1. On x86, if the `EAX` register holds the value 0x712ab211, what value does the `AH` register have?

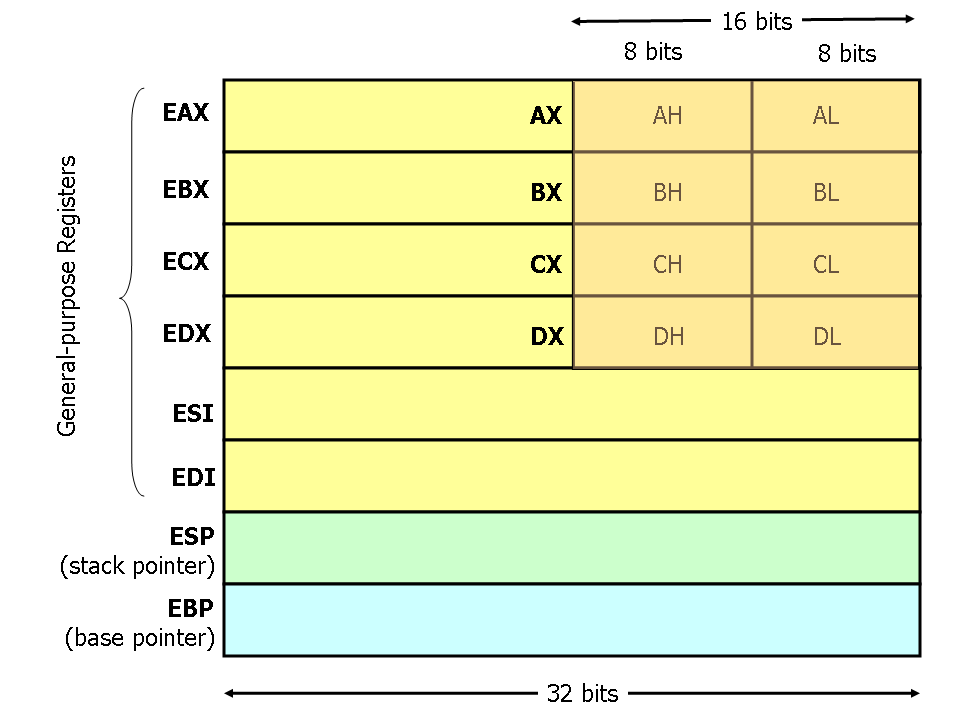
EAX is the full 32-bit value

AX is the lower 16-bits

AL is the lower 8 bits

AH is the bits 8 through 15 (zero-based)

* AX-B211
* AH-b2
* AL-11



B2

In the following assembly program, what is the value of the `EAX` register when the `done` label is reached?

start:

mov $0, %eax

jmp two

one:

mov $1, %eax

two:

cmp %eax, $1

je done

call one

mov $10, %eax

done:

jmp done

**1**

1. Below is the code for `fetchint` and `argint` in xv6:

// Fetch the int at addr from the current process.

int

fetchint(uint addr, int \*ip)

{

if(addr >= proc->sz || addr+4 > proc->sz)

return -1;

\*ip = \*(int\*)(addr);

return 0;

}

// Fetch the nth 32-bit system call argument.

int

argint(int n, int \*ip)

{

return fetchint(proc->tf->esp + 4 + 4\*n, ip);

}

Suppose we removed the check `(addr >= proc->sz || addr+4 > proc->sz)` (which, as you will recall, is there to guard against malicious user-space programs trying to crash the kernel or read memory they're not supposed to). Now, finish the following snippet of a malicious user-space program written in assembly so that it will crash the xv6 kernel:

**Mov $FFFFFFF**, %esp

// Your code here

mov $0x6, %eax ; kill(int pid) is system call 6

int $0x40 ; execute system call interrupt

Move the esp pointer to anywhere outside of the memory stack

1. Using the xv6 system calls below, write C code that creates a file named `hello.txt` and puts the string `hello world` into it. You do not need to write out the include statements or even a proper main function; just include the operations needed to open, write to, and close the file.

