Operating Systems

Inter-process Communication (IPC)

Cooperating Processes

- **Independent** processes cannot affect or be affected by the execution of other processes
  - Part of the goal of the process abstraction is to insure this
- **Cooperating** processes can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to coordinate their actions
- Signals
- Shared Memory
  - Designated region of memory visible across processes
- Byte streams
  - Pipes (unnamed)
  - Named Pipes
- Message Passing
  - processes communicate with each other without resorting to shared variables

Operating Systems

Inter-process Communication (IPC)

Signals

Signals

- A signal is an *asynchronous* event which is delivered to a process
  - Asynchronous implies that the event can occur at any time
  - e.g. user types `ctrl-C` which generates `SIGINT`
- Often associated with system events
- Generating a signal involves setting a flag to deliver that signal to the process when it is rescheduled
  - Generation is asynchronous
  - The delivery doesn't happen until the process runs
  - Preemption and resumption happen asynchronously, so the net result is asynchronous delivery

5

POSIX Signals

- SIGFPE: Illegal mathematical operation.
- SIGHUP: Controlling terminal hang-up.
- SIGILL: Execution of an illegal machine instruction.
- SIGINT: Process interruption. Can be generated by ^C
- SIGTERM: Graceful process termination
- SIGKILL: Kill a process. Cannot be caught or ignored "***kill** -9 <process_id>*" command
- SIGPIPE: Illegal write to a pipe.
- SIGQUIT: Process quit. Generated by *<crtl_\>* keys.
- SIGSEGV: Segmentation fault. generated by dereferencing an invalid pointer.

6

Signal blocking

- The signal mask is the set of signals that are currently blocked
  - Block some signals, so they don't reach the process right away.
- The modern calls are as follows: (the older `signal()` is less portable)
  - `sigemptyset(&newsigset);`
  - `sigaddset(&newsigset, SIGINT);`
  - `sigprocmask(SIG_BLOCK,&newsigset, NULL);`
    - SIG_BLOCK The new mask is the union of the current mask and the specified set.
    - SIG_UNBLOCK The new mask is the intersection of the current mask and the complement of the specified set.
    - SIG_SETMASK The current mask is replaced by the specified set

7

Signal Actions

- In a `struct sigaction act`:
  - `sa_handler` is the function called upon receiving the signal
  - `sa_flags` are special flags
  - `sa_mask` are the additional signals to block while handling this signal
- `sigaction(SIGUSR1, &act1, NULL)` associates the handler `act1()` with `SIGUSR1`.

8

Other Notes on Signals

- You can control the signal masks on a per-thread basis
  - New threads inherit creator's signal mask
  - `pthread_sigmask(SIG_BLOCK, &newsigset, &oldsigset)` changes it
    - SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
  - Designate one thread as the signal hander, or one per signal…
- You can create an alternate stack for signal handlers with `sigaltstack()`
  - from the man page: "The most common usage of an alternate signal stack is to handle the **SIGSEGV** signal that is generated if the space available for the standard stack is exhausted: in this case, a signal handler for **SIGSEGV** cannot be invoked on the standard stack; if we wish to handle it, we must use an alternate signal stack."

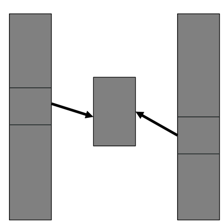
Operating Systems

Inter-process Communication (IPC)

Shared Memory

Shared Memory Communication Model

- Shared Memory
  - Memory is inherently shared with threads, but not processes
  - Can be mapped to be shared between processes
  - Only explicitly specified regions are shared, limiting access and potential damage from errant execution
- As opposed to separated Memory
  - Byte streams or messages
- The user must manage synchronization



System V Shared Memory

- Process creates a shared memory segment with `shmget()`
- Another process attaches to the segment with `shmat()`
- `shmctl()` allows properties to be changed after creation
- `shmdt()` detaches

Shared Memory Example

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main()
{
   /* the identifier for the shared memory segment */
   int segment_id;
   /* a pointer to the shared memory segment */
   char* shared_memory;
   /* the size (in bytes) of the shared memory segment */
   const int segment_size = 4096;
   /* allocate  a shared memory segment */
   segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);
   /* attach the shared memory segment */
   shared_memory = (char *) shmat(segment_id, NULL, 0);
   printf("shared memory segment %d attached at address %p\n",
   segment_id, shared_memory);
```

13

Shared Memory Example (2)

```
   /* write a message to the shared memory segment   */
   sprintf(shared_memory, "Hi other process!");

   /* now print out the string from shared memory */
   printf("*%s*\n", shared_memory);

   /* now detach the shared memory segment */
   if ( shmdt(shared_memory) == -1) {
       fprintf(stderr, "Unable to detach\n");
   }
   /* now remove the shared memory segment */
   shmctl(segment_id, IPC_RMID, NULL);
   return 0;
}
```

14

Shared Memory Notes

- There is a POSIX version that we will come back to after we talk about memory management and files
  - Slightly different model with a file as backing store
- Shared memory is import in database systems like Oracle

15

Operating Systems

Inter-process Communication (IPC)

Byte Streams

16

Byte Stream Communication

- Pipe
- Named Pipe / FIFO
- Sockets
- Remote Procedure Calls

17

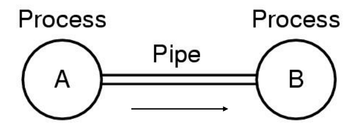
IPC with a pipe

man -s 2 pipe or man 2 pipe

