CS 370: OPERATING SYSTEMS
[INTER PROCESS COMMUNICATIONS]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- When you fork() are objects and data of the process shared or is a new copy of the heap created?
  - Everything is copied
- Why is wait() called in the parent and exec() in the child?
  - Can you wait for multiple children?
- When you call exec() on child, is the parent affected?
  - What does exec() destroy? COPY of the memory image of the parent
- Zombies and Orphans
  - What happens after adoption by init?

Frequently asked questions from the previous class survey

- Why would you ever make copies of programs like we did in the code snippets?
- As you fork processes, upon completion of the process creation are they considered ready for scheduling by the kernel?
- Automatic variables? What are they?
- Kernel strategies for preventing some of the attacks?
  - ASLR: Address space layout randomization
  - Non-executable stack

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
- **Daemon** is a background process
  - Runs indefinitely

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 a 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

September 4, 2018
Professor: Shrideep Pallickara

CS370: *Operating Systems* [Fall 2018]
Dept. Of Computer Science, Colorado State University

L5.11

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**

September 4, 2018
Professor: Shrideep Pallickara

CS370: *Operating Systems* [Fall 2018]
Dept. Of Computer Science, Colorado State University

L5.12

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

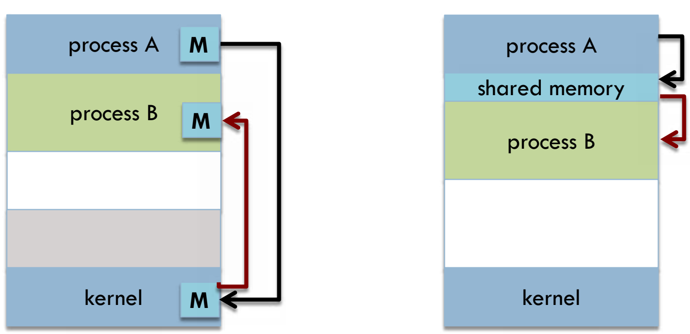
Why have cooperating processes?

- 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

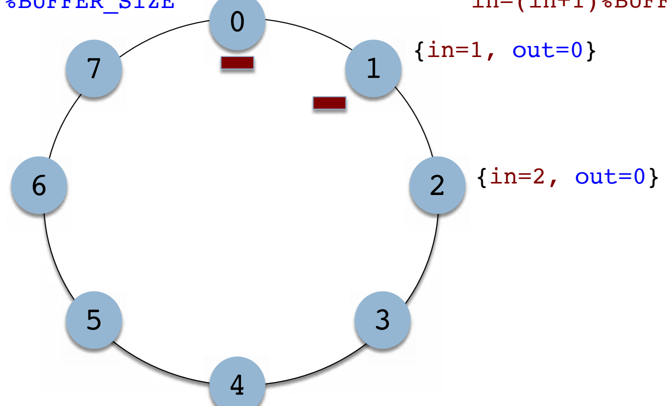
- Bounded: Assume **fixed** size
  - Consumer waits if buffer is empty
  - Producer waits if buffer is full
- Unbounded: **Unlimited** number of entries
  - Only the consumer waits WHEN buffer is empty

