**Frequently asked questions from the previous class survey**

- Are process IDs assigned sequentially?
- Worm vs Process
- Worm: self replicating and tries to spread itself
- How can attacks via buffer overflows be prevented?
  - ASLR: Address space layout randomization
  - Non-executable stack

- Process tree creation: How deep?
- Does the process group id change automatically?
- Practical use of process group ids?
- Is it critical that `exec()` wipes out parent's memory image?
- Where does a child begin execution after `fork()`?
- If you change a process group ID for a child and there are existing children, do their process group id's change?
- Core dump
- Waiting processes
- Why wait? Zombie processes? Why can't a process clear all its data? Can you kill a zombie process?
- `waitpid` and the return value from `fork()`?
- `Shell`

**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`
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

---

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

---

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)

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

Why have cooperating processes?

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

Contrasting the two IPC approaches

<table>
<thead>
<tr>
<th>Easier to implement</th>
<th>Maximum speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best for small amounts of data</td>
<td>System calls to establish shared memory</td>
</tr>
<tr>
<td>Kernel intervention for communications</td>
<td>System calls to establish shared memory</td>
</tr>
</tbody>
</table>

Independent and Cooperating processes

- Independent: **CANNOT** affect or be affected by other processes
- Cooperating: **CAN** affect or be affected by other processes
Shared memory systems
- Shared memory resides in the address space of the process creating it
- Other processes must attach a 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: \( \text{out} = (\text{out} + 1) \mod \text{BUFFER_SIZE} \)
- After producing: \( \text{in} = (\text{in} + 1) \mod \text{BUFFER_SIZE} \)

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

After consuming:
\[ \text{in} = \text{in} + 1 \mod \text{BUFFER_SIZE} \]

After producing:
\[ \text{out} = \text{out} + 1 \mod \text{BUFFER_SIZE} \]

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

After consuming:
\[ \text{out} = \text{out} + 1 \mod \text{BUFFER_SIZE} \]

After producing:
\[ \text{in} = \text{in} + 1 \mod \text{BUFFER_SIZE} \]

POSIX IPC: Shared Memory
Creating a memory segment to share

- First create shared memory segment \( \text{shmget}() \)
  \[ \text{shmget}(\text{IPC_PRIVATE}, \text{size}, \text{S_IRUSR} | \text{S_IWUSR}) \]
  - IPC_PRIVATE: key for the segment
  - size: size of the shared memory
  - S_IRUSR | S_IWUSR: Mode of access (read, write)

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

Inter Process Communications
Shared Memory
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

```c
(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
    - `printf(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 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

Communications
- Direct or indirect

Synchronization
- Synchronous or asynchronous

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 two 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 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