CS1550 Fall 2015

Cyrus Ramavarapu

May 11, 2016

Contents

Chapter 1

Overview

A brief overview of the course.

1.1 Operating Systems Manage Resources and Abstract Details

Resources: CPU Time, Memory, I/O Devices, Security

- Operating system needs its own resources to make decisions
- A layered structure between request and resources

Detail Abstraction: Sharing

- device context and method calling
- unifed interface for application devices
- exclusive access 1 process on a constrained computer (virtual memory)

Different types of OS open up choices in scheduling algorithms, etc depending on load, tasks, resources

- Mainframes
- Realtime Has dealines on tasks
  - **Soft** - (Can miss dealines) A dvd player and its FPS
  - **Hard** - if a deadline is missed, might as well have not tried (nuclear plant auto-pilot)
- Embedded A Car, A linking library to abstract I/O
- Server Linux is still a server class OS

We will look at medium sized systesm since they are constrained enough so we can't be naive, but they have enough resources so that we can share.
In computer science, often the same problem will be solved, historically, twice - Paradigm shifts
Computer time is expensive
Von Neumann Architecture - data and code occupied a unifed memory.
We will start with so few resources that we can't share.

Chapter 2

Basics and Terminology

2.1 How to support multiprogramming

Can I divide RAM?

- A process could access an address that is not in its RAM block
- Protect a block and check?

Where is the best place to do checks? **Hardware can help perform OS tasks** Do I pack tightly in memory? Many Processes but there are issues with **dynamic allocation**

2.2 A Process's Address Space

INSERT FIGURE OF ADDRESS SPACE
All address in range are your process's. None belong to another process.
A large region to support dynamic allocation.
Exclusive Access (This is not real)

- Does not make sense for same reason as dividing RAM – we don't have enough resources
- **The lie of virtual memory**

The OS is a layer between user and resources. Insert CS0449 Diagram. We make system calls from userspace to kernel space. All requests for a resources must go through the OS – you should **not** be able to side step OS.

- Hardware provides us with a Partioned Instruction Set (Some for Userspace, Some for Kernel Space)
- Some instructions are safe (add int)
- Some instructions are priveleged (kernel mode) – Idea of a **mode bit**

If you try a priveleged instruction in user mode, an **exception** is raised and OS sends a **signal** and terminates the process.

Syscalls are mechanistically different than function calls – mode changes. Can't jump and link to our different address space.

A syscall is an interrupt. In x86 you put a trap in eax. These are **software interrupts**. Later we will talk about **hardware interrupts**.

Interrupt Vector – Indexted by ints. When interrupt occurs, goes to correct index. Gets address (to syscall handler?)

Syscall table – grab new address for code of the syscall

OS only needs priveleged mode to change machine state. Not everything in OS is priveleged.

The OS is not the same as other processe. It does not compete for CPU time. It does not schedule itself. It is basically **pure overhead**.

The OS does not need to exist, but we are afraid a process may misbehave. As a result, the OS exists out of practical necessity. We don't really want this, but we have code (OS) and it needs resources. However, the OS only runs when it needs to: Reacting to events. This takes time.

Think back to CS0447 and assembly. On a function call, everything had to be returned to the original state. A clean up needed to be done. Similarly, the OS needs to make room for its code. The **caller context** state will have to be saved and restarted. **A context switch**.

A syscall does not save context. **The OS does it before the syscall**. Context will be saved it to RAM and put at the top of the caller's stack.

- Safe
    1. Code was interrupted (caller) – can't execute until OS returns
    2. RAM is a shared resource as a whole – address space abstraction, pieces of memory are mine

We believe that a single context switch is **optimized** (from a hardware/software end). Only way to go faster is to have fewer context switches. If we have two solutions and one uses **fewer** context switches, we will say it goes **faster**.

Resources to protect and share:

- CPU Time – Preemption
- Memory – Virtual Memory
- I/O – Spooling
- Security – *'Tis black magic. . . ' (Cyrus)*

Memory trade off – cost, speed, capacity

There are also hardware interrupts. However the actual action is that of a software interrupt. Think about a **bus signal**. It has some basic steps that allow the OS to react.

2.3 OS Design Schemes

There are two big types of OS Designs: **Monolithic** and **Microkernel**

2.3.1 Monolotic OS

INSERT FIGURE HERE
Think about the OS as another application which controls everything. **Monolithic Design** is how we normally write an application. In this design, the OS is priveleged. Consider scheduling: need a data structure, scheduling algorithm. This is a lot of code. In a monolithic design all of this can be done **without privelege**! Priveleged instructions came when you make context switch, set up memory space, etc. This low level state is is not doable by unpriveleged instructions.