Circular buffer: Bounded

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

`{in=0, out=0}`

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

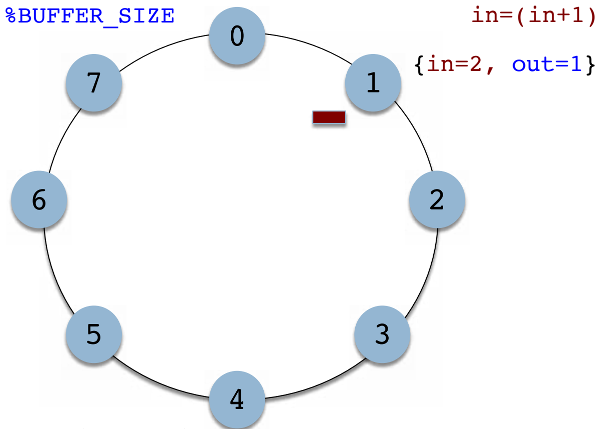


`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

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

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

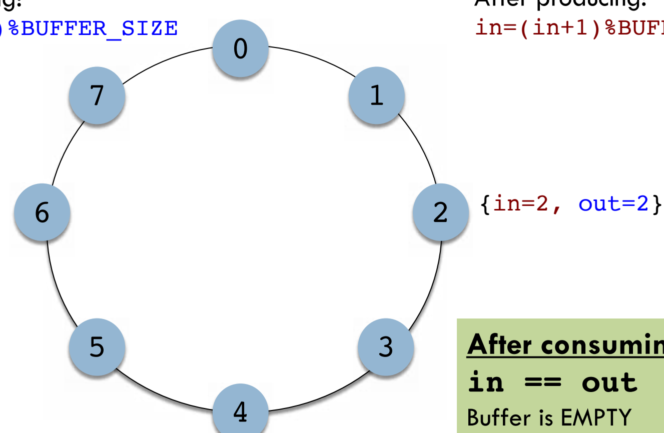


`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

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

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



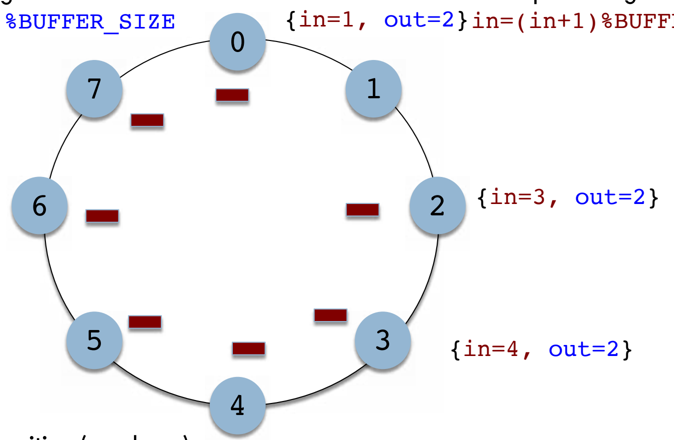
After consuming
`in == out`
Buffer is EMPTY

`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`

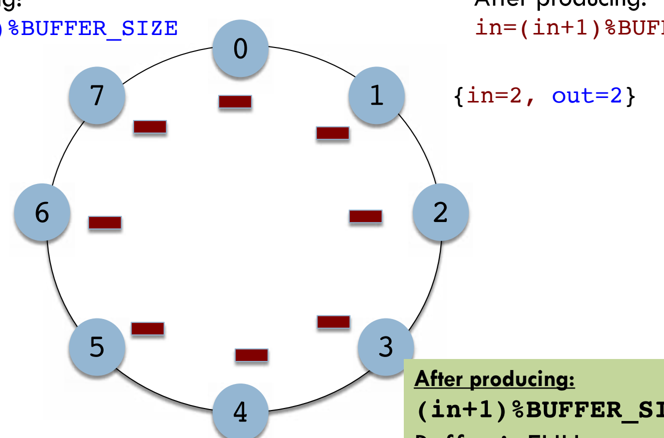


`in`: next free position (producer)
`out`: first full position (consumer)

Circular buffer: Bounded

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

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



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

- **`(void *) 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("*%s\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

- 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

Communications: 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 ← Explicitly name recipient and sender of message
  - `send(P, message)`
  - `receive(Q, message)`
- Asymmetric addressing ← Only sender names recipient Recipient does not
  - `send(P, message)`
  - `receive(id, message)`
    - Variable `id` set to name of the sending process

Direct Communications: Disadvantages

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

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 issues

- Owned by process
- Owned by the OS

Mailbox ownership issues: Owned by process

- Mailbox is part of the **process's address space**
  - Owner: Can *only receive* messages on mailbox
  - User: Can *only send* messages to mailbox
- When process terminates?
  - Mailbox disappears

Mailbox ownership issues: 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***

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]