Logistics

- Project 0 due tomorrow
- Project 1 to be posted by COB today
  - Briefly discuss at the end of lecture
Outline

- Brief introduction to process scheduling
- Interprocess communication
- Remote procedure calls
Context Switch
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - *Job queue* – set of all processes in the system
  - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
  - *Device queues* – set of processes waiting for an I/O device
- Processes migrate among the various queues
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling

- Process scheduling queueing diagram represents:
  - queues, resources, flows

- Processes move through the queues
Schedulers

- **Short-term scheduler (or CPU scheduler)**
  - Selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)
    - must be fast because involves context switching

- **Long-term scheduler (or job scheduler)**
  - Selects which programs should be launched
    - program is not yet running as a process
    - once a program is launched, its process is brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)
    - does not have to be fast
  - The long-term scheduler controls the degree of multiprogramming
  - It is possible that a process can be removed temporarily
Process Mix

Processes can be described as either:

- I/O-bound process
  - spends more time doing I/O than computations
  - many short CPU bursts

- CPU-bound process
  - spends more time doing computations
  - few very long CPU bursts

- There can be sub-categories of these

- Long-term scheduler strives for good process mix
Addition of Medium Term Scheduling

- Medium-term scheduler can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution
  - Refer to this as job swapping
Interprocess Communication

- Generally think of processes within a system as being independent or cooperating
  - 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
- An independent process does not fork any child processes or interact with other processes, including its parent
- Cooperating process typically interact with other processes through coordinate actions and including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
Process Communication

- Process cooperation is a fundamental aspect of an OS and of the computing environment it is maintaining on behalf of executing applications.
- Processes need to interact and share information.
- Process model is a useful way to isolate running programs (separate resources, state, and so on):
  - Can simplify programs (no need to worry about other processes running).
  - But processes do not always work in isolation.
- How to support process interoperation, communication, and sharing of information.
- Discuss a variety of ways:
  - Not talk about sharing of files or signals.
Process Communication (Interoperation)

- When is communication necessary?
- Lots of examples in operating systems
  - Threads with access to same data structures
  - Kernel/OS access to user process data
  - Processes sharing data via shared memory
  - Processes sharing data via system calls
  - Processes sharing data via file system
- In general, there are numerous examples in computer science where interoperation is important
  - DB transactions, P/L parallelism issues
**IPC Mechanisms**

- Interprocess communication (IPC) supports the exchange of data between processes.

- Two fundamental methods:
  - Shared memory
    - pipes, shared buffer
  - Message Passing
    - mailboxes, sockets

- Which one would you use and why?

- Depends on the application.
Consider two processes sharing a memory region
- Producer writes
- Consumer reads

Producer action
- While the buffer not full …
- … stuff can be written (added) to the buffer

Consumer actions
- When stuff is in the buffer …
- … it can be read (removed)

