Frequently asked questions from the previous class survey

- Heap vs global variables
- Resource leak
- Process and process groups
  - Orphaned children and zombies: when would a parent not be waiting? Why should the parent wait?
  - What if zombies aren’t removed?
  - Can grandparenters wait for a grandchild?
- Any benefits of having no process hierarchy?
- fork()
  - Similar to recursion?
- Is the child initially an exact copy of the parent? Does it have the same heap, stack, and program counter? WHY?
- exec(): What does it do?
- Exit and return codes
- Difference between wait and wait_pid()

Topics covered in this lecture

- Shells and Daemons
- POSIX
- Inter Process Communications

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

Shells and Daemons

February 4, 2016
CS370: Operating Systems [Spring 2016]
Dept. Of Computer Science, Colorado State University
Background processes and daemons

- Shell interprets commands ending with `&` as a background process
- No waiting for process to complete
- Issue prompt immediately
- `Ctrl-C` has no effect
- **Daemon** is a background process
- Runs indefinitely

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**

POSIX

- **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)
  - Revisit, 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
  - **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)**

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

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

- **Message passing**
  - 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`

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

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

- `out`: next free position (producer)
- `in`: first full position (consumer)
Circular buffer: Bounded

After consuming:
\[ \text{out} = (\text{out} + 1) \mod \text{BUFFER\_SIZE} \]
\[ \text{in} = (\text{in} + 1) \mod \text{BUFFER\_SIZE} \]

In: next free position (producer)
Out: first full position (consumer)

After consuming:
\[ \text{out} = (\text{out} + 1) \mod \text{BUFFER\_SIZE} \]
\[ \text{in} = (\text{in} + 1) \mod \text{BUFFER\_SIZE} \]

After producing:
\[ \text{in} = (\text{in} + 1) \mod \text{BUFFER\_SIZE} \]

Inter Process Communications

Inter Process Communications

Shared Memory

POSIX IPC: Shared Memory

Creating a memory segment to share

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

\[
\text{shmget}(\text{IPC\_PRIVATE, size, S\_IRUSR | S\_IWUSR})
\]

- 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
    - `O`: reads and writes to shared memory

Routine memory accesses using `*` from `shmat()`

- Write to it
  - `*`: from `shmat()`
- Print string from memory
  - `*`: from `shmat()`

**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 Detach
     - `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

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**
  - `send(P, message)`
  - `receive(Q, message)`

- **Asymmetric addressing**
  - `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 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 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

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

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