumer Problem with Shared Memory

- The producer-consumer problem (生产者-消费者问题) is a common paradigm for cooperating processes.
  - A producer **produces** information that is **consumed** by a consumer.
  - A buffer **shared** by these two processes is designed to be filled by the producer and emptied by the consumer
  - The producer and the consumer are running concurrently and must be **synchronized**, so that the consumer does not try to consume an item that has not yet been produced
- Two types of buffers can be used
  - **Unbounded** Buffer
    - with no practical limit on the size of the buffer
    - consumer may have to wait for new items, producer can always produce new item
  - **Bounded** Buffer
    - assumes a fixed buffer size
    - consumer must wait if **empty**; producer must wait if **full**

Producer-Consumer Problem with Shared Memory

- Shared data

```
#define BUFFER_SIZE 10

typedef struct {
    ...            /* item structure */
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`.
  - The variable `in` points to the next free position in the buffer;
  - The variable `out` points to the first full position in the buffer
  - The buffer is empty when `in == out`
  - The buffer is full when `((in + 1) % BUFFER_SIZE) == out`
  - This scheme allows at most `BUFFER_SIZE-1` items in the buffer at the same time. **WHY**?

Producer-Consumer Problem with Shared Memory

- Producer:

```
item next_produced;
while (true) {

    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer:

```
item next_consumed;
while (true) {
    while (in == out)
        ;      /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

Producer-Consumer Problem with Shared Memory

- Producer vs. Consumer

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

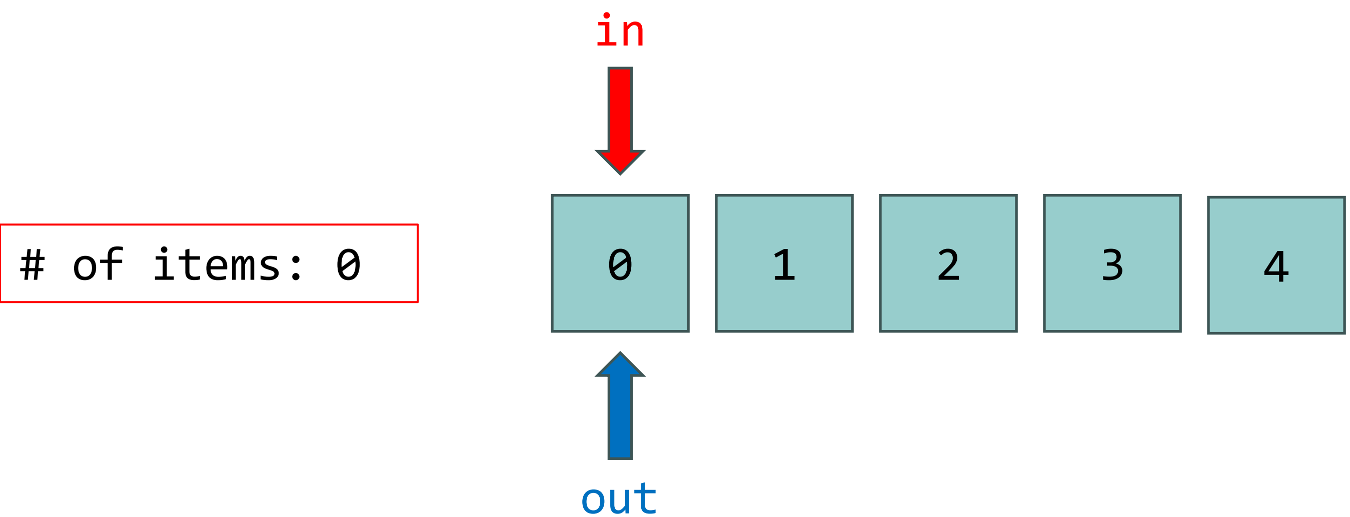
```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

- The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`.
  - The variable `in` points to the next free position in the buffer;
  - The variable `out` points to the first full position in the buffer
  - The buffer is empty when `in == out`
  - The buffer is full when `((in + 1) % BUFFER_SIZE) == out`
  - This scheme allows at most `BUFFER_SIZE-1` items in the buffer at the same time. **WHY**?

Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

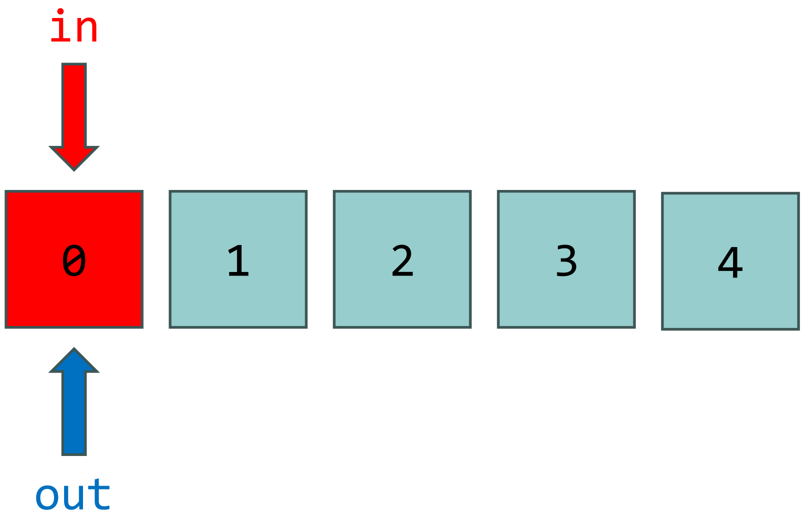


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 1

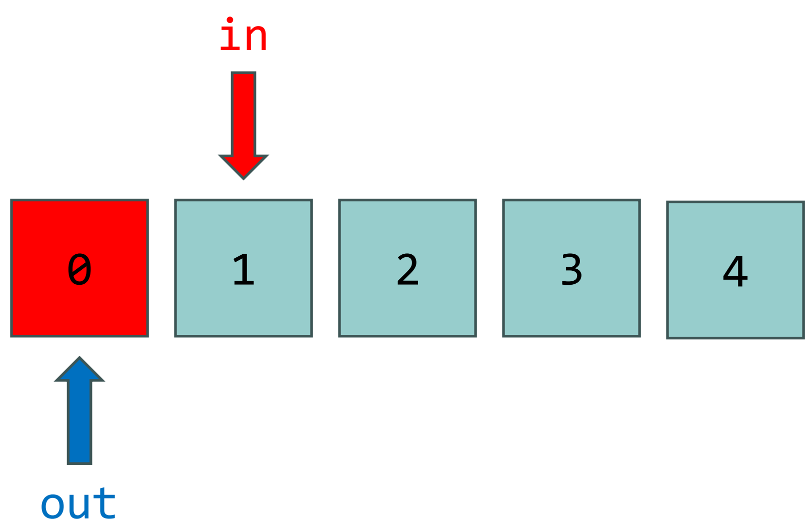


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 1

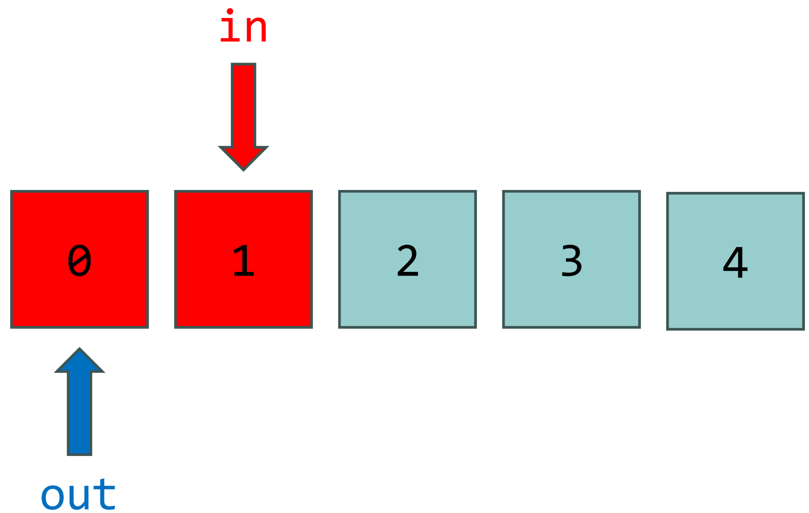


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 2

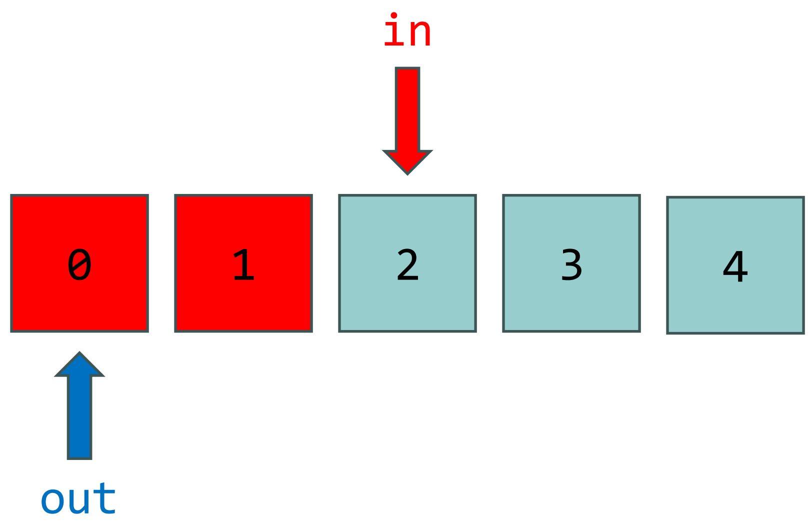


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

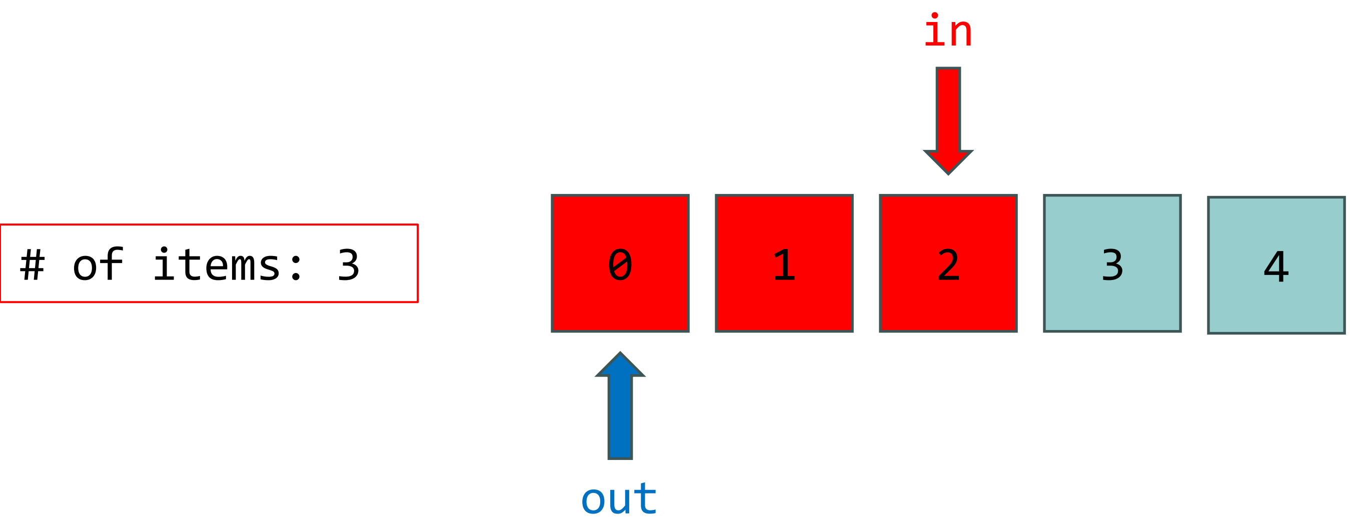
