CS 370: OPERATING SYSTEMS
[INTER PROCESS COMMUNICATIONS]

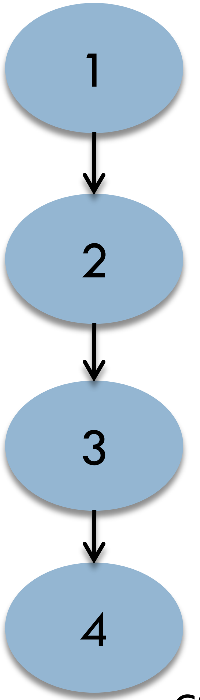
Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALLICKARA

Creating a chain of processes

```
for (int i=1; i < 4; i++) {
    if ((childid = fork())) {
        break;
    }
}
```



value of **i**
when process leaves loop

<u>For each iteration:</u>

Parent has non-ZERO childid

- So it breaks out

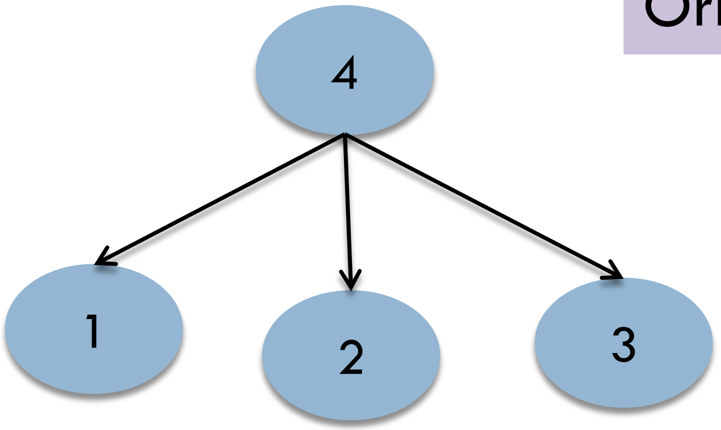
Child process

- Parent in NEXT iteration

Creating a process fan

```
for (int i=1; i < 4; i++) {
    if ((childid = fork()) <= 0) {
         break;
    }
}
```

Newly created process breaks out
Original process continues



value of **i**
when process leaves loop

Making Sure Conditionals in C are Clear

```
for (int i=1; i < 4; i++) {
    if ((childid = fork())) {
        break;
    }
}
```

Conditional is true when fork() returns non-zero value (so, fail or child)

```
for (int i=1; i < 4; i++) {
    if ((childid = fork()) != 0) {
        break;
    }
}
```

Conditional is true when fork() returns non-zero value (so, -1, 42, etc.: fail or child)

```
for (int i=1; i < 4; i++) {
    if ((childid = fork()) <= 0) {
        break;
    }
}
```

Conditional is true when fork() returns negative or zero value (so, fail or parent)

```
for (int i=1; i < 4; i++) {
    if ((childid = fork()) > 0) {
        break;
    }
}
```

Conditional is true when fork() returns positive value (so, child)

fork() == -1 is a failure, rest of code executed by parent

fork() == 0 is a success, rest of code executed by parent

fork() > 0 is a success, rest of code executed by child

Topics covered in this lecture

- Shells and Daemons
- POSIX
- Inter Process Communications

SHELLS AND DAEMONS

Shell: Command interpreter

- Prompts for commands
- Reads commands from standard input
- Forks children to execute commands
- Waits for children to finish
- When standard `I/O` comes from terminal
  - Terminate command with the interrupt character
    - Default `Ctrl-C`

Background processes and daemons

- Shell interprets commands ending with **&** as a background process
  - No waiting for process to complete
  - Issue prompt immediately
    - Accept new commands
  - `Ctrl-C` has no effect, but Shell commands to manipulate processes (fg, bg)
- **Daemon** is a background process
  - Runs "indefinitely": not dependent on Shell termination

POSIX

Portable Operating Systems Interface for UNIX (POSIX)

- 2 **distinct**, **incompatible** flavors of UNIX existed
  - System V from AT&T
  - BSD UNIX from Berkeley
- Programs written from one type of UNIX
  - Did not run correctly (sometimes even compile) on UNIX from another vendor
- Pronounced *pahz-icks*

