Logistics

- Homework 1 due!
- Homework 2 posted later today!

- Programming assignment #1 to be posted Thursday.
  - This will be a 2 week assignment.
  - It will have two parts corresponding to chapters 3 and 4, one part each.
Continuing with processes

- Last time we discussed processes in isolation.
  - What is a process?
  - How is a process structured?
  - How does the OS represent a process?

- This time we delve further and think about:
  - How do we make new processes?
    - We started this briefly at the very end of class last time.
  - How do processes coordinate and communicate to work together.
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes.
- Generally, process identified and managed via a **process identifier** (pid).

- Multiple potential situations for resource sharing:
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

- **Execution**:
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont)

- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it

- **Important detail:**
  - Address space is *duplicate* of parent. It is not the same physical address space though.

- **UNIX examples**
  - *fork* system call creates new process
  - *exec* system call used after a *fork* to replace the process’ memory space with a new program
Process Creation

fork() -> child -> exec() -> exit() -> wait

parent -> resumes

fork()
int main()
{
    pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete*/
        wait (NULL);
        printf("Child Complete");
        exit(0);
    }
}
A tree of processes on a typical Solaris
### OSX ps output

```bash
magnolia:~ matt$ ps -ef

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME TTY</th>
<th>TIME CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00:39 ??</td>
<td>00:08.54 /sbin/launchd</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>01:03 ??</td>
<td>00:01.19 /usr/libexec/kextd</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>00:77 ??</td>
<td>00:01.84 /usr/sbin/DirectoryService</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>00:43 ??</td>
<td>00:09.63 /usr/sbin/notifypd</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>02:41 ??</td>
<td>00:05.99 /usr/sbin/syslogd</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>00:92 ??</td>
<td>00:01.67 /usr/sbin/daemon</td>
</tr>
<tr>
<td>65</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>00:19 ??</td>
<td>00:09.16 /usr/sbin/mDNSResponder -launchd</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>00:46 ??</td>
<td>00:01.14 /usr/sbin/istated</td>
</tr>
<tr>
<td>0</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>00:12 ??</td>
<td>00:00.39 /usr/sbin/securityd -i</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>1</td>
<td>0</td>
<td>00:20 ??</td>
<td>00:00.28 /usr/sbin/ntpd -c /private/etc/ntp-restrict.conf -n -g -p /var/run/ntpd.pid -f /var/</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>1</td>
<td>0</td>
<td>04:46 ??</td>
<td>04:46 /usr/sbin/update</td>
</tr>
<tr>
<td>0</td>
<td>66</td>
<td>1</td>
<td>0</td>
<td>00:09 ??</td>
<td>00:09.01 /System/Library/Frameworks/CoreServices.framework/Frameworks/MetaData</td>
</tr>
<tr>
<td>0</td>
<td>70</td>
<td>1</td>
<td>0</td>
<td>24.78 ??</td>
<td>1:03.40 /System/Library/Frameworks/CoreServices.framework/Frameworks/MetaData</td>
</tr>
</tbody>
</table>

ata.framework/Supp

501 71 1 0 03:45 ??

indow console

| 0   | 72  | 1    | 0   | 00:00 ??  | 00:00.00 /usr/sbin/KernelEventAgent |
| 0   | 73  | 1    | 0   | 00:14 ??  | 00:09.24 /usr/sbin/kdcmcmd -n -a |
| 0   | 75  | 1    | 0   | 07:76 ??  | 00:16.00 /usr/sbin/libexec/hidd |
| 0   | 76  | 1    | 0   | 01:39 ??  | 00:02.26 /System/Library/Frameworks/CoreServices.framework/Versions/A/Frameworks/CarbonCore.framework/CoreComponent |

works/CarbonCore.framework

| 0   | 78  | 1    | 0   | 00:03 ??  | 00:09.94 /sbin/system_profiler -F /private/var(vm).swapfile |
| 0   | 81  | 1    | 0   | 00:12 ??  | 00:09.28 /usr/sbin/diskarbitrator |
| 0   | 85  | 1    | 0   | 00:06 ??  | 00:09.09 /usr/sbin/blued |
| 0   | 86  | 1    | 0   | 00:09 ??  | 00:09.09 /usr/sbin/autofsd |
| 0   | 88  | 1    | 0   | 01:13 ??  | 00:03.27 /usr/sbin/libexec/ApplicationFirewall/socketfilterfw |
| 0   | 97  | 73   | 0   | 00:01 ??  | 00:08.02 /usr/sbin/krb5kdc -r -LKDC:SHA1.97822AEFBD469488FD9135C6C96FDA8 |

77674486

| 0   | 100 | 1    | 0   | 00:57 ??  | 00:01.17 /System/Library/CoreServices/coreservicesd |
| 0   | 102 | 1    | 0   | 56.28 ??  | 1:52.59 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/Frameworks/|
```
Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process’ resources are deallocated by operating system