# of items: 2



Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```



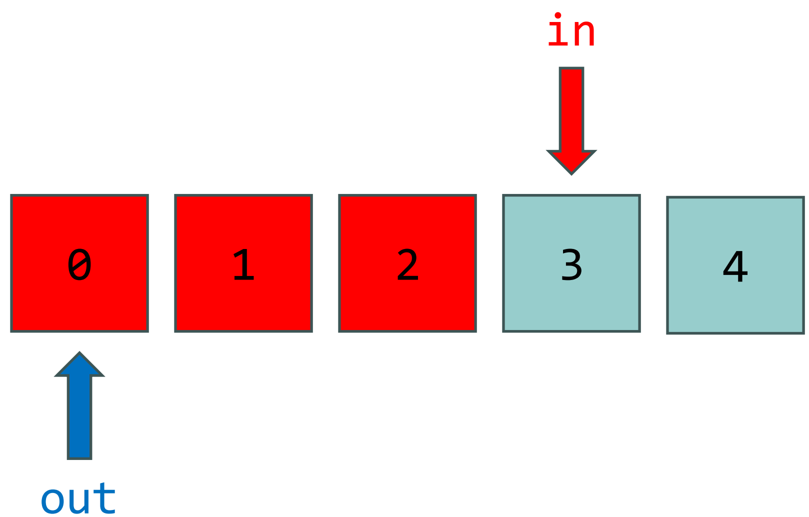
Share-Memory Systems

Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 3

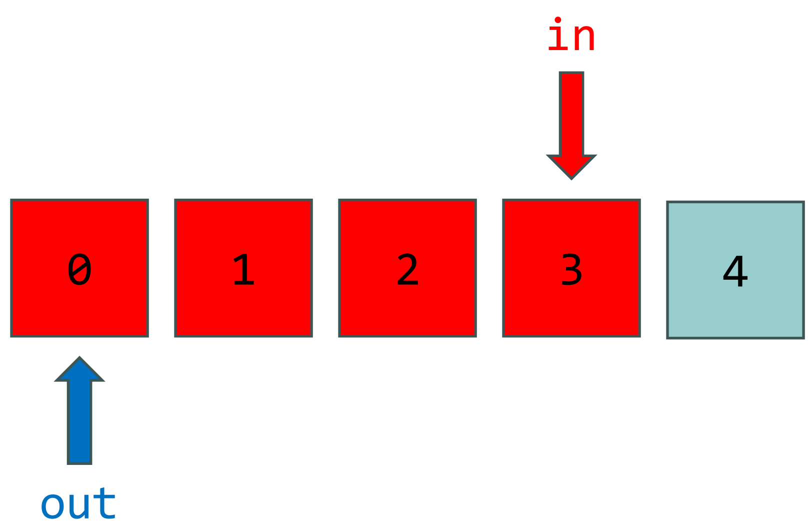


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 4

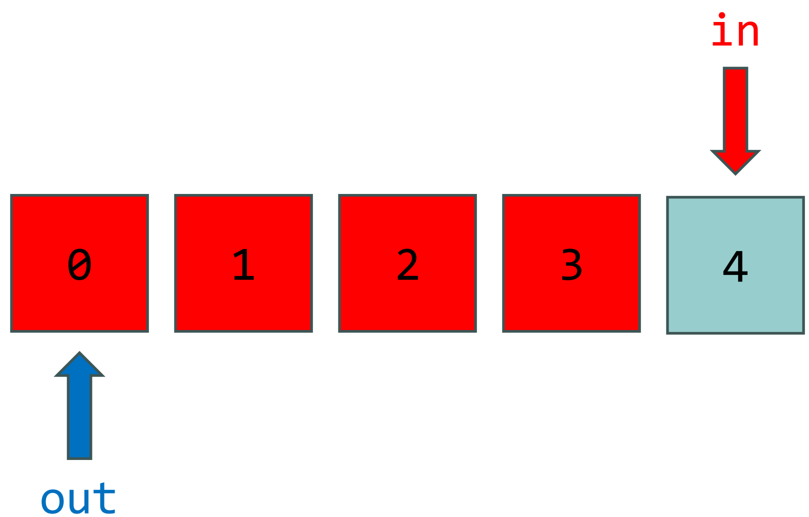


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 4

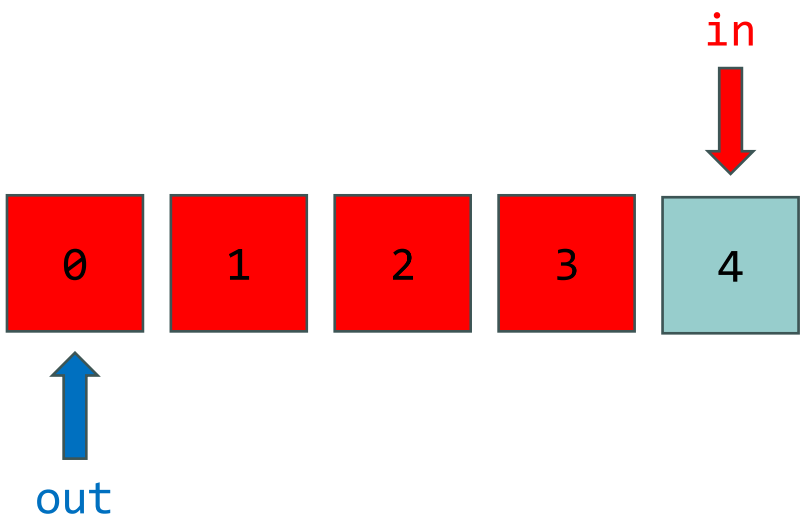


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 4



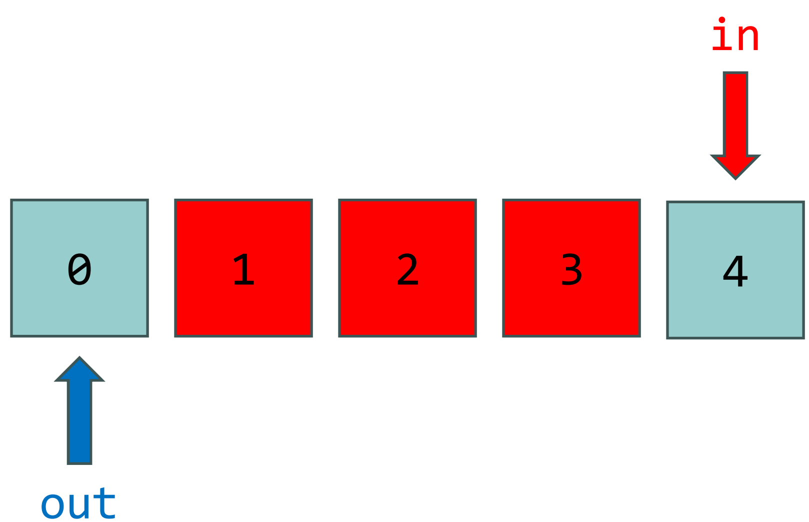
Full!

Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;        /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 3

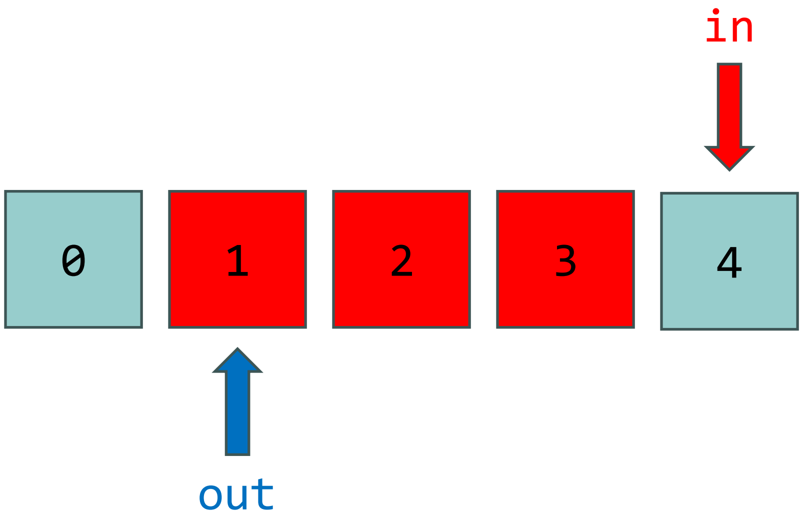


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;        /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 3