IEEE attempt to develop a standard for UNIX libraries

- **POSIX.1** published in 1988
  - Covered a small subset of UNIX
- In 1994, X/Open Foundation had
  - Much more comprehensive effort
    - Called **Spec 1170**
  - Based on System V
- Inconsistencies between POSIX.1 and Spec 1170

The path to the final POSIX standard

- **1998**
  - Another version of the X/Open standard
  - Many additions to POSIX.1
  - **Austin Group** formed
    - Open Group, IEEE POSIX, and ISO/IEC tech committee
      - International Standards Organization (ISO)
      - International Electrotechnical Commission (IEC)
    - Revise, combine and update standards

The path to the final POSIX standard: Joint document

- Approved by IEEE & Open Group
  - End of 2001
- ISO/IEC approved it in November 2002
- Single UNIX spec
  - Version 3, IEEE Standard 1003.1-2001
  - **POSIX**

If you write for POSIX-compliant systems

- No need to contend with small, but critical variations in library functions
  - Across platforms

INTER PROCESS COMMUNICATIONS (IPC)

Independent and Cooperating processes

- Independent: **CANNOT** *affect* or *be affected* by other processes
- Cooperating: **CAN** *affect* or *be affected* by other processes

Why have cooperating processes?

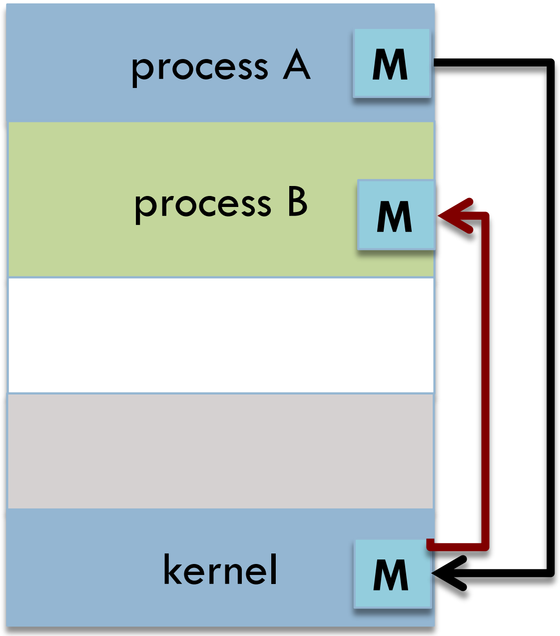
- Information sharing: shared files
- Computational speedup
  - Sub tasks for concurrency
- Modularity
- Convenience: Do multiple things in parallel
- Privilege separation

Cooperating processes need IPC to exchange data and information

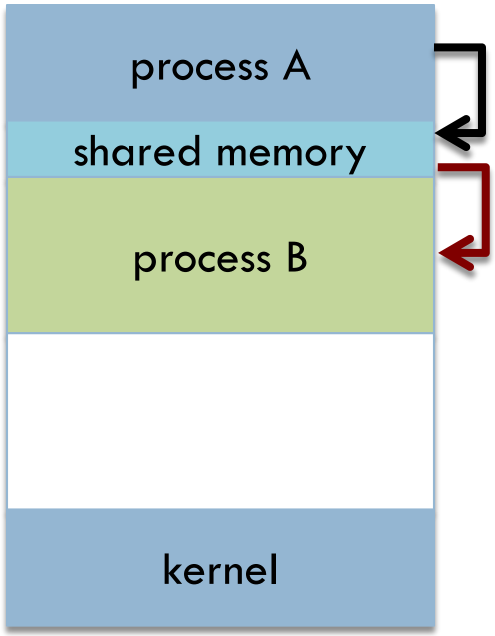
- **Shared memory**
  - Establish memory region to be shared
  - Read and write to the shared region

- **Message passing**
  - Communications through message exchange

Contrasting the two IPC approaches



Easier to implement
Best for **small** amounts of data
Kernel intervention for communications



Maximum **speed**
System calls to **establish** shared memory

Shared memory systems

- Shared memory resides **in** the address space of process creating it

- Other processes must **attach** segment to their address space

Using shared memory

- But the OS typically ***prevents*** processes from accessing each other's memory, so …
  1. Processes must agree to **remove** this ***restriction***
  2. Processes also ***coordinate*** access to this region

Let's look a little closer at cooperating processes

- **Producer-consumer** problem is a good exemplar of such cooperation
- Producer process *produces* information
- Consumer process *consumes* this information