In a Monolithic OS:

- Code to maintain scheduler code
- Priveleged code to do context work

All of this bundled together in a monolithic OS.

2.3.2 Microkernel OS

INSERT FIGURE HERE
In comparison, a microkernel OS strives to pare down OS size by extracting unpriveleged code into separate processes. **Servers** communicate with the microkernel to ge right answer. The microkernel then goes back and acts on it.

2.3.3 Which is better?

It depends.

- Context Switches: The microkernel makes more context switches. The monolithic kernel only needs to make 2.
- Code surface: The microkernel has less code – smaller attack surface. Also less code is easier to validate. Therefore system wide effects are less likely.
- Crashing: When a crash occurs the OS runs last. In a microkernel if a user server crashes, just need to pick another server.
- Speed: **A monolithic kernel is FAST**

Linux is a monolthic kernel. Windows (NT Line) is a sort of microkernel/monolthic hybrid.

2.3.4 Virtual Machines

Java has JVM between application and OS. Compartively, VMware runs as a guest OS (A different architecture). Between hardware - hypervisor. In all cases there is some notion of **resource management**. Virtual machines are not a new idea. They grant exclusive access and as a result came back because they could. We now have a lot of resources and can run many more systems. Think about a webserver.

Chapter 3

Processes

3.1 Handling Processes

Multiprogamming - a single program will not saturate a CPU. The OS needs to decide which process to run. The CPU seems to only get a continuous stream of instructions. INSERT FIGURE. Every time we stop a process we introduce work. We can do this work during I/O.

RAM does not get faster with more transistors. The improvement is linear in speed; however, RAM **capacity grows exponentially**.

Ahmdahl's Law – Diminishing returns in increasing speed. I/O is idle process time.

The hope is to interleave the waits of a process with the runs of another.

Process - a running program and its associated data.

Ready - A process is ready if it has everything it needs to run except CPU time.

Pseudo-Parallelism - Juggling processes. Blocked means you are just a choice for scheduling.

In a process life-cycle, a process can leave ready state by calling **exit()**. Processor time is fast in compared to I/O time. OS can block (A program is blocked wiating for I/O). Since the program is not ready, it is not a canidate for scheduling. OS can instead run another process. Hardware interrupt will tell OS that a blocked process got resources. The process then becomes ready.

Batch System - The only way to stop a process is to exit().

If you choose to block, you are vulnerable to programs taht neither exit or block. Consider the following: *jump: jmp jump* (infinite loop that does nothing). Greedy process with CPU time.

Remember: The OS is not a ready process that is scheduled. A greedy process starves even the OS from CPU time, but this is tied to the greedy process.

One solution is to create an even in the long program – a **Syscall**. Asumes programmer did this (**yield()** syscall for example). This gives us a **Cooperative multitasking system** which is not the ultimate goal. A lot depends on how the process is written.

We can't do this in code – do this in **hardware internal clock**. A hardware interrupt due to a time.

Taking a way a resource is called premption. This gives us a **Preemptive Multitasking System**.

3.2 How to choose a process

Table or Linked List containing:

- Process Management
  - State
  - Priority
  - PID
  - PPID
  - signal handlers
  - stats (start time/total CPU usage)
- File Management
  - File descriptor
  - Root directory
  - Current Working Directory
  - UID
  - GID
- Memory Management
  - Page table pointer
  - Pointers to text on text stack data

Thread – a stream of instructions and associated state. A thread is different than a process if we have more than one of them. Tasks can communicate between multiple separate threads (different processes) via traditional I/O OS.

3.3 User Threads and Kernel Threads

pthreads abstracts system threading implementation.
Kernel Threading – OS knows about threads.
User Threading – OS knows nothing about threads and only has a process table.
User threading depends upon library – **This is not pthreads**. Pthreads exists regardless of user/kernel threading.

If kernel threadign: pthread.create() is a syscall. If user threading: library that is linked against

Kernel threading costs context switches – therefore user threading may be faster.

Synchronization is not the problem right now

Issues with preemption:

- in kernel threading we can switch to a new thread or process (schedule)
- in user threading schedule can only choose between between processesses (the greedy thread does not know of preemption – kernel would have to do it and kernel can't see it). A processes can yield(). It only hurts one process (The one containing the greedy thread)
- in kernel threading pthread.yield() maybe a NOOP

In user threading we can customize scheduling decisions.