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination
Parents and child processes

- Ever login to a remote machine, run a process in the background and log out?
  - What happened?

- Most of the time the process in the background dies.
  - Or, the shell won’t let you logout.

- The “nohup” command lets you spawn a child and tell it to not go away if the parent does.
  - It tells the child to ignore the “SIGHUP”, or hang-up, signal.
  - We talk about signals in chapter 4 in more detail (next lecture).
Termination and error codes

- One way to think about a process spawned by a parent is as an asynchronous subroutine call.
  - Child executes specialized code.
  - Asynchronous execution means parent can do other things while child works.
  - Eventually the child says `exit()`, and specifies a return code.
  - This return code is handed back to the parent to examine and see how the child did.

- In most instances the return code is used to indicate success or failure of a program.
  - Integer return code allows for more than boolean information when indicating failure.
  - This allows parent to infer a cause of failure.
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (IPC)
- Two models of IPC
  - Shared memory
  - Message passing
Communications Models

(a) process A
    process B
    kernel

(b) process A
    shared
    process B
    kernel

1 2
Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Producer-Consumer

- Processes cooperate by sharing information.
- Name implies a direction.
  - Producer creates data.
  - Consumer consumes data.

- The channel between them is implemented as a buffer.
  - *unbounded-buffer*: no limit placed on size of buffer.
    - May experience runtime overflow errors though.
  - *bounded-buffer*: limit placed on buffer size.
Bounded-Buffer – Shared-Memory Solution

- Shared data

  ```c
  #define BUFFER_SIZE 10
  typedef struct {
      . . .
  } item;
  
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

- Solution is correct, but can only use BUFFER_SIZE-1 elements
while (true) {
    /* Produce an item */
    while ( ((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing -- no free slots */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
Bounded-Buffer – Consumer

```java
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```
Danger

- Shared memory is quite dangerous to work with.
  - Easy to make an error with coordination.
  - Nondeterministic execution due to error can make debugging a nightmare.

- We will spend a whole chapter on synchronization and concurrency control methods in ch. 6.

- Message passing is a slightly safer way to build programs that share information, although at a cost.
  - Potential performance degradation relative to sh. mem.
Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables

- IPC 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 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 \(Q\)

  - Exhibits symmetry: both sides explicitly state who the other side is.

- 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:
  \( \text{send}(A, \text{message}) \) – send a message to mailbox A
  \( \text{receive}(A, \text{message}) \) – 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 is notified who the receiver was.
Direct vs indirect

- Direct:
  - Requires prior agreement on naming so sender and receiver can know about each other.
  - Asymmetric situation (i.e., receive doesn’t say who from) may require additional data in messages if receiver needs to know who sent data.

- Indirect:
  - Potential contention for mailboxes.
  - Mailboxes are a finite resource. Possible (although unlikely) to exhaust OS resources.
Direct vs. Indirect

- Traditional socket communications adopt the direct approach.
  - Say who send/receive pairs are, possibly use “accept()” to dynamically determine sender on receiver side.