One solution to the producer-consumer problem uses *shared-memory*

- Buffer is a shared-memory region for the 2 processes

- Buffer needed to allow producer & consumer to run **concurrently**
  - Producer fills it
  - Consumer empties it

Buffers and sizes

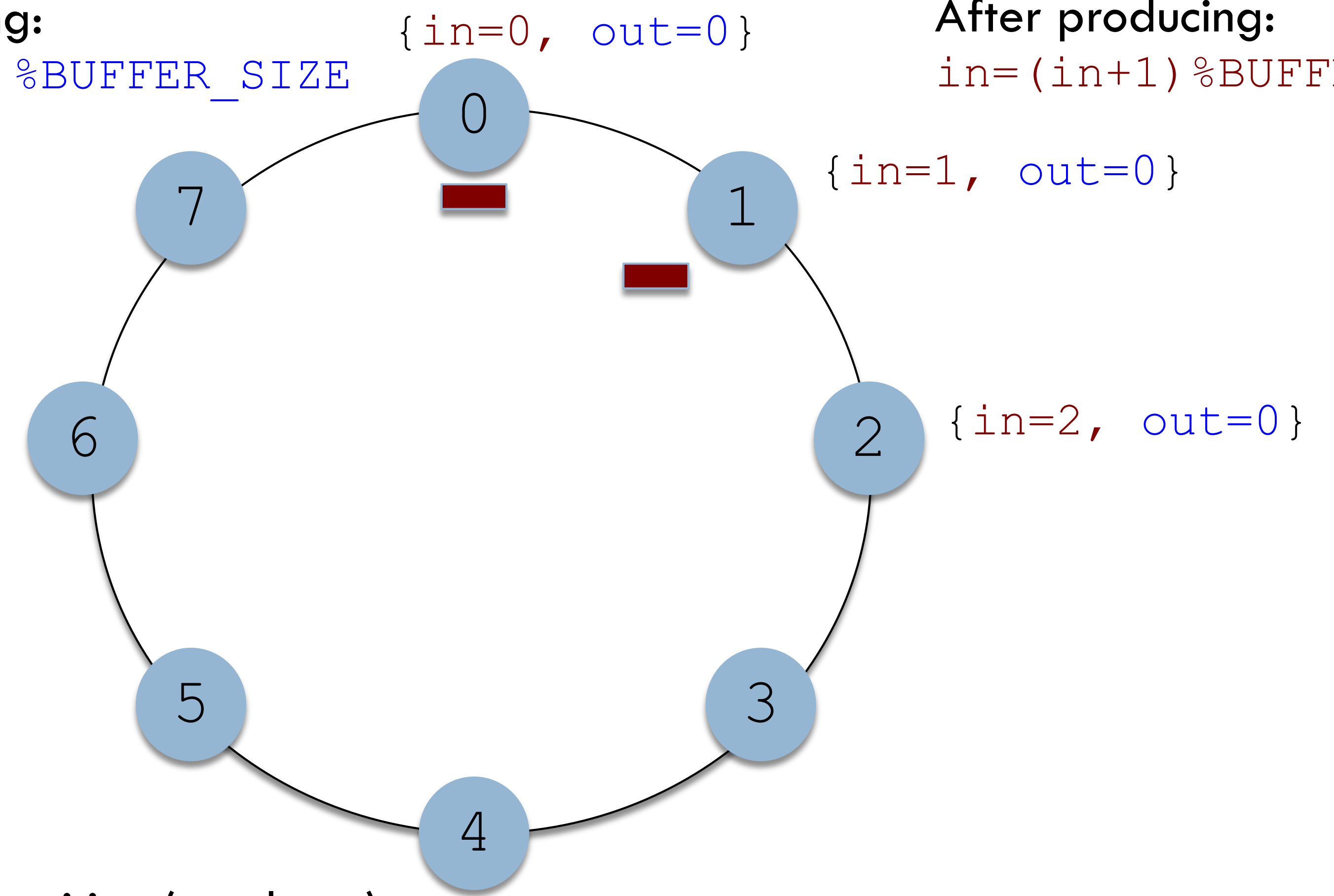
- Bounded: Assume **fixed** size
  - Consumer waits if empty
  - Producer waits if full