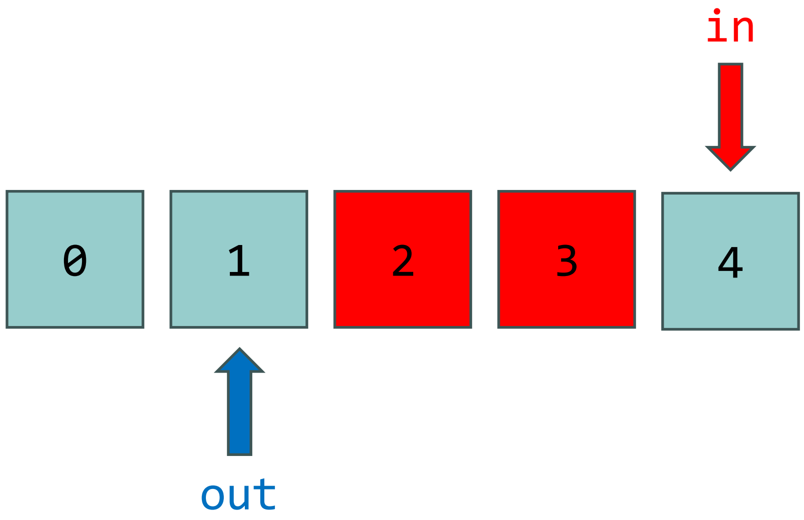


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 2

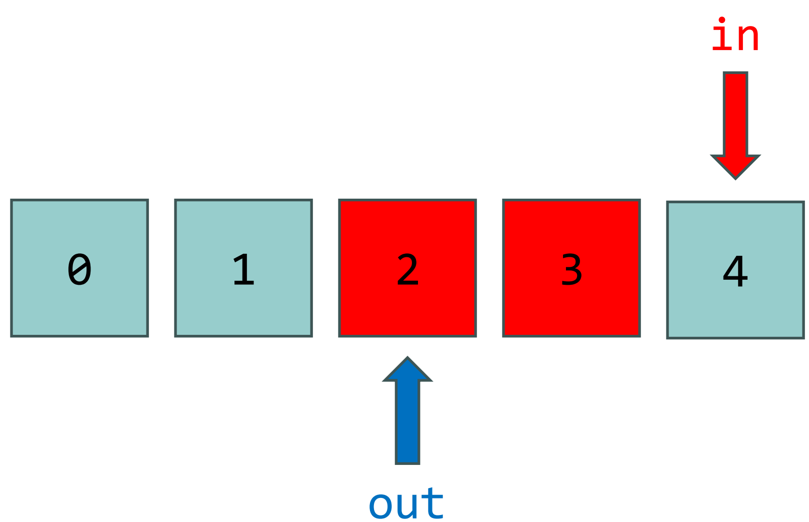


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 2

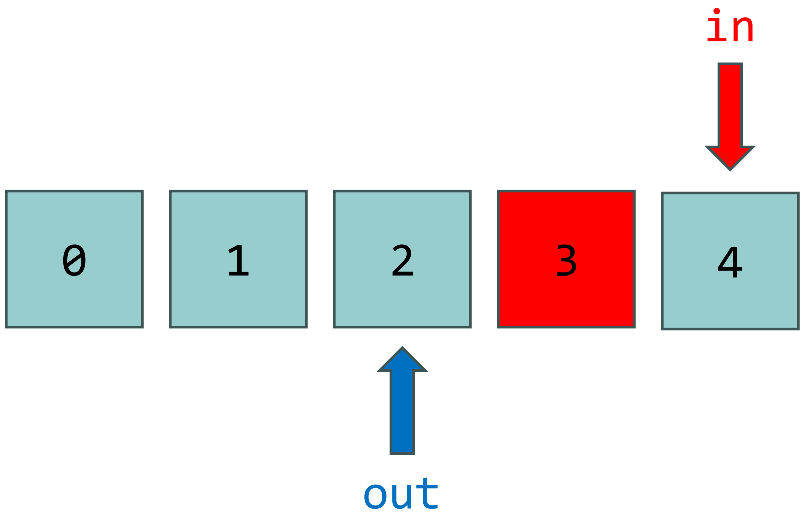


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

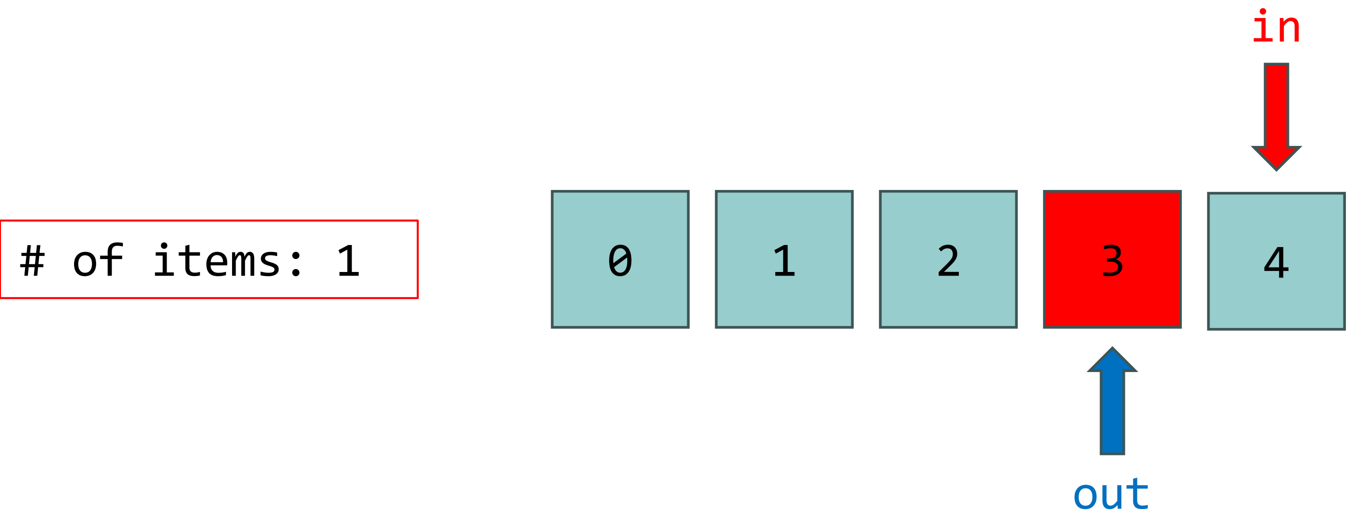
# of items: 1



Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

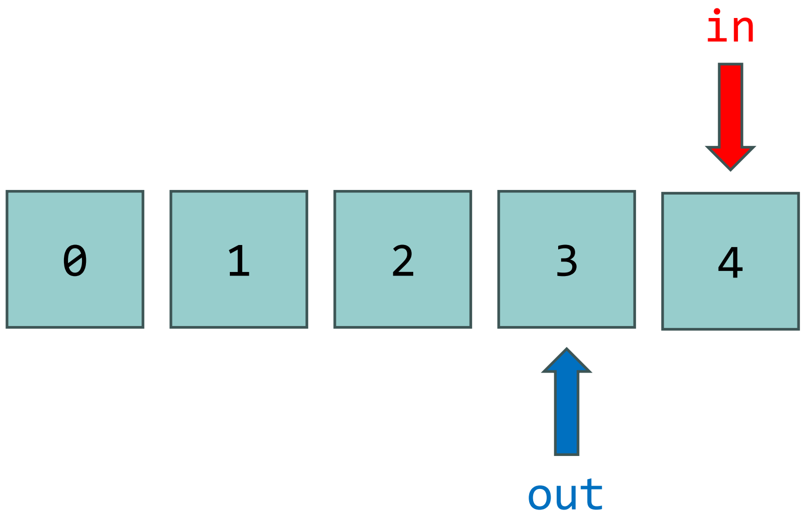


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 0

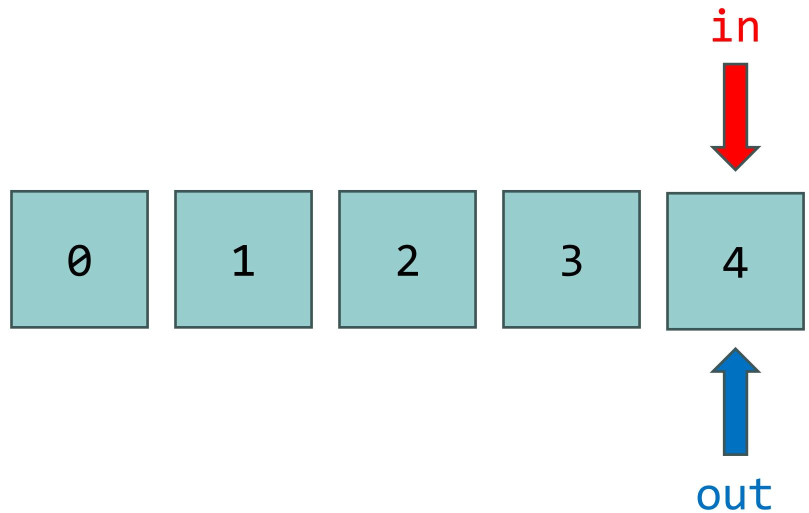


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;        /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