User Threading - a thread calls scanf and the whole process computation is blocked. However, in kernel threading other threads can keep working.

There is the nonblocking I/O or syscall. It immediately returns and more than likely your data is not ready.

In the user threading library, scanf can make a different syscall of the nonblocking type. The behavior of this syscall is along the lines of "try again or try again and I'll get it. This can be in the library. - the thread table is there so we can simulate "blocking" and switch to a different thread.

Examples:
Select() on a file descriptor, checks if the data is ready.
A videogame is drawing state with select and read, in a loop.

Select is a unix syscall. In linux there is poll() and epoll().
Between User threading and Kernel threading try to hybridize. **Scheduler Activation** (upcall) is pure hybrid because the OS is told what to do by process.

You need to have kernel threading and link user threading on top (Another hybrid aproach).
More Syscalls? Linux 2.6 had faster kernel threading than Linux 2.4 – due to a bad implementation.

Chapter 4

Scheduling

4.1 How to pick?

Scheduling is the process of choosing which of the **READY** processes/threads get to run next.

We can use the CPU intensely or be more I/O bound as a process:

- A CPU intensive process is **cpu bound**
- An I/O bound process will do a burst of CPU work and then I/O

Note: Both types of processes can have the same **wall clock run time**

If two process are CPU bound back to back, it will take $2t$ where t is 1 process time. It is hard/impractical to inerleave two CPU bound processes because switching processes is not free. If you dont switch properly you loose illusion of independent program.

A cpu bound program wil llikely get pre-empted often. This leads to loosing a processors time due to the switch. Running two CPU intesive programs at the "same" time may not be better than sequentially running. As mentioned above, it is not easy to schedule a bunch of CPU bound processes together.

In a premptive system, a process can get preempted and end up exiting (unlikely) or being blocked (I/O bound block a lot).

If the system is careful with the preemption timer, an I/O bound process may never see preemption – it will automatically block because it has to do I/O.

Can overlap CPU bursts and I/O blocks of two processes.
It only makes sense if I/O bound processes are much greater in number than CPU bound processes. **Reasonable: interactive programs tend to be I/O bound. Amhdahl's Law**.

Over time CPUs increase exponentially in speed (See Moore's law..although the quantum limit is rapidly being reached). On the other hand, I/O speeds grow linearly. As a result, a CPU bound program will in "time" become an I/O bound process because we gain the ability to do a lot of computation faster than we can read from a disk.

4.1.1 When to schedule?

- Software
  - Process Creation
  - Process Exit
  - Blocked
- Hardware Interrupts
  - I/O interrupt
  - clock interupt

4.1.2 Where to schedule?

We assume a **Von Neumann Architecture** where a process can be prevented to run by denying it RAM. Once a process is in RAM, CPU scheduler picks a process.

4.1.3 Classes of scheduling

Admission Schedule selects jobs to go into RAM. In an interactive system, this might be the user.

If admission scheduler did not do a great job, there is also a **Memory Scheduler** which can kick out a job from RAM. The problem is that a process may have made progress before the memory scheudler gets to it. The process has state that has to be saved. Therefore the memory scheduler will do a "temporary" eviction and hold the state of the process on disk. When system is better, the memory scheduler can bring process and its state back into RAM.

CPU scheduling – Batch scheduling can be used for non-interactive systems for jobs that can run overnight. (I like to think of scheduling jobs running on a cluster - Cyrus). This type of system can be preemptive.

4.2 Scheduling Algorithms

4.2.1 Metrics

Throughput – *Number of Jobs/Unit Time* or in Frequency (Hz)
Turnaround Time – Time from job submission to job completion. This is **not** execution time! The execution time is the time a process has **available** to run.
Average Turn Around Time – Average of all turn around times for a set of jobs.
Fairness – comparable processes get comparable service. Of course, comparable is not really defined. You can be egalitarian and say all processes are created equal... but are they?
Big-O – Asymptotic worst case run time. The key is that it is **Asymptotic**.
Implementation difficulty – we like easy things even if they are only close to perfect. Easy is also relative and hard to define.

4.2.2 Algorithm 1: First Come, First Serve

Queue	4	3	6	3	Runtime
	A	B	C	D	Process

After FCFS (first come, first serve):

Queue	4	3	6	3	Runtime
	A	B	C	D	Process

Such change, much wow! Also this algorithm is $\mathcal{O}(1)$ – so yay!
Throughput: $4\ jobs/16\ time = 1/4$
Average turn around: $40/4 = 10$

Process	Completion	Arrival	$\triangle t$
A	4	0	4
B	7	0	7
A	13	0	13
D	16	0	16
Sum			40

No matter what is the schedule, the through put is the same. We don't consider reality and delays.