```
int
pipe(int *fildes);
```

The **pipe**() function creates a pipe, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors.

fildes[1] is the write end, fildes[0] is the read end



18

Named Pipe / FIFO

- FIFO in UNIX
  - mkfifo()
  - open()
  - read()
  - write()
  - close()
- Named pipes in Windows
  - CreateNamedPipe()
  - ConnectNamedPipe()
  - ReadFile()
  - WriteFile()

19

Sockets

- The Sockets API originated in BSD Unix
- A socket is an endpoint uniquely identified by an Internet Protocol (IP) address and port
  - The socket **129.79.39.208:80** refers to port **80** on host **129.79.39.208**
- Communication occurs between a pair of sockets
- Unix Domain Sockets are similar to Named Pipes
  - No network protocols
  - Communication occurs within the kernel
  - Can deliver datagrams

20

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- Some interface definition language used to generate client and server stubs in various languages
- Some mechanism to discover interfaces and bind to a particular server
  - Directory service

Various Implementations

- Common Object Request Broker Architecture (CORBA)
- Component Object Model (COM)
  - Windows
- Binder in Android
- Web Services

Operating Systems

Inter-process Communication (IPC)

Message Passing

Message Passing

- Message Passing is Inter-Process Communication (IPC) in which one process transfers data (a message) to another
  - Can be local to a machine or traverse a network
- Some operating systems (microkernels) use this as fundamental functionality
  - Even module to module communication
- Message passing solutions can be based around mailboxes, or pickup points
  - Indirect operation: drop off, pick up
- Or a solution may be point to point
  - Direct operation: send to a specific process

Coordination

- Message passing can provide process coordination if a process can be suspended until a message arrives
  - Can message passing be used in place of semaphores for synchronization?
- Two main distinctions:
  - Synchronous – if a receiver attempts to receive before a message is available, the process is blocked
  - Asynchronous – a message can arrive at any time and the receiver is notified or must check

Message Passing

- A message passing facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation / Design Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender could be notified about who the receiver was.

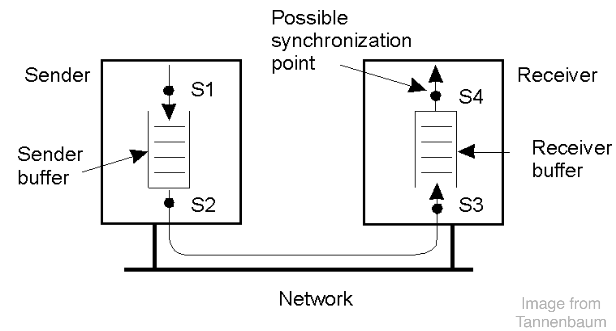
Synchronous vs Asynchronous

- Synchronous – "existing or occurring at the same time" but we might expand "existing" to mean *synchronized so that both are "in" the calls at the same time*
  - This maps well to traditional models
  - Invoke system call and return on completion
- In the synchronous case, what about the sender and receiver?
  - If the sender sends before the receiver receives, then the sender blocks
  - If the receiver receives before the sender send, then the receiver blocks
  - Buffering of one or more messages makes this a bounded buffer case
- Asynchronous might involve polling (checking periodically), or it might operate like a signal
  - Additional overhead but can be more convenient
- Hybrid approaches are obviously possible
  - Wait for a while, or ask if it would block if waited on…

Blocking vs Non-Blocking

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender “send” the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null
- Post Office visit or put it in the mailbox?

Synchronization Points for Network Message Passing



Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Naming

- We can contrast direct and indirect naming
- Direct communication
  - Send to a particular process
  - Receive from a particular process or wildcard
- Indirect communication
  - Mailboxes or ports
  - Abstract object into which the message is placed
- Direct messages to an indirect name?
  - Like the web server www.iu.edu – you communicate *directly* with an instance of it but there is *indirection* in the name
- The key idea of indirect messaging is that the binding (to a recipient) can occur after the message is sent

IPC with Mach Ports

- Mach is a microkernel, which are minimal in terms of privileged execution – ideally only for messaging (which necessitates elevated privilege)
- Ports
  - Microkernel protected communication channel
  - Communication occurs by sending messages to ports
  - Microkernel object reference mechanism
  - Allow objects to transparently reside anywhere in network
  - Threads have port rights (send & receive)
- Port sets
  - Group of ports sharing a common message queue
  - By receiving messages for a port set, a thread can service multiple ports
  - Similar to `select()` call in BSD Unix – check a set of things
- Messages
  - Typed collection of data objects

Mach Ports

- Communication channels and object reference mechanism
  - Methods on objects invoked via messages
  - Enable a task to send data to another task in a controlled manner
    - Kernel protected; implemented as a bounded queue
- Send & receive *rights*
  - Only one task with receive rights
  - Can be multiple with send rights
  - Sending receive rights to another task causes ownership of receive rights to change

Mach Ports

- Ports are location independent
  - Sending a message to port will result in the receiver task getting the message independent of where the receiver is on network
  - Tasks and corresponding ports can be migrated to another computer with the same machine architecture
- The idea is great, but the performance is problematic
  - We've talked about the effects of latency throughout the stack
  - For going over the network, the granularity of operations is small
  - Even on a single machine, the overhead of data copying makes pure messaging too slow
  - Apple's XNU (the kernel of mac OS, etc.) is hybrid, with more functionality in privileged space to eliminate copying

Windows ALPC

- Windows implements an interface called Advanced Local Procedure Call (ALPC)
- Uses kernel "port" objects
- Short messages (<256B) are copied into the kernel
- Larger messages are copied into a shared memory region that is mapped between processes
- Direct read/write with rendezvous when the amount of data is too large for shared region
- ALPC added I/O completion ports to provide an asynchronous interface to eliminate blocking