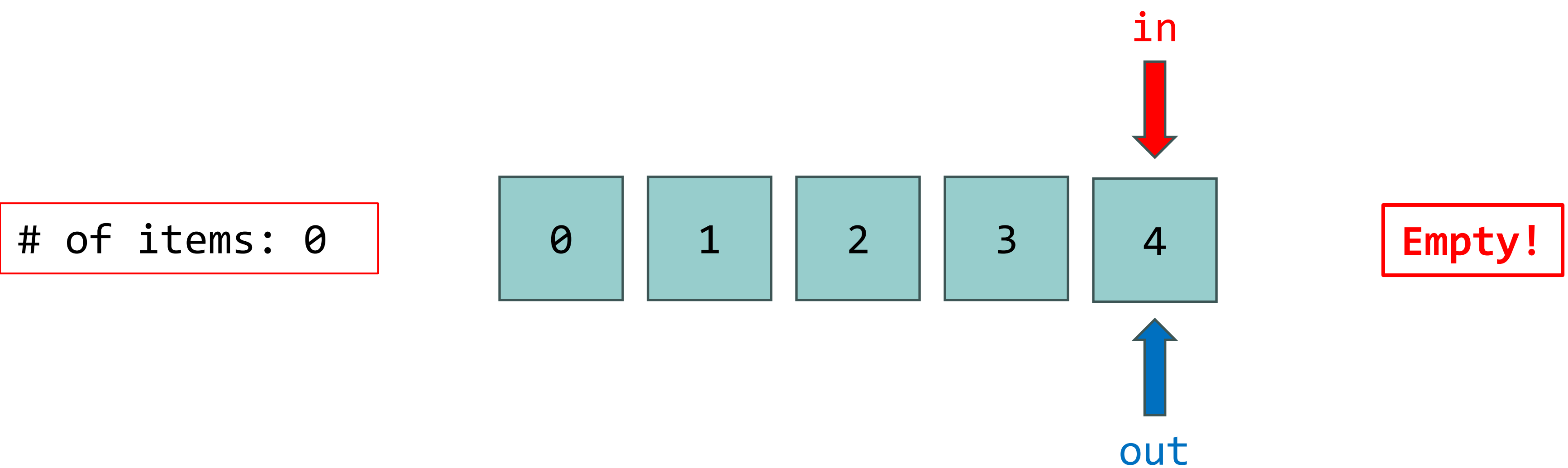
# of items: 0



Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

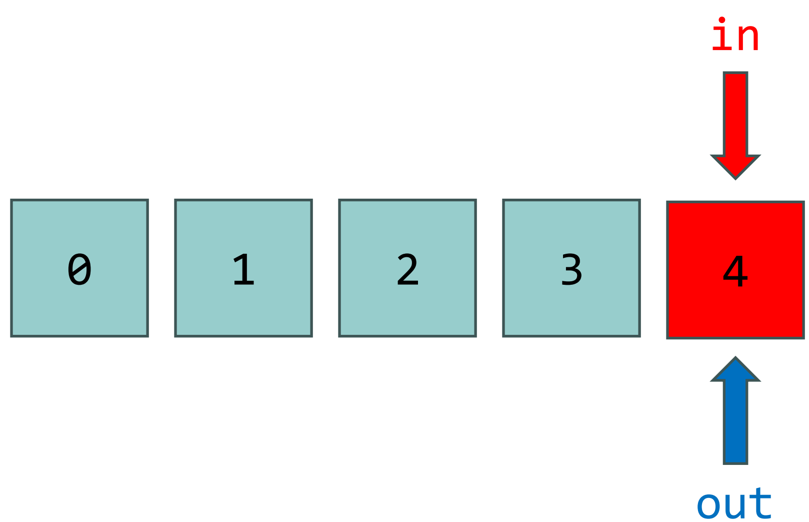


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 1

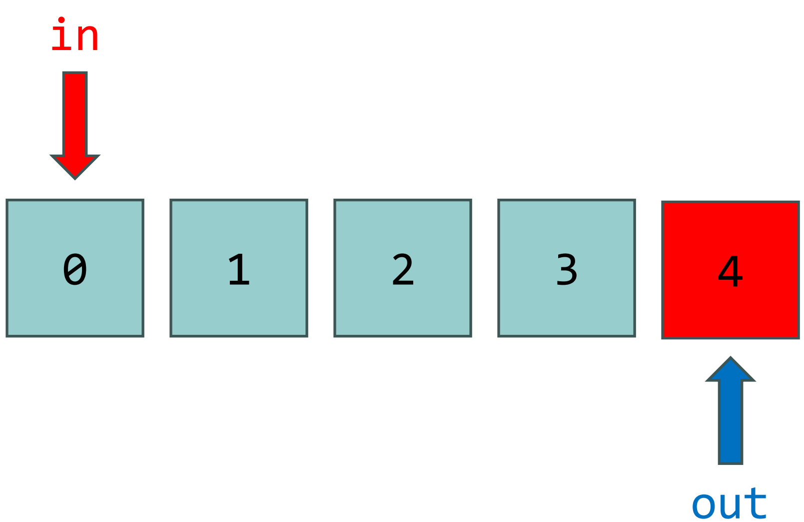


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;      /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 1

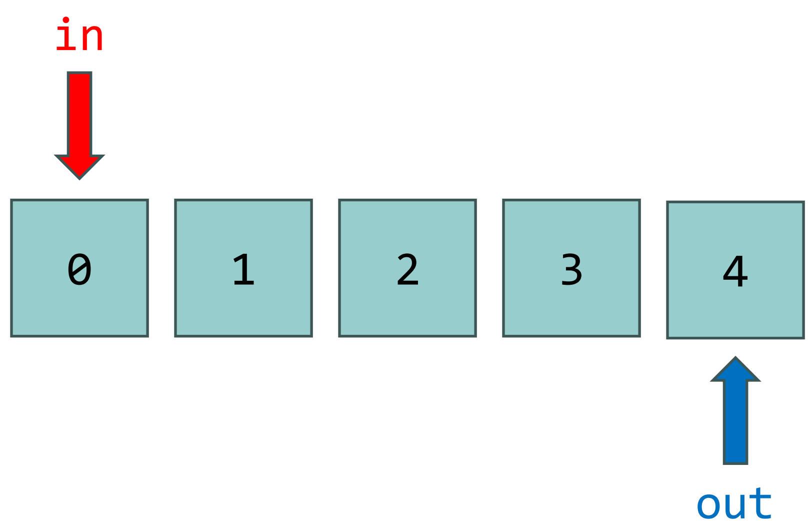


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 0

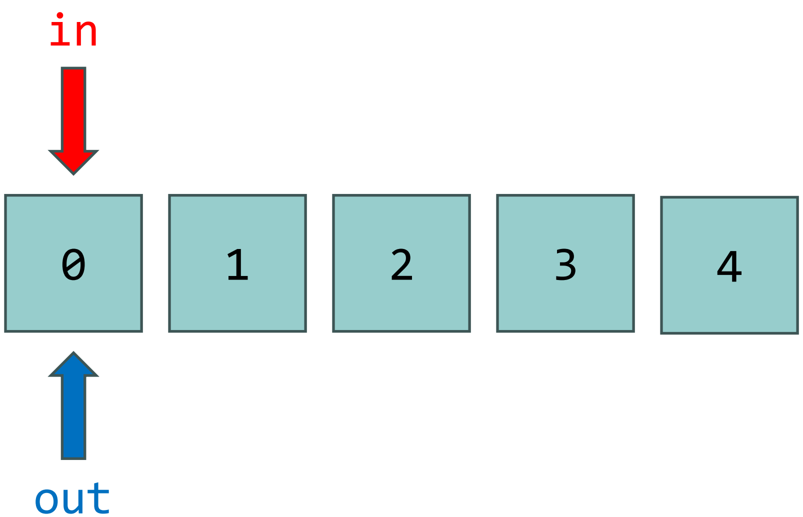


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 0

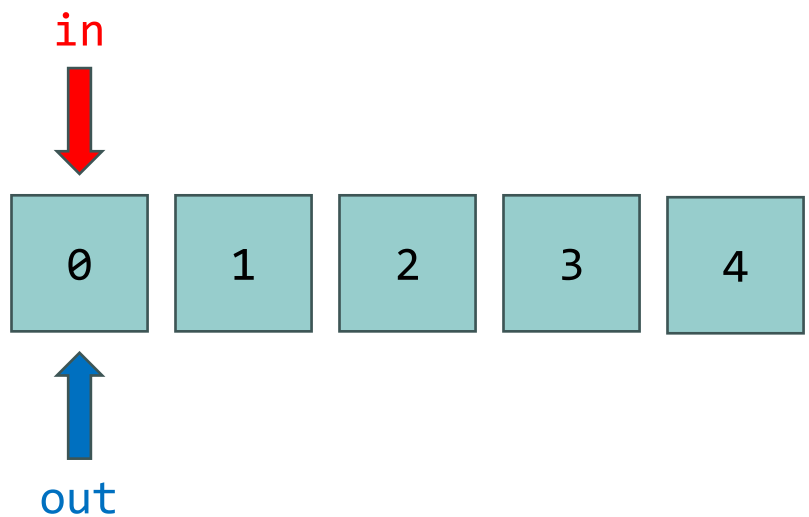


Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 0



Empty!