#define O\_RDONLY 0x000

#define O\_WRONLY 0x001

#define O\_RDWR 0x002

#define O\_CREATE 0x200

int open(char \*filename, int mode);

int write(int fd, void \*buf, int sz);

int close(int fd);

fd =open(“hello.txt”, O\_CREATE |

O\_WRONLY );

char buffer[] = “hello world”;

write(fd, buffer, strlen(buffer));

close (fd)

**Memory Management and Virtual Memory**

1. What is the difference between a physical address and a virtual address?

Real Memory uses Physical addresses. These are the numbers that the memory chips react on the bus. Virtual addresses are the logical addresses that refer to a process address space. A machine with 32 bits can generate address up to 4GB regardless of whether or not the machine has enough memory

1. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of:

\* 12 MB

\* 10 MB

\* 9 MB for first fit? Now repeat the question for best fit.

First Fit: 20MB, 10MB, 18MB

Best Fit: 12MB, 10MB, 9MB

1. Why is the principle of locality crucial to the use of virtual memory?

Processes exhibit a locality of reference, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. The set of pages that a process if currently using is it’s working set. If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase.

1. What does TLB stand for and what is its purpose

Translation Lookaside Buffer. Used in the MMU and consists of a small cache that contains entries that map virtual pages to physical page frames

1. Consider the following C program

Int X[N];

Int step = M; /\* M is some constant \*/

For (int i=0; I < N; i+= step) X[i] = X[i] + 1;

1. If this program is run on a machine with a 4KB page size and 64 entry TLB, what values of M and N will cause a TLB miss for every execution of the inner loop?

M has to be at least 4096 to ensure a TLB miss every time we access X. Since N affects only how many times X is accessed, any value of N will do.

1. Would your answer in part (a) be different if the loop were repeated many times? Explain

M should be at least 4096 to ensure a TLB miss for every access to an element of X. But now N should be greater than 256K

**Processes, Threads, and Scheduling**

1. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU. But when the process blocks, all entries better be removed from the list of runnable processes.

1. The register set is generally considered to be a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped, the registers must be saved. Multiprogramming thread is no different than multiprogramming processes, so each thread needs its own register save area

1. Describe the conditions that need to occur for a \*priority inversion\* bug to happen

The priority inversion bug occurs when a low-priority process is in its critical region and suddenly a high priority process becomes ready and is scheduled. If it uses busy waiting it will run forever. With user level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a high-priority thread to run. There is no preemption. With kernel level threads this problem can arise

**Drivers and I/O**

1. What problem does double buffering solve?

Having double the amount of space to store data in the kernel and being able to let the user process read data from one of the buffers while the second one is being filled

1. Suppose a printer prints one character at a time, and issues an interrupt when it is ready to print another. An interrupt handler for this device might look like:

// count: total bytes to be printed

// p: the data buffer containing data to print

// i: the index of the next byte to be sent to the printer

if (count == 0) {

unblock\_user();

} else {

\*printer\_data\_register = p[i];

count = count − 1;

i = i + 1;

}

acknowledge\_interrupt();

return\_from\_interrupt();

In this code, the interrupt is not acknowledged until after the next character has been output to the printer. Could it have equally well been acknowledged right at the start of the interrupt service procedure? If so, give one reason for doing it at the end. If not, why not?

If the printer can only print one character at a time, it probably only has space for one character so if it acks the interrupt before being ready to print another, it might run out of storage space and loose data

1. The rate at which a 300 dpi scanner produces data is 1 MB/sec. An 802.11b wireless network has a maximum transmission rate of 900KB/s. Can documents be sent out on the network as fast as they are scanned? Why or why not?

Even assuming double buffering, we can’t send them as fast because 900KB < 1 MB. We produce 1MB and we can only send 900Kb.

**Concurrency**

1. Suppose that we have an atomic compare-and-swap instruction that atomically compares a variable with some value and swaps them if they are not equal

int compare\_and\_swap(int \*var, int val);

Write implementations of `void acquire(int \*lock)` and `void release(int \*lock)` that use this instruction to implement a \*spin lock\* (that is, a lock that loops until it is able to acquire exclusive access to the lock). Note that we will assume here that each lock is represented by a global integer variable.