- Unbounded: **Unlimited** number of entries
  - Only the consumer waits WHEN buffer is empty

Circular buffer: Bounded

After consuming:
`out=(out+1)%BUFFER_SIZE`

After producing:
`in=(in+1)%BUFFER_SIZE`

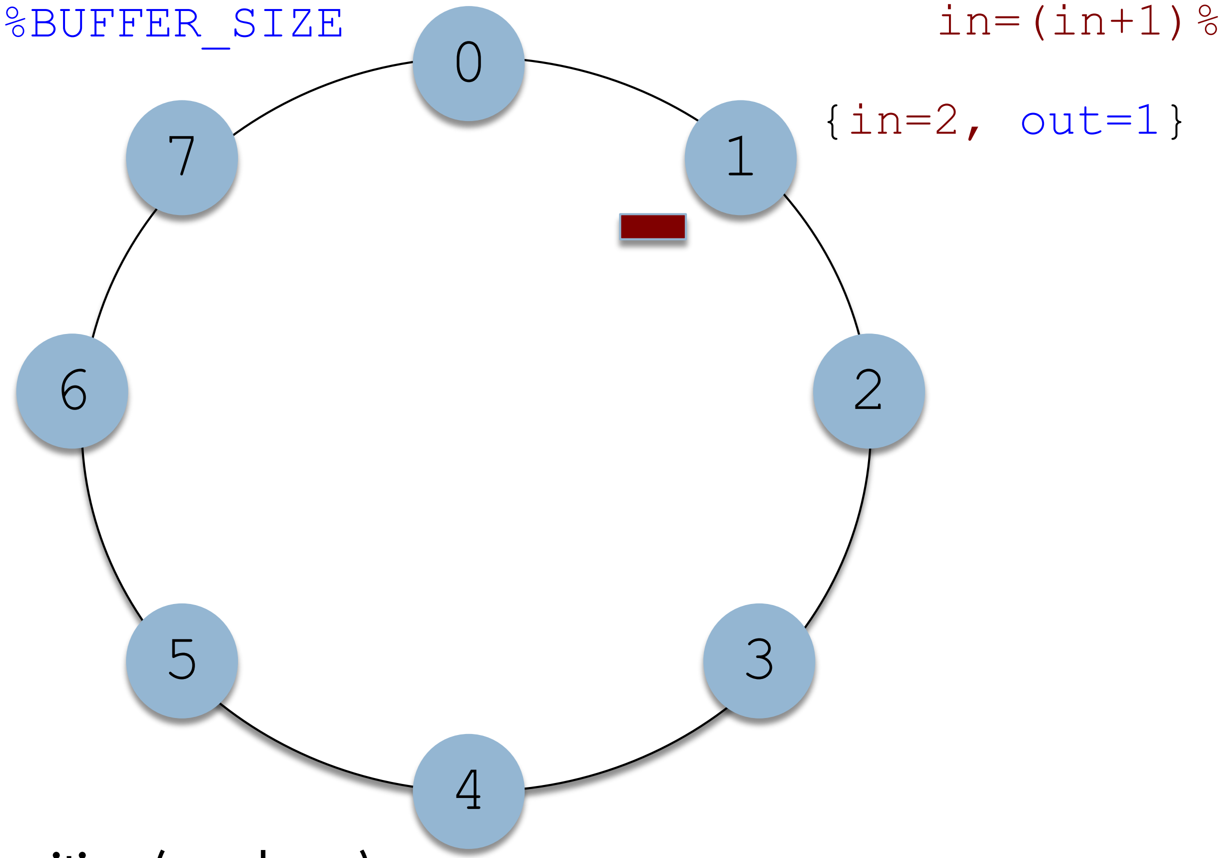


`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

After consuming:
`out=(out+1)%BUFFER_SIZE`