In a batch system without premption jobs will run to completion and then another is chosen.

Calculating turn around time:

$$Turnaround\ Time\ =\ Finish\ -\ Available \text{ where } Finish\ =\ Execution\ Time\ +\ Wait$$

So first come, first serve is sort of crappy and boring. The only option is to alter wait time:

- 1st job has the shortest wait (0 time)
- 2nd job has the least if 1st is min

This brings us to algorithm two!

4.2.3 Algorithm 2: Shortest Job First

Queue	3	3	4	6	Runtime
	B	D	A	C	Process

Repeating the previous analysis for average turn around: 35/4.

Process	Completion	Arrival	$\triangle t$
A	10	0	10
B	3	0	3
A	6	0	6
D	16	0	16
Sum			35

There is no better way to lower average turn around time. This could also be easily implemented using a **minheap** which sorts in $\sim \mathcal{O}(nlog(n))$

Imagine a scenario with a long job that will never run if jobs added are always shorter. This is unfair (a process is starved). Theoretically this is an impossible thing to determine because the **Halting Problem** says we can't know *a priori* what will be the run time of a general process. However, the history of a program's execution could be used to *predict* how long a repeat run will take. Another technique could be to have the user input a time after which the OS will kill the process. If you over estimate the run time you loose optimal ATT (but who cares?).

Interactive Scheduling impatient users waiting.
Response Time Initiating an even → seeing response.

4.2.4 Algorithm 3: Round Robin Scheduling

$$\begin{gathered}\textit{5 Ready Processes}\\ A \rightarrow B \rightarrow C \rightarrow D \rightarrow E\end{gathered}$$

Idea: Schedule slices of each process in the same order. The time of a slice is a **Quantum**. The quantum is the maximum amount of time for operations to run.

As soon as a process blocks, the scheduler will pick the next process – **We are never idle**. As a result performance is heavily influenced by the choice of quantum.

We want processes to block themselves.

- Dont want to preempt right before a block
- Want to accomodate IO bound processes (CPU bound processes are screwed)

$$Click\ Button\ |-------------\triangle\ t-------------|\ WindowAppears$$

Worst case is $\triangle\ t * N$. Therefore want $Response\ Time\ \propto\ \triangle\ t * N$. Want response time smaller in an interactive system no admission/memory scheduler. Only can reduce $\triangle\ t$ in an interactive system.
Let $\triangle\ t\ =\ 1$ and $Context\ Switch\ (CS)\ =\ 1$

$$\text{Example: } |\triangle\ t\ |\ CS\ |\triangle\ t\ |$$

For every unit of useful work, two time units are needed (1 work, 1 CS). **50% CPU over head!**. Only way to improve is to increase the amount of useful work we do. This is due to the following relationship:

$$\frac{Useful\ Work}{Useful\ Work\ +\ CS}$$

We are pressured into both minimizing and maximizing $\triangle\ t$. There is probably no analytical solution, but we can use benchmarks which show that a quantum of $\sim 20\ -\ 60\ ms$ is adequate. Also assume it is statically coded into the program.

Priority Scheduling normally a number attached to each process. Think the unix command **nice**. Imagine there are 4 priority levels. A priority 4 job may get 4 times as much attention as a priority 1 job. One implementation of this system is to run the job for a quantum that is 4x as long as priority 1 process. It may just block during this time, so it does not guarantee the priority – a waste.

A better implementation would be to increase the frequency a process appears in the queue – as long at they are not lumped together (blocking problem again). One scheme to do this is that every time you run a priority level, go back and run all higher priority levels – avoids lumping together.

4.2.5 Other Scheduling Algorithms

- Shortest Process Next - Shortest Process First to an interactive system
- Guaranteed Scheduling
- Lottery Scheduling – random
- Fair Share

Shortest Process Next: Proceses run as long as user wants. Not predictive and seems a little crazy. Instead think of a process as a burst of computation. I/O has a short burst whereas CPU bound has a longer burst. The bias is towards I/O bound.

Guaranteed Scheduling: N processes get $1/N$ of the CPU time. This is done by looking at ratios of time and checking if any process is below the $1/N$ guarantee. This can happen if a process is I/O bound (uses less CPU then allotted). We have a preference for scheduling I/O bound processes sooner. Do we have to worry about starvation? As time passes, a new CPU bound job will eventually fall below $1/N$ if I/O bound processes are constantly being picked.