void acquire (int \*lock) {

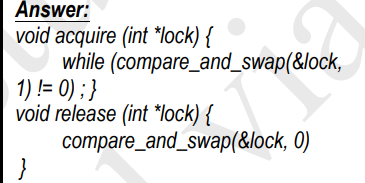
// your code here

}

void release (int \*lock) {

// your code here

}



1. Recall the parallel hashtable implementation from Homework 4:

#define NUM\_BUCKETS 5 // Buckets in hash table

typedef struct \_bucket\_entry {

int key;

int val;

struct \_bucket\_entry \*next;

} bucket\_entry;

bucket\_entry \*table[NUM\_BUCKETS];

// Inserts a key-value pair into the table

void insert(int key, int val) {

int i = key % NUM\_BUCKETS;

bucket\_entry \*e = (bucket\_entry \*) malloc(sizeof(bucket\_entry));

if (!e) panic("No memory to allocate bucket!");

e->next = table[i];

e->key = key;

e->val = val;

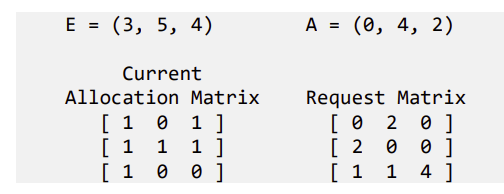
table[i] = e;

}

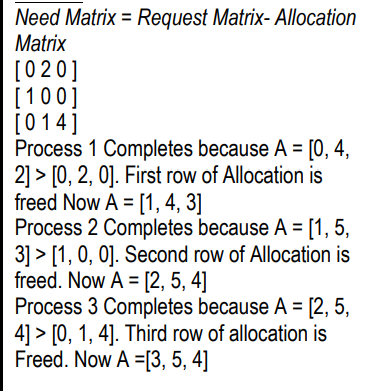
Suppose we have two threads inserting keys with `insert()` at the same time. Describe the exact sequence of events that results in a key getting lost.

Both of the threads set e->next = table[i] and/or table[i]=e. The threads will try to map to the same place in the same table while not seeing each other. They are not supposed to be in the same table. Therefore one key maybe seen and one may not be seen.

1. Consider the following allocation and request matrices, where E is the vector representing the resources of each type that exist in the system and A is the vector representing the resources currently available.



Is this system deadlocked? (Show how you arrived at that answer)



**Filesystems**

Suppose we have a non-journaled filesystem that uses i-nodes, and a file delete operation that consists of the following actions:

1. Mark the i-node for the file as free in the filesystem bitmap.
2. Mark the data blocks for the file as free in the filesystem bitmap.
3. Remove the directory entry for the file from the directory.

Now suppose that we have a crash after step 2.

1. Describe a scenario where this results in file data being corrupted.

Since event #3 above didn’t happen(remove directory entry) for file1, the entry for the file is still there when we reboot. However the block where the file lived, is marked as free so another file say file2 could come along and use this block. This would lead to file corruption

A new process opens a file and writes to this location. Because the inode is still there, the data will get confused. A check system can be used to check every inode if it is linked or not.

1. How would a filesystem checker like `fsck` that runs at boot detect and fix this condition

Check that all the directory entry files have corresponding i-nodes

Question 2.

In the xv6 logging filesystem, filesystem operations are grouped into transactions, where each transaction consists of the following operations

1. Write each modified block to the log area, along with its eventual destination.
2. Write a commit record.
3. For each entry in the log, copy the block to its final destination.
4. Clear the log.

For each of these steps, describe what would happen if the system crashed during that step, saying what xv6 would do when it reboots and how this would guarantee that the transaction is carried out atomically (that is, every operation is carried out, or none of them are).

Step 1: While you write log to memory, it crashes. If it crashed while you were writing logs into memory, the changes weren’t made. It doesn’t commit anything. Nothing happens

Step 2: What about if you commit and XV6 doesn’t know how much you changed? Nothing gets committed or changed because it doesn’t change anything because it doesn’t know what to change. If it crashes before writing a commit record, it doesn’t do anything.

Step 3: Copy the blocks to its final destination. Now you know what blocks you changed, and how many you changed. While saving, it crashes. You lose everything because you didn’t save it to disk. Will attempt to redo this step.

Step 4: Clear the log by setting the count field in the log header to 0. You already make all the changes. When the XV6 reboots, it sees that there are changes left over so it changes log header to 0. (but the changes are made).

Question 3

Suppose we have a filesystem with a block size of 512 bytes and an i-node defined as follows:

#define