After producing:
`in=(in+1)%BUFFER_SIZE`

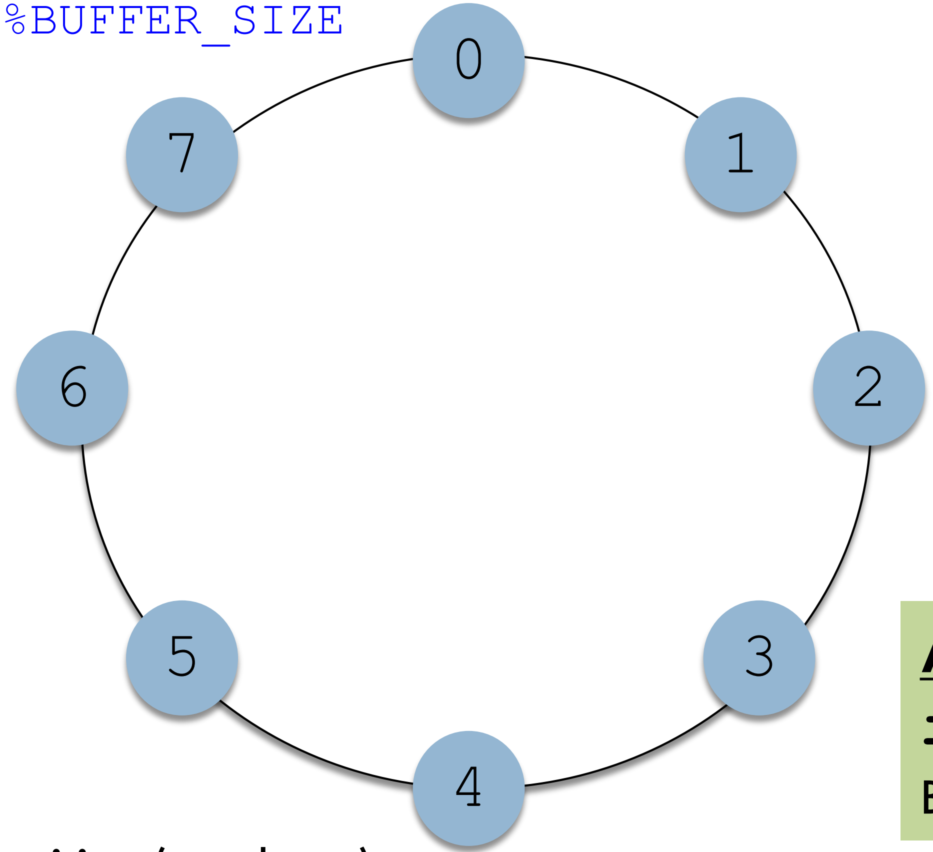


`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

After consuming:
`out=(out+1)%BUFFER_SIZE`

After producing:
`in=(in+1)%BUFFER_SIZE`



`{in=2, out=2}`

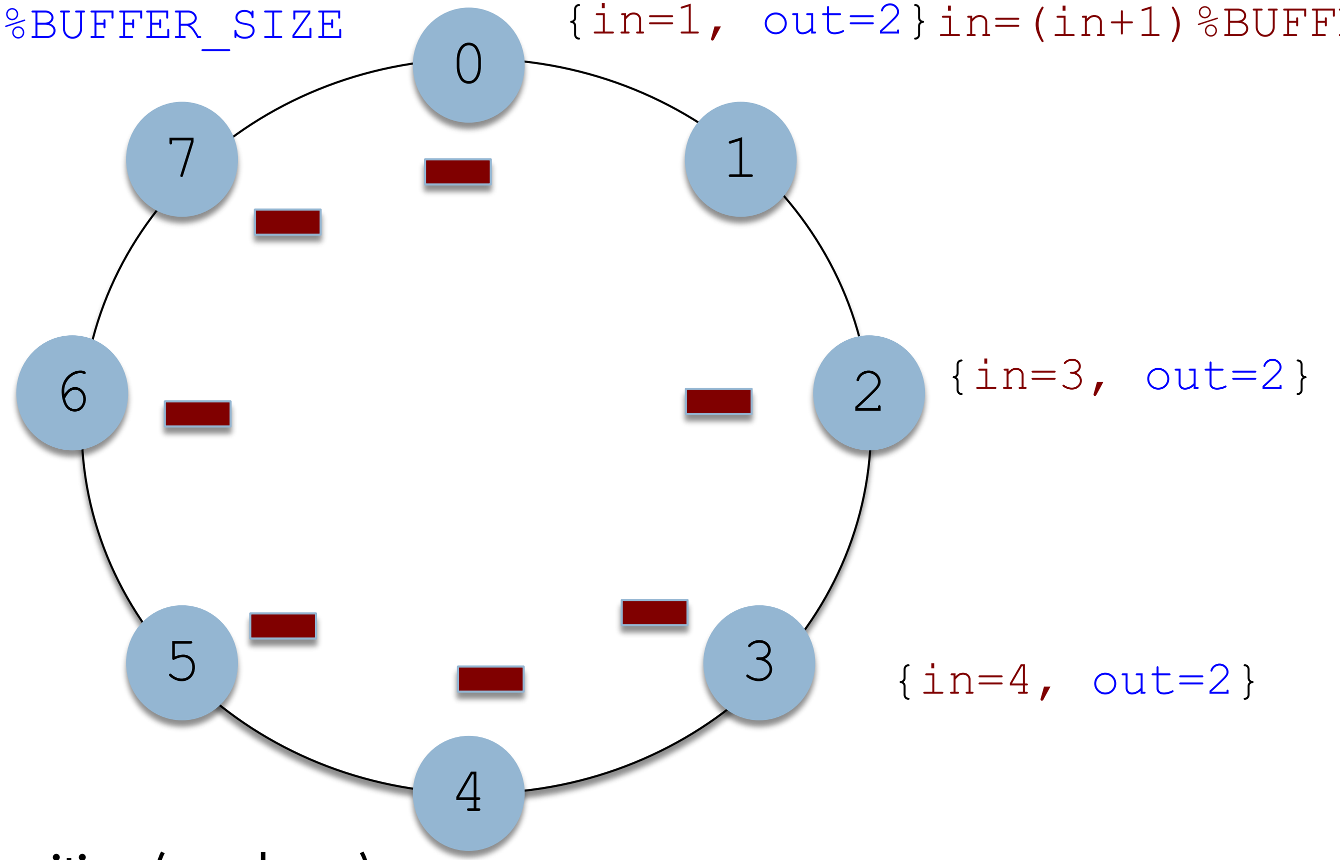
<u>After consuming</u>
`in == out`
Buffer is EMPTY