Fair share: Each user gets $1/N$ CPU time

Lottery Scheduling: Every process gets lottery tickets. Guaranteed scheduling has multiple implementations, lottery scheduling provides a **mechanism** by handing out tickets, picking a ticket, and scheduling the winner.

Guaranteed Scheduling is a Policy The rules a particular mechanism should follow. As seen above this can be implemented as just lottery scheduling.

We can also do fair share scheduling by lottery scheduling because a shell can be given N tickets that it hands out to processes. Additionally, priority can also be done by lottery scheduling because a higher priority process can be given more tickets. However, if order is key (round-robin) lottery will not be a good mechanism.

Real Time systems have deadlines, either hard or soft. A scheduling algorithm for real time systems is **Earliest Deadline First**.

- Real Time: how you do homework
- The difficulties arises due to dynamicity (think about juggling 5 projects)

What do real OSs do? All attemp to scheduling in $\mathcal{O}(1)$

Chapter 5

Interprocess Communication

5.1 Preliminaries

On a single processor system we can run at least two processes. When we have parallelism, it creates its own problems (ie. The Race Condition).

Note: *mmap* can be used to share an adress space between parent and child processes (See Project 2).

Preemption is trigged by a hardware interrupt – at any machine instruction boundary an interrupt can go off (For the more curious look up **precise** and **inprecise** interrupts.

Our job is to find regions of code where race conditions can exist – **critical regions**.

Example:

```
enter_critical_region(); // Mutex or Semaphore?
   A[tail++] = 20;
leave_critical_region(); // Mutex or Semaphore?

enter_critical_region() {
   disable_interrupts(); // This is wrong because it can be abused
                    // It is too coarse..preemption is not the
                    // real reason for race conditions
}

leave_critical_region() {
   enable_interrupts();
}
```

Imagine good pseudo-parallelism: $A \rightarrow C \rightarrow A$. C has nothing to do with A's shared data.

What we want:
Insert Figure Here
Goals:

- No two processes should be in CR at the same time
- No assumptions about cpu speed or number of CPUs
- No process outside its critical region may block another process (subtle point)
- No process should have to wait for ever to enter its critical region

These are the goals of *enter_critical_region()* and *leave_critical_ region()*

5.2 Implementation 1: Strict Alternation

Example:

```
/* This is from Process A Perspective */
int turn;

enter_CR() {
   while(turn != 0); /* Or 1 for Process B */
}

leave_CR() {
   turn = 1; /* Or 0 for Process B */
}
```

This does give us mutual exclusion and is known as **busy waiting**. Condition does not change unless process leaves CR.

INSERT FIGURE

Aside: See priority inversion problem.

Enforced alternation is a problem, so lets modify the above code a bit!

```
locked = 0;
enter_critical_region() {
   while(locked);
   /* There is no notion of possesion and no forced turn
    But this does not work! Since we can be pre-empted
    here. Can salvage the problem via hardware!
   */
    locked = 1;
}

leave_critical_region() {
   locked = 0;
}
```

Hardware Support can come from a single machine instruction (TSL, XCHG). For more information see Tannembaum. The problem win our previous lock system is preemption between test and set. Can there be a software solution to mutual excluson - Yes... **Peterson's Solution**.

5.3 Peterson's Solution

Share a global state with an array of size N for N processes. We will do 2. Consider if a process is interested and who requested last.

```
enter_critical_region(int process) {
   other = 1 - process;
   interested[process] = true; /* you are interested */
   last_request = process; /* without this you have a race condition (live lock) */
   /* a live lock is when two processes just pass possesion neither doing any work
    the last_request line is needed to break the tie */
   while (interested[other] == true && last_request == process);
   /* Conditional asks: Did i make the last request? */
}

leave_critical_region(int process) {
   interested[process]= false; /* no longer interested */
}
```

Peterson's solution works by exploiting the race condition. If the last request was not me, there was a preemption. I won if the other process overwrote my variable. I made first request if other made last. The key is that enter/leave cr functions are user space implementable mutual exclusion [Solves the same problem].

