Outline

- More process scheduling
- Interprocess communication
- Remote procedure calls
- Project 1

Reminders
- Assignment 1 due October 17
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

![Diagram of queues and process control blocks (PCBs)]
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)

- Long-term scheduler (or job scheduler)
  - Selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the degree of multiprogramming
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
- 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: swapping
Processes within a system may be 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

Cooperating process can affect or be affected by other processes, including sharing data

Reasons for cooperating processes:
- Information sharing
- Computation speedup
- Modularity
- Convenience
Process Communication

- Processes need to share information
- Process model is a useful way to isolate running programs (separate resources, state, etc)
  - Can simplify programs (no need to worry about other processes running)
  - But processes don’t always work in isolation
- Discuss a variety of ways
  - Does not include regular files and signals
Process communication

- 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
- And in general computer science
  - 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?

![Diagram](image.png)
Shared Memory

- Two processes share a memory region
  - One writes: Producer
  - One reads: Consumer

- Producer action
  - While buffer not full
  - Add stuff to buffer

- Consumer actions
  - When stuff in buffer
  - Read it

- Must manage where new stuff is in the buffer...
Shared Memory -- Producer

item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out) 
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
Shared Memory -- Consumer

```c
item nextConsumed;

while (1) {
    while (in == out) {
        /* do nothing */
        nextConsumed = buffer[out];
    }
    out = (out + 1) % BUFFER_SIZE;
}
```
Shared Memory

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

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

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

- Also, operations for detach and control
Pipes

- Producer-Consumer mechanism
  - `prog1 | prog2`
  - The output of `prog1` becomes the input to `prog2`
  - More precisely,
    - standard output of `prog1` is connected to standard input of `prog2`

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

- Producer
  - Write to buffer, if space available

- Consumer
  - Read from buffer if data available
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?
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
IPC -- Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- 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?
**IPC -- 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 bi-directional
**IPC -- Asynchronous Messaging**

- Indirect communication from one process to another
- Asynchronous send
  - `Send(M, message)`
  - Producer sends message to a buffer M (like a mailbox)
  - No waiting (modulo busy mailbox)
- Asynchronous receive
  - `Receive(M, 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 -- the sender is blocked until the message is received
    - Blocking receive -- the receiver is blocked until a message is available
  - Non-blocking is considered asynchronous
    - Non-blocking send -- the sender sends the message and continues
    - Non-blocking receive -- the receiver receives:
      - A valid message, or
      - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous
**IPC -- 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

- Remember open, read, write, and close?
  - POSIX system calls for interacting with files
  - `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
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!
- Just like with files....
  - your program can have multiple network channels open at once
  - you need to pass `read()` and `write()` a file descriptor to let the OS know which network channel you want to write to or read from
- A file descriptor used for network communications is a socket
Examples of sockets

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

- **Client X**: 10.12.3.4
- **Socket**: 10.12.3.4:5544
- **Client Y**: 44.1.19.32
- **Socket**: 44.1.19.32:7113
- **Web Server**: 128.95.4.33
  - **Socket**: 128.95.4.33:80

Connects through the Internet.
Web server

<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: 44.1.19.32:7113</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>

OS's descriptor 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

1. create socket
2. communicate
IPC -- Sockets

- Issues
- 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

- 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

1. User calls kernel to send RPC message to procedure X.
2. Kernel sends message to matchmaker to find port number.
3. Matchmaker receives message, looks up answer.
4. Matchmaker replies to client with port P.
5. Kernel places port P in user RPC message.
6. From: server To: client Port: kernel Re: RPC X Port: P.
7. Daemon listening to port P receives message.
9. Daemon processes request and processes send output.
10. Daemon processes request and processes send output.
12. 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

client

val = server.someMethod(A,B)

stub

A, B, someMethod

remote object

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

skeleton

boolean return value
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 subproblem answer 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
- Car, 2
- River, 1

Shuffle

- Bear, 1
- Car, 2
- Deer, 1
- River, 1

Reduce

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

Final

- Deer, 1
- Bear, 1
- Car, 2
- River, 1
MapReduce

```java
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!)
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 – Master Control Program (MCP)

- Create a MCP that can schedule processes from a workload of programs
- Slightly less than 3 weeks to finish
- Will post later this evening
Next Class

- Threads