`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

After consuming:
out=(out+1)%BUFFER_SIZE

After producing:
in=(in+1)%BUFFER_SIZE

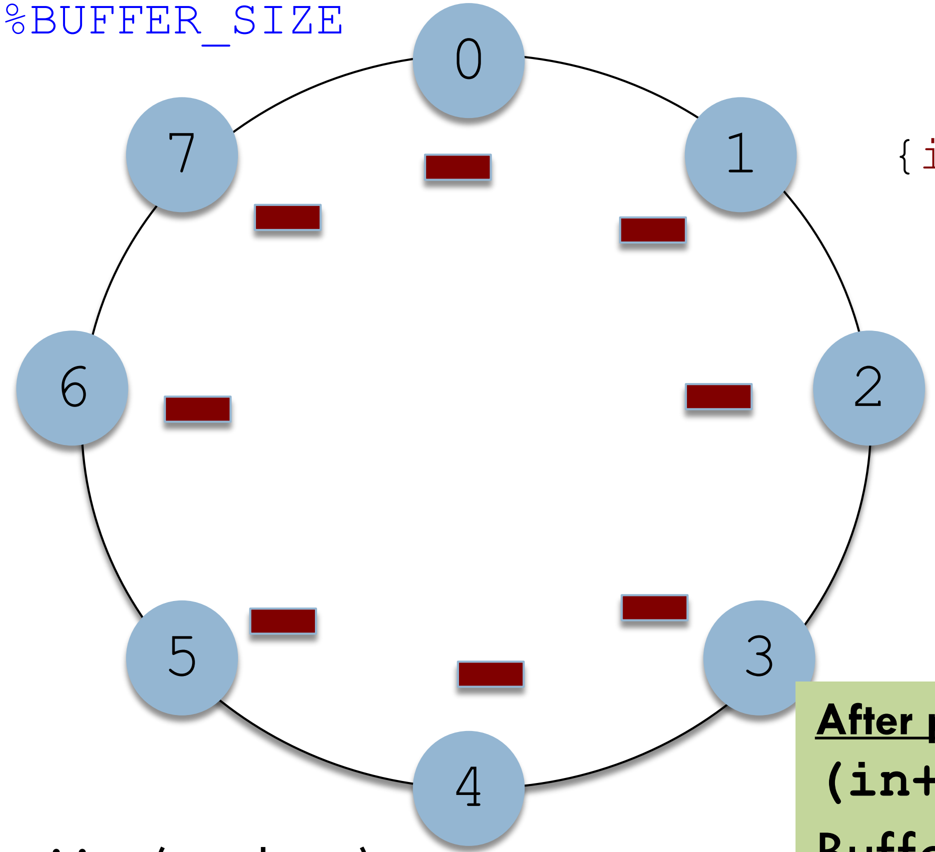


in: next free position (producer)
out: first full position (consumer)

Circular buffer: Bounded

After consuming:
`out=(out+1)%BUFFER_SIZE`

After producing:
`in=(in+1)%BUFFER_SIZE`



`{in=2, out=2}`

After producing:
`(in+1)%BUFFER_SIZE==out`
Buffer is FULL

`in`: next free position (producer)
`out`: first full position (consumer)

INTER PROCESS COMMUNICATIONS

SHARED MEMORY

POSIX IPC: Shared Memory
Creating a memory segment to share

- First **create** shared memory segment `shmget()`

```
shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)
```

  - `IPC_PRIVATE`: key for the segment
  - `size`: size of the shared memory
  - `S_IRUSR|S_IWUSR`: Mode of access (read, write)

- Successful invocation of `shmget()`
  - Returns integer ID of shared segment
    - Needed by other processes that want to use region

Processes wishing to use shared memory must first attach it to their address space

- Done using `shmat()`: SHared Memory ATtach
  - Returns pointer to beginning location in memory

- `(char *) shmat(id, asmP, mode)`
  - `id`: Integer ID of memory segment being attached
  - `asmP`: Pointer location to attach shared memory
    - `NULL` allows OS to *select* location for you
  - Mode indicating read-only or read-write
    - `0`: reads and writes to shared memory

IPC: Use of the created shared memory

- Once shared memory is attached to the process's address space
  - Routine memory accesses using * from `shmat()`
    - Write to it
      - `sprintf(`**`shared_memory`**`, "Hello");`
    - Print string from memory
      - `printf("*%\n", `**`shared_memory`**`);`
- **RULE:** First attach, and then access

IPC Shared Memory: What to do when you are done

① **Detach** from the address space.

- `shmdt()` :SHared Memory DeTtach
- `shmdt(shared_memory)`

② To **remove** a shared memory segment

- `shmctl()` : SHared Memory ConTroL operation
  - Specify the segment ID to be removed
  - Specify operation to be performed: `IPC_RMID`
  - Pointer to the shared memory region

INTER PROCESS COMMUNICATIONS

MESSAGE PASSING

Communicate and synchronize actions without sharing the same address space

- Useful in distributed environments (e.g., Message Passing Interface)
- Two main operations
  - `send(message)`
  - `receive(message)`
- Message sizes can be:
  - Fixed: Easy
  - Variable: Little more effort

Communications between processes

- There needs to be a communication link
- Underlying physical implementation
  - Shared memory
  - Hardware bus
  - Network

Aspects to consider for IPC

① **Communications**

- Direct or indirect

② **Synchronization**

- Synchronous or asynchronous

③ **Buffering**

- Automatic or explicit buffering

Naming allows processes to refer to each other

- Processes use each other's identity to communicate
- Communications can be
  - Direct
  - Indirect

Direct communications

- Explicitly name recipient or sender
- Link is established automatically
  - Exactly one link between the 2 processes
- Addressing
  - Symmetric
  - Asymmetric

Direct Communications: Addressing

- Symmetric addressing
  - `send(P, message)`
  - `receive(Q, message)`

Explicitly name recipient and sender of message

- Asymmetric addressing
  - `send(P, message)`
  - `receive(id, message)`
    - Variable `id` set to name of the sending process

Only sender names recipient
Recipient does not

Direct Communications: Disadvantages

- **Limited modularity** of process definitions
- **Cascading effects** of changing the identifier of process
  - Examine *all* other process definitions

Indirect communications: Message sent and received from mailboxes (ports)

- Each **mailbox** has a unique identification & owner
  - POSIX message queues use `integers` to identify mailboxes
- Processes communicate *only* if they have **shared mailbox**
  - `send(`**`A`**`, message)`
  - `receive(`**`A`**`, message)`

Indirect communications: Link properties

- Link established only if both members share mailbox
- Link may be associated with more than two processes

Indirect communications

- Processes `P1`, `P2` and `P3` share mailbox `A`
  - `P1` sends a message to `A`
  - `P2`, `P3` execute a `receive()` from `A`