5.4 Producer and Consumer Problem

Shared buffered of fixed size. A potential race conditions exists in changing the counter (c++ and c–).

```
lw      $t0, 0(Address of Counter)
addi    $t0, $t0, 1
#PREMPTION
sw      $t1, 0(Address of Counter)
```

Counter can go from 3 to 4 to 4 as a result. In x86 one can do INC[counter]. More interestingly is the issue of a lost signal. Consumer runs first – empty buffer and should sleep(). Can be preempted right before we sleep. Producer runs, makes 1 , says wake up consumer...but signal is lost since the consumer is awake. Producer fills buffer, sleeps. Consumer is scheduled, sleeps. Both are a sleep and we have **Dead Lock**.

Race conditions occur when there is a multiple step update of shared global state.

How to (maybe) fix this:

```
/* Consumer */
enter_critical_region();
if (counter == 0)
   sleep();
leave_critical_region();

/* Producer */
enter_critical_region();
if (counter == n)
   sleep();
leave_critical_region();

enter_critical_region();
   counter++;
leave_critical-region();
```

This is even worse: guaranteed deadlock (sleep while holding lock) – condition variable. pthread_cord_signal will also reacquire mutex. Instead of blaming sleep() you can blame wake(). Keep a memory of wake ups and it becomes a problem in accounting. An int can keep track of this. This gives a **semaphore**.

5.5 Semaphores

Returning to the producer/consumer problem if you are already awake and get a weake up, maybe we should keep track of weake ups and sleeps.
If value is positive then wake ups without being asleep. If subtraction yields zero or negative, no one was trying to wake up so I should sleep (**down**). The negative cardinality of value is the number of sleepers.
If value on add is 0 or less, someone should wake up. This can be an arbitrary decision. The length of queue is the number of negatives. This can be thought of as the following conditional:

```
if (value <= 0)
   process = dequeue(process_array);
wake(process);
```

Put all of this into an object called a semaphore. They were invented by Dijkstra.
We can use 3 semaphores to solve producer/consumer problem.

- empty(n) – initialized by consumer
- full(0)
- mutex(1)

The mutex is used for mutual exclusion. Mutex.down and mutex.up surround our CR. Initially mutex has 1 – one missed wakeup. Start with down – we start our mutex at 1 which makes that which runs first enter first. Mutex.down is a subtraction and a potential sleep(). Lets say we get preempted. Consumer runs – downs mutex to -1 and sleeps. Producer runs again finishes CR ups mutex to 0 and therefore someone wakes up – the consumer. Consumer can now run.

Empty(n) and Full(0) – the value of a semaphore can be though of as the number of compies of a resource we can access simultaneously. Example lab with 5 printers, first 5 print jobs go and the 6th job has to wait. Empty(n) means I have n empty spaces [empty space is a resource consumed by something made]. Producer makes full spaces and consumes empty spaces. Consumer makes empty spaces and consumes full spaces. The interpretation between mutex and our empty and full are the same. Our 1 in mutex is number of access to critical region – only 1 process can access our critical region.

Semaphores are tricky – if you swap mutex lines you get a deadlock:

mutex.down	mutex.down
empty.down	full.down
producer	consumer (start here)

The result is that we can do mutual exclusion/synchronization without busy waiting.

5.5.1 Other synchronization primitives

Counting semaphore: A simplified version of a semaphore that can only be locked or unlocked.
Binary Semaphore: Semaphore that only takes the values 0 or 1 (a single bit). This is not a mutex. The first process to up will wake everyone else up. Also known as a **barrier**.
Monitors: Automate the process of acquisation of locks and bundle everything into one object. Original way to get synchronization in Java (eww) is to make a synchronized method. A method in Java being synchronized has to check lock on object. For producer/consumer while(true) needs to be outside.

There is a unifying bassis for all of our synchronization primitives – some amount of shared memory and uniform access.

Not all computers have this (distributed systems). Own OS, scheduler, etc. Can't depend on synchronization for reading/writing. Need to tell some how about CR status. Networks are unreliable.

Some solutions to this problem are:

- Elect a computer the leader. Shared state is moved from local to shared.
- Have nodes vote if neighbors in critical or if they entered. Ask only about $\sqrt{n}$

5.6 Why the fuss?

Why are we talking about this in operating systems? The semaphore can have a race condition with itself. You can't solve this with one huge hardware instruction (CISC instruction). Can't wrap this in petersons – you sleep with lock. Condition vars can be used, but it is the same problem. The shape of the problem is the same. You can, however, disable interrupts.

Move up() and down() into the operating system so it is the OS that is putting processes to sleep. In linux you will spinlock (not disable interrupts). OS code will/should run quickly so spin locking is not a problem. The spin lock is also around a very small amount of code – a best practice.