- An interesting language recently gaining attention is Erlang.
  - Based on the “Actor model” of concurrency.
  - Each process has mailboxes, messages are sent between mailboxes.
  - Processes are “Erlang processes”, not necessarily OS processes.
    - They live in a virtual machine that most likely uses OS threads for concurrency.
-module(simple).
-export([loop/0, doSomething/2, starter/0]).

starter() ->
    spawn(simple, loop, []).

doSomething(Pid,Request) ->
    Pid ! {self(), Request},
    receive
        Response -> Response
    end.

loop() ->
    receive
        {From, X} ->
            From ! X+1,
            loop()
    end.
A little Erlang

Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.5 (abort with ^G)
1> c(simple).
{ok,simple}

2> Pid=simple:starter().
<0.38.0>

3> Pid!{self(),1}.
{<0.31.0>,1}

4> receive
4>   X -> X
4> end.

2
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
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
Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    \[
    \text{segment id} = \text{shmget}(\text{IPC PRIVATE, size, S IRUSR | S IWUSR});
    \]
  - Process wanting access to that shared memory must attach to it
    \[
    \text{shared memory} = (\text{char} *) \text{shmat}(\text{id}, \text{NULL}, 0);
    \]
  - Now the process could write to the shared memory
    \[
    \text{sprintf}\left(\text{shared memory, "Writing to shared memory"}\right);
    \]
  - When done a process can detach the shared memory from its address space
    \[
    \text{shmdt}(\text{shared memory});
    \]
Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation - Kernel and Notify
    - Used by the OS to communicate with processes
  - Only three system calls needed for message transfer
    - `msg_send()`, `msg_receive()`, `msg_rpc()`
      - Send/recv for messages; RPC for remote procedure call
  - Mailboxes needed for communication, created via
    - `port_allocate()`
Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem’s connection port object
    - The client sends a connection request
    - The server creates two private communication ports and returns the handle to one of them to the client
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies
Local Procedure Calls in Windows XP

Client

Connection request

Connection Port

Handle

Server

Handle

Client Communication Port

Handle

Server Communication Port

Shared Section Object (<= 256 bytes)
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

- These address the problem of allowing processes to communicate and coordinate with each other *without* being required to live on the same machine.
  - Sockets are the bread and butter of our internet apps.
  - RPC/RMI are also common in distributed systems for providing an abstraction layer above raw sockets.
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
Socket Communication

host $X$
(146.86.5.20)

socket
(146.86.5.20:1625)

web server
(161.25.19.8)

socket
(161.25.19.8:80)
RPC and RMI

- RPC and RMI systems provide an abstraction layer above the mechanism by which two processes are connected.

- Can give the illusion of function invocation.
  - Client says “foo()”, and RPC/RMI system hides the fact that the execution of foo() is a sequence of messages.
  - Implementation of “foo()” is written as a procedure, and is called by the RPC/RMI system as though it was a normal call.

- Obviously there are limitations.
  - Passing pointers can be tricky or prohibited.
  - Error situations not possible in a single program can appear.
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
Execution of RPC

- **User Call:**
  - User calls kernel to send RPC message to procedure X

- **Kernel Action:**
  - Kernel sends message to matchmaker to find port number
  - Kernel places port P in user RPC message
  - Kernel sends RPC
  - Kernel receives reply, passes it to user

- **Matchmaker:**
  - Receives message, looks up answer
  - Matches to server
    - From: server
      - To: client
      - Port: kernel
      - Re: RPC X
      - Port: P
    - Matchmaker replies to client with port P
  - Daemon listening to port P receives message
  - Daemon processes request and processes send output

- **Server Action:**
  - From: client
    - To: server
    - Port: P
    - Re: <contents>
  - Server sends reply to client
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.
Marshalling Parameters

client

val = server.someMethod(A,B)

stub

remote object

boolean someMethod (Object x, Object y)
{
    implementation of someMethod
    ...
}

skeleton

A, B, someMethod

boolean return value
Further details on RMI/RPC

- These are typically a topic for a distributed systems course.
  - CIS630 for people entering the graduate program here next year.

- For those not going on to CIS630 (or equivalent somewhere else), and curious for more, speak to me about getting access to materials on the topic.
  - I can provide lecture notes from my CIS630 class this past fall.
End of Chapter 3