- Possibilities? Allow …
  1. Link to be associated with at most 2 processes
  2. At most 1 process to execute `receive()` at a time
  3. System to arbitrarily select who gets message

Mailbox ownership: Owned by OS

- Mailbox has its own existence
- Mailbox is **independent**
  - Not attached to any process
- OS must allow processes to
  - Create mailbox
  - Send and receive *through* the mailbox
  - Delete mailbox

Message passing: Synchronization issues Options for implementing primitives

- Blocking send
  - Block *until* received by process or mailbox
- Nonblocking send
  - Send and *promptly resume* other operations
- Blocking receive
  - Block *until* message available
- Nonblocking receive
  - Retrieve *valid* message or *null*
- Producer-Consumer problem: Easy with blocking

Message Passing: Buffering

- Messages exchanged by communicating processes reside in a **temporary** queue

- Implementation schemes for queues
  - ZERO Capacity
  - Bounded
  - Unbounded

Message Passing Buffer: Consumer always has to wait for message

- ZERO capacity: No messages can reside in queue
  - Sender ***must block*** till recipient receives
- BOUNDED: At most `n` messages can reside in queue
  - Sender **blocks *only if* queue is *full***
- UNBOUNDED: Queue length potentially infinite
  - Sender ***never blocks***

MICROKERNELS

The Microkernel Approach [1/2]

- Mid 1980's at Carnegie Mellon University
  - **Mach**
- Structure OS by *removing non-essential components* from the kernel
  - Implement other things as system/user programs
- Provide minimal process and memory management
- Main function: Provide communication facility between client and services
  - **Message passing**

The Microkernel Approach [2/2]

- Traditionally all the layers went in the kernel
  - But this is not really necessary
- In fact, it may be best to *put as little as possible* in the kernel
  - Bugs in the kernel can bring down the system instantly
- Contrast this with setting up user processes to have less power
  - A bug may not be fatal

Getting there …

- Achieve high reliability by splitting OS in small, well-defined modules
  - One of these, the microkernel, runs in the kernel mode
  - The rest as relatively powerless ordinary user processes
- Running each device driver as a separate process?
  - Bugs cannot crash the entire system

Communications in the micro-kernel

- Client and service never interact directly
- Indirect communications by exchanging messages with the microkernel
- Advantages
  - Easier to port to different hardware
  - More security and reliability
    - Most services run as user, rather than kernel
- **Mac OS X kernel based on Mach microkernel**

Increased system function overhead can degrade microkernel performance

- Windows NT: First release, layered microkernel
  - Lower performance than Windows 95
- Windows NT 4.0 solution
  - Move layers from user space to kernel space
- By the time Windows XP came around
  - More monolithic than microkernel

IPC communications: Mach

- Tasks are similar to processes
  - Multiple threads of control
- Most communications in Mach use **messages**
  - System calls
  - Inter-task information
  - Sent and received from mailboxes: *ports*

Mach: Task creation and mailboxes

- Task creation results in 2 more mailboxes
  1. Kernel mailbox: Used by kernel to communicate with task
  2. Notify mailbox: Notification of event occurrences

- System calls for communications
  - `msg_send()`, `msg_receive()` and `msg_rpc()`

Mach: Mailbox creation

- Done using the `port_allocate()`
  - Allocate space for message queue
    - MAX_SIZE default is 8 messages
- Creator is owner and can also receive
- Only task can own/receive from mailbox
  - BUT these **rights can be sent** to other tasks

Mach: Message queue ordering

- FIFO guarantees for messages from same sender
- Messages from multiple senders queued in any order

Mach: Send and receive operations

- If mailbox is not full, copy message
- If mailbox is FULL
  1. Wait indefinitely till there's room
  2. Wait at most **n** milliseconds
     - Don't wait, simply return
  3. Temporarily cache the message
     - **Only 1** message to a full mailbox can be *pending* for a *given* sending thread
- Receive can specify mailbox or mailbox set

Another idea related to microkernels

- Put **mechanisms** for doing something in the *kernel*
  - But <u>not the policy</u>
- Example: Scheduling
  - Policy of assigning priorities to processes can be done in the user-mode
  - The mechanism to look for the highest priority process and schedule it is in the kernel

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 3]
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 2, 3]
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2]