Must manage where new stuff is in the buffer

Can think of the buffer as being
- Bounded (Problems?)
- Unbounded (Realistic?)
Shared Memory -- Producer

```c
item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out) ; /* do nothing ... buffer is full */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Assume that the consumer is modifying the `out` variable. Do you see any problems here?
Shared Memory -- Consumer

item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing ... buffer is empty */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}

Circular buffer

Assume that the producer is modifying the in variable. Do you see any problems here?
IPC with Shared Memory

- Communicate by reading/writing from a specific memory location
  - Setup a shared memory region (segment) in your process
  - Permit others to attach to the shared memory region

- `shmget(2)` -- create shared memory segment
  - Permissions (key) (read and write)
  - Size
  - Returns an identifier for segment

- `shmat(2)` -- attach to existing shared memory segment
  - Specify identifier
  - Location in local address space
  - Permissions (read and write)

- Also, operations for detach and control
**IPC with Pipes**

- Producer-Consumer mechanism for data exchange
  - prog1 | prog2 (shell notation for pipe)
  - Output of prog1 becomes the input to prog2
  - More precisely, a connection is made so that the *standard output* of prog1 is connected to *standard input* of prog2

- OS sets up a fixed-size buffer
  - System calls: `pipe(2)`, `dup(2)`, `popen(2)`

- Producer
  - Write to buffer, if space available

- Consumer
  - Read from buffer if data available
Management of Pipes

- **Buffer management**
  - A finite region of memory (array or linked-list)
  - Wait to produce if no room
  - Wait to consume if empty
  - Produce and consume complete items

- **Access to buffer**
  - Write adds to buffer (updates end of buffer)
  - Reader removes stuff from buffer (updates start of buffer)
  - Both are updating buffer state

- **Issues**
  - What happens when end is reached (e.g., in finite array)?
  - What happens if reading and writing are concurrent?
  - Who is managing the pipe?
Shared Memory Machines

- SGI UV (Ultra Violet) 1000 (Pitt SC)
  - 256 blades, each with 2 8-core Xeon processors
  - Each core has 8 GB RAM = 128 GB per blade

- Coherent shared-memory machine = all memory accessible to the machine
  - 32 TB of RAM

- Why? Certain problems hard to chunk up
  - Example: graphs
We are interested in mechanisms for processes to communicate and to synchronize their actions.

Messaging system
- Processes communicate with each other without resorting to shared variables
- Use messages instead

Establish communication link
- Producer sends on link
- Consumer receives on link

IPC Operations
- Y: Send(X, message)
- X: Receive(Y, message)

Issues
- What if X wants to receive from anyone?
- What if X and Y aren’t ready at same time?
- What size message can X receive?
- Can other processes receive the same message from Y?
Synchronous Messaging

- Direct communication from one process to another
- Synchronous send
  - `Send(X, message)`
  - Producer must wait for the consumer to be ready to receive the message
- Synchronous receive
  - `Receive(id, message)`
  - Id could be X or anyone
  - Wait for someone to deliver a message
  - Allocate enough space to receive message
- Synchronous means that both have to be ready!
Properties of Communication Links

- In direct communication:
  - 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 bidirectional
Asynchronous Messaging

- Indirect communication from one process to another
- Asynchronous send
  - \( \text{Send}(M, \text{message}) \)
  - Producer sends message to a buffer \( M \) (like a mailbox)
  - No waiting (modulo busy mailbox)
- Asynchronous receive
  - \( \text{Receive}(M, \text{message}) \)
  - Receive a message from a specific buffer (get your mail)
  - No waiting (modulo busy mailbox)
  - Allocate enough space to receive message
- Asynchronous means that you can send/receive when you’re ready
  - What are some issues with the buffer?
Properties of Communication Link

- In indirect communication:
  - 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
Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send
    - sender is blocked until the message is received
  - Blocking receive
    - receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send
    - sender sends the message and continue
  - Non-blocking receive
    - receiver receives: a valid message or a null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous
**IPC with Sockets**

- Communication end point
  - Connect one socket to another (TCP/IP)
  - Send/receive message to/from another socket (UDP/IP)

- Sockets are named by
  - IP address (roughly, machine)
  - Port number (service: ssh, http, etc.)

- Semantics
  - Bidirectional link between a pair of sockets
  - Messages: unstructured stream of bytes

- Connection between
  - Processes on same machine (UNIX domain sockets)
  - Processes on different machines (TCP or UDP sockets)
  - User process and kernel (netlink sockets)
Files and File Descriptors

- POSIX system calls for interacting with files
  - `open()`, `read()`, `write()`, `close()`
  - `open()` returns a file descriptor
    - an integer that represents an open file
    - inside the OS, it’s an index into a table that keeps track of any state associated with your interactions, such as the file position
    - you pass the file descriptor into `read`, `write`, and `close`
  - File descriptors are kept as part of the process information in the process control block
Networks and Sockets

- UNIX likes to make all I/O look like file I/O
  - The good news is that you can use `read()` and `write()` to interact with remote computers over a network!

- File descriptors are used for network communications
  - The socket is the file descriptor

- Just like with files....
  - Your program can have multiple network channels (sockets) open at once
  - You need to pass `read()` and `write()` the socket file descriptor to let the OS know which network channel you want to use
Examples of Sockets

- HTTP / SSL
- email (POP/IMAP)
- ssh
- telnet
**IPC and Sockets**

Client X: 10.12.3.4
Socket: 10.12.3.4:5544

Internet

Socket: 128.95.4.33:80
Web server: 128.95.4.33

Socket: 128.95.4.33:80

Socket: 10.12.3.4:5544

Client Y: 44.1.19.32
Socket: 44.1.19.32:7113
**File Descriptors**

**Web server**

- fd 5
- fd 8
- fd 9
- fd 3

**Internet**

- 128.95.4.33
- 10.12.3.4:5544
- 44.19.32:7113

**Client**

- 44.19.32:7113

**OS’s file descriptor table**

<table>
<thead>
<tr>
<th>file descriptor</th>
<th>type</th>
<th>connected to?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pipe</td>
<td>stdin (console)</td>
</tr>
<tr>
<td>1</td>
<td>pipe</td>
<td>stdout (console)</td>
</tr>
<tr>
<td>2</td>
<td>pipe</td>
<td>stderr (console)</td>
</tr>
<tr>
<td>3</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 44.19.32:7113</td>
</tr>
<tr>
<td>5</td>
<td>file</td>
<td>index.html</td>
</tr>
<tr>
<td>8</td>
<td>file</td>
<td>pic.png</td>
</tr>
<tr>
<td>9</td>
<td>TCP socket</td>
<td>local: 128.95.4.33:80 remote: 102.12.3.4:5544</td>
</tr>
</tbody>
</table>
Types of Sockets

- **Stream sockets**
  - For connection-oriented, point-to-point, reliable bytestreams
    - uses TCP, SCTP, or other stream transports

- **Datagram sockets**
  - For connection-less, one-to-many, unreliable packets
    - uses UDP or other packet transports

- **Raw sockets**
  - For layer-3 communication
    - raw IP packet manipulation
Stream Sockets

- Typically used for client / server communications
  - But also for other architectures, like peer-to-peer

- Client
  - An application that establishes a connection to a server

- Server
  - An application that receives connections from clients
Datagram Sockets

- Used less frequently than stream sockets
  - They provide no flow control, ordering, or reliability
- Often used as a building block
  - Streaming media applications
  - Sometimes, DNS lookups
Issues using Sockets

- Communication semantics
  - Reliable or not

- Naming
  - How do we know a machine’s IP address? DNS
  - How do we know a service’s port number?

- Protection
  - Which ports can a process use?
  - Who should you receive a message from?
    - Services are often open -- listen for any connection

- Performance
  - How many copies are necessary?
  - Data must be converted between various data types
Remote Procedure Calls (RPC)

- IPC via a procedure call
  - Looks like a “normal" procedure call
  - However, the called procedure is run by another process
    - maybe even on another machine

- RPC mechanism
  - Client stub
  - “Marshall" arguments
  - Find destination for RPC
  - Send call and marshalled arguments to destination
  - Server stub
  - Unmarshalls arguments
  - Calls actual procedure on server side
  - Return results (marshall for return)
Remote Procedure Calls

user calls kernel to send RPC message to procedure X

kernel sends message to matchmaker to find port number

matchmaker receives message, looks up answer

matchmaker replies to client with port P

daemon listening to port P receives message

daemon processes request and processes send output

From: client
To: server
Port: kernel
Re: address for RPC X

From: server
To: client
Port: kernel
Re: RPC X
Port: P

From: client
To: server
Port: port P
<contents>

From: RPC
Port: P
To: client
Port: kernel
<output>

kernel places port P in user RPC message

kernel sends RPC

kernel receives reply, passes it to user
Remote Procedure Calls

- Supported by systems
  - Java RMI
  - CORBA

- Issues
  - Support to build client/server stubs and marshalling code
  - Layer on existing mechanism (e.g., sockets)
  - Remote party crashes... then what?

- Performance versus abstractions
  - What if the two processes are on the same machine?
Remote Procedure Calls

- Marshalling