- **Shared Memory API**
  - **Direct Sharing**: System V **low-level** system calls, where processes directly manipulate shared memory segments.
    - `shmget()`: Allocate(Get) a shared memory segment
    - `shmat()`: Attaches segment to the process's address space
    - `shmdt()`: Detaches segment from the process's address space
    - `shmctl()`: Perform control operations on the shared memory segment, such as marking it for deletion
  - **Indirect Sharing**: POSIX **high-level** library calls (wrappers of system calls) that use file descriptors (**fd**) to handle shared memory objects, which can be mapped into the address space using `mmap()`.
    - `shm_open()`: Opens a POSIX shared memory object
    - `shm_unlink()`: Removes(Unlink) the shared memory object
    - `ftruncate()`: Set the size of the shared memory object
    - `mmap()`: Maps the shared memory object into process's address space
    - `munmap()`: Unmaps the share memory object from process's address space

Shared Memory API

- **Direct Sharing:** System V **low-level** system calls, where processes directly manipulate shared memory segments.
  - `shmget()`: Allocate(Get) a shared memory segment
  - `shmat()`: Attaches segment to the process's address space
  - `shmdt()`: Detaches segment from the process's address space
  - `shmctl()`: Perform control operations on the shared memory segment, such as marking it for deletion
- **Indirect Sharing:** POSIX **high-level** library calls (wrappers of system calls) that use file descriptors (**fd**) to handle shared memory objects, which can be mapped into the address space using `mmap()`.
  - `shm_open()`: Opens a POSIX shared memory object
  - `shm_unlink()`: Removes(Unlink) the shared memory object
  - `ftruncate()`: Set the size of the shared memory object
  - `mmap()`: Maps the shared memory object into process's address space
  - `munmap()`: Unmaps the share memory object from process's address space

POSIX Shared Memory

- POSIX Shared Memory is organized using **memory-mapped files,** which associates the region of shared memory with a file (typically in `/dev/shm/` under Linux systems, basically a **file system mounted on physical memory**, instead of hard disks).
- For memory sharing, a process must first create a shared-memory object using the `shm_open()` function call
  - Prototype (see `man 3 shm_open`):

```
int shm_open(const char *name, int oflag, mode_t mode);
```

    - `name`: specifies the name of the shared memory object
    - `oflag`: is a bit mask created by `OR`ing together `O_RDONLY`, `O_RDWR`, `O_CREAT`, etc.
    - `mode`: **file permission**, e.g., 0666, 0755, 0700
    - `RETURN VALUE`: A successful call to `shm_open()` returns an integer **file descriptor** for the shared-memory object
  - Example: `fd = shm_open(name, O_CREAT | O_RDWR, 0666);`

POSIX Shared Memory

- Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes.
  - Prototype (See `man 2 truncate`)

    ```
    int ftruncate(int fd, off_t length);
    ```
  - Example: `ftruncate(fd, 4096);` sets the size of the shared memory object to 4096 bytes
- Finally, the `mmap()` function establishes a memory-mapped file containing the shared object. It also returns a pointer to the memory-mapped file that is used for accessing the shared memory object.
  - Prototype (See `man 2 mmap`)

    ```
    void *mmap(void *addr, size_t length, int prot, int
               flags, int fd, off_t offset);
    ```
  - Examples:

    ```
    shmptr = mmap(0, size, PROT_READ, MAP_SHARED, fd, 0);
    shmptr = mmap(0, size, PROT_READ|PROT_WRITE,
                  MAP_SHARED, fd, 0);
    ```

POSIX Shared Memory

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "SYSUOS";
    /* strings written to shared_memory */
    const char *msg = "Hello, World!\n";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char *ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                  PROT_WRITE, MAP_SHARED, fd, 0);

    printf("ptr addr: %p\n", ptr);

    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);

    ptr += strlen(msg);

    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "SYSUOS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char *ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    if (fd == -1) {
        fprintf(stderr, "shm_open() failed.\n");
        exit(1);
    }
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                  MAP_SHARED, fd, 0);

    printf("ptr addr: %p\n", ptr);

    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);

    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "SYSUOS";
    /* strings written to shared_memory */
    const char *msg = "Hello, World!\n";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char *ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                  PROT_WRITE, MAP_SHARED, fd, 0);

    printf("ptr addr: %p\n", ptr);

    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);

    ptr += strlen(msg);

    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "SYSUOS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char *ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    if (fd == -1) {
        fprintf(stderr, "shm_open() failed.\n");
        exit(1);
    }
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                  MAP_SHARED, fd, 0);

    printf("ptr addr: %p\n", ptr);

    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);

    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
              PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
              MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

POSIX Shared Memory Example

```
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
              PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
              MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000
```

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                  PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                  MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

shm_producer has successfully created a shared memory object named SYSUOS in /dev/shm/.

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                       PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                       MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS

# Running Consumer...
$ ./shm_consumer
ptr addr: 0x7fd2f08ab000
Read ptr: Hello, World!
```

POSIX Shared Memory

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                       PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                       MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS

# Running Consumer...
$ ./shm_consumer
ptr addr: 0x7fd2f08ab000
Read ptr: Hello, World!
```

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                   PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                   MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS

# Running Consumer...
$ ./shm_consumer
ptr addr: 0x7fd2f08ab000
Read ptr: Hello, World!

# Checking for Shared Memory...
$ ls -l /dev/shm
total 0
```

POSIX Shared Memory Example

```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
              PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
              MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS

# Running Consumer...
$ ./shm_consumer
ptr addr: 0x7fd2f08ab000
Read ptr: Hello, World!

# Checking for Shared Memory...
$ ls -l /dev/shm
total 0
```

POSIX Shared Memory

POSIX Shared Memory Example

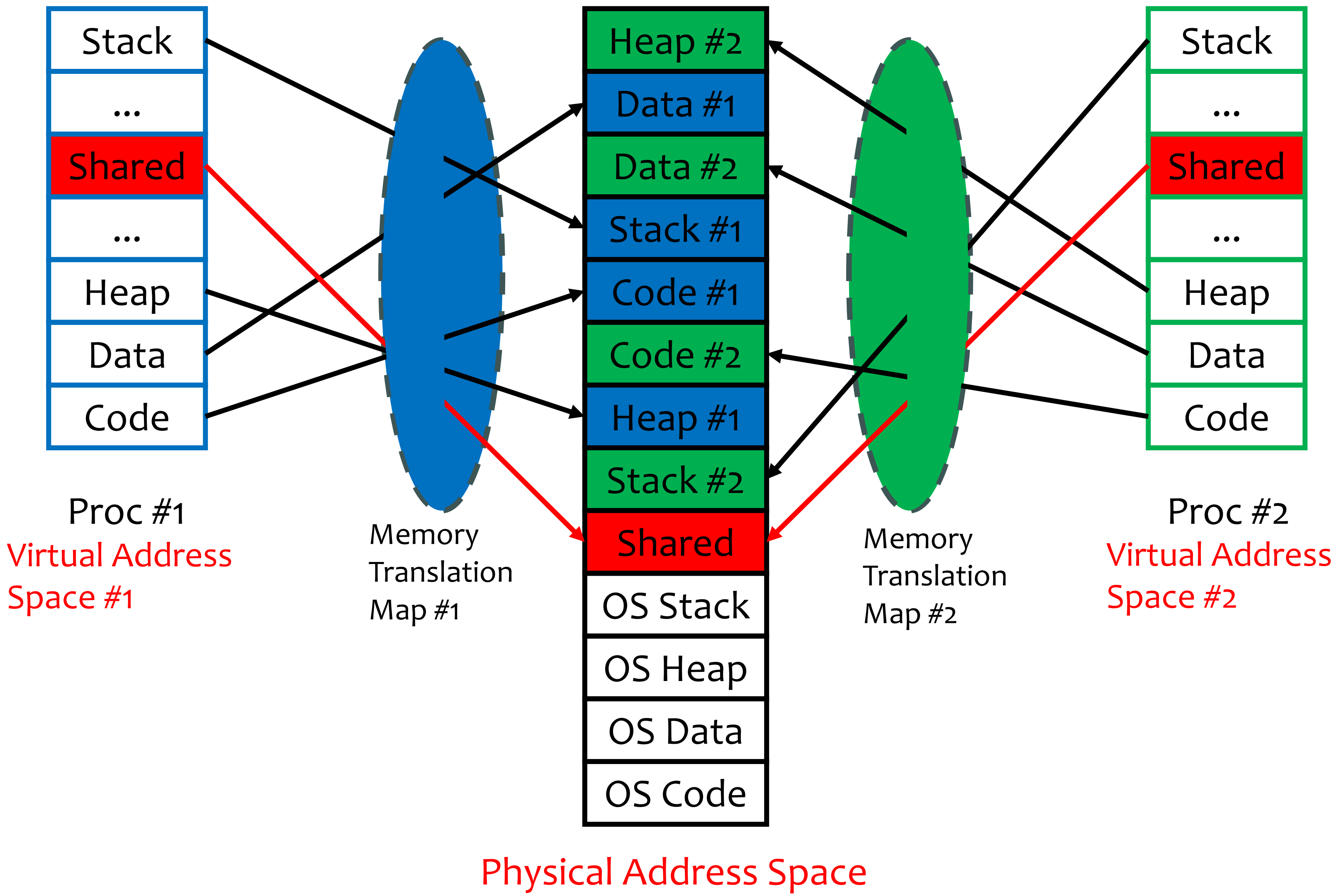
```c
/* shm_producer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared object */
    ftruncate(fd, SIZE);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ |
                  PROT_WRITE, MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* write to the shared memory object */
    sprintf(ptr, "%s", msg);
    return 0;
}
```

```c
/* shm_consumer.c */
int main() {
    /* create the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *)mmap(0, SIZE, PROT_READ,
                  MAP_SHARED, fd, 0);
    printf("ptr addr: %p\n", ptr);
    /* read from the shared memory object */
    printf("Read ptr: %s", ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

```
# Compiling producer and consumer with -lrt
$ gcc -o shm_producer shm_producer.c -lrt
$ gcc -o shm_consumer shm_consumer.c -lrt

# Running Producer...
$ ./shm_producer &
ptr addr: 0x7fb9d69b6000

# Checking for Shared Memory...
$ ls -l /dev/shm
total 1
-rw-rw-r-- 1 zxx zxx 4096 Mar 16 5:15 SYSUOS

# Running Consumer...
$ ./shm_consumer
ptr addr: 0x7fd2f08ab000
Read ptr: Hello, World!

# Checking for Shared Memory...
$ ls -l /dev/shm
total 0
```

Why the address of ptr different in producer and consumer?

Shared Memory



System V IPC Shared Memory

```c
/* sysvshm_producer.c */
#include ...
#define SIZE 5

int main() {
    // Generate a unique key for the shared memory segment
    key_t key = ftok("SYSUOS", 42);
    // Create a shared memory segment
    int shmid = shmget(key, SIZE * sizeof(int), 0666 | IPC_CREAT);
    // Attach shared memory segment to process's address space
    int *arr_shared = (int *)shmat(shmid, NULL, 0);

    for (int i = 0; i < SIZE; i++) {
        arr_shared[i] = i+1;
    }
    printf("Parent wrote to shared memory: %p\n", arr_shared);

    sleep(10);
    // Detach the shared memory segment
    shmdt(arr_shared);
    // Remove the shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

System V IPC Shared Memory

```c
/* sysvshm_consumer.c */
#include ...
#define SIZE 5

int main() {
    // Generate a unique key for the shared memory segment
    key_t key = ftok("SYSUOS", 42);
    // Create a shared memory segment
    int shmid = shmget(key, SIZE * sizeof(int), 0666);
    // Attach shared memory segment to process's address space
    int *arr_shared = (int *)shmat(shmid, NULL, 0);

    printf("Child read from shared memory: %p\n", arr_shared);
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr_shared[i]);
    }
    printf("\n");

    // Detach the shared memory segment
    shmdt(arr_shared);

    return 0;
}
```

System V IPC Shared Memory

```c
/* sysvshm_producer.c */
int main() {
    key_t key = ftok("SYSUOS", 42);
    int shmid = shmget(key, SIZE * sizeof(int),
                       0666 | IPC_CREAT);
    int *arr_shared = (int *)shmat(shmid, NULL, 0);
    for (int i = 0; i < SIZE; i++)
        arr_shared[i] = i+1;
    printf("Parent wrote to shared memory: %p\n",
           arr_shared);
    sleep(10);
    shmdt(arr_shared);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

```c
/* sysvshm_consumer.c */
int main() {
    key_t key = ftok("SYSUOS", 42);
    int shmid = shmget(key, SIZE * sizeof(int),
                       0666);
    int *arr_shared = (int *)shmat(shmid, NULL, 0);
    printf("Child read from shared memory: %p\n",
           arr_shared);
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr_shared[i]);
    }
    printf("\n");
    shmdt(arr_shared);
    return 0;
}
```

```
# Compiling producer and consumer
$ gcc -o sysvshm_producer sysvshm_producer.c
$ gcc -o sysvshm_consumer sysvshm_consumer.c

# Running Producer...
$ ./sysvshm_producer &
Parent wrote to shared memory:
0x7f633db78000

# Running Consumer...
$ ./sysvshm_consumer
Child read from shared memory:
0x7f2fca094000
1 2 3 4 5
```

Thank you!