```java
val = server.someMethod(A,B)
```

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

Diagram:

- Client
  - val = server.someMethod(A,B)
  - Stub
  - A, B, someMethod
  - boolean return value

- Remote object
  - boolean someMethod (Object x, Object y)
  - Skeleton
MapReduce

- Distributed computing framework for working on large data sets on compute clusters
- Divide data into subset that are “mapped” to each node involved in computation
- Collect all sub-problem answers and “reduce” to form the final output
- Uses:
  - Distributed sort and grep
  - Graph reversal and search
  - Statistical analysis and web analytics, bioinformatics
MapReduce: Word Count

Input
- Deer Bear River
- Car Car River
- Deer Car Bear

Split
- Deer Bear River
- Car Car River
- Deer Car Bear

Map
- Deer, 1
- Bear, 1
- River, 1

Shuffle
- Bear, 1
- Bear, 1

Reduce
- Bear, 2
- Car, 3
- Deer, 2
- River, 2

Final
- Bear, 2
- Car, 3
- Deer, 2
- River, 2
void map(String name, String document):
    // name: document name
    // document: document contents
    for each word w in document:
        EmitIntermediate(w, "1");

void reduce(String word, Iterator partialCounts):
    // word: a word
    // partialCounts: a list of aggregated partial counts
    int sum = 0;
    for each pc in partialCounts:
        sum += ParseInt(pc);
    Emit(word, AsString(sum));

Concepts come from functional programming
(pay attention in CIS 425!)
Hadoop & Map/Reduce

WordCount.java

```java
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.*;

public class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter
           reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());  /* splits lines into words */
                output.collect(word, one);
            }
        }

        public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
            public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter
                reporter) throws IOException {
                int sum = 0;
                while (values.hasNext()) {
                    sum += values.next().get();  /* sums all the collected words */
                }
                output.collect(key, new IntWritable(sum));
            }
        }
    }
}
```
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class); /* collects all values together */
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}

Scalable framework: works on single-node machine, “pseudo-distributed” (single machine, multiple processes), or fully distributed cluster (depending on how Hadoop installation is set up)
IPC Summary

- Lots of mechanisms
  - Pipes
  - Shared memory
  - Sockets
  - RPC

- Trade-offs
  - Ease of use, functionality, flexibility, performance

- Implementation must maximize these
  - Minimize copies (performance)
  - Synchronous vs Asynchronous (ease of use, flexibility)
  - Local vs Remote (functionality)
Summary

- **Process**
  - Execution state of a program

- **Process Creation**
  - fork and exec
  - From binary representation

- **Process Description**
  - Necessary to manage resources and context switch

- **Process Scheduling**
  - Process states and transitions among them

- **Interprocess Communication**
  - Ways for processes to interact (other than normal files)
Project 1 – Ghost of the MCP

- Master Control Program (MCP)
- Create a MCP that can schedule processes from a workload of programs
- Due 11:59 pm, October 31 (Halloween night)
- Will post later this evening
Next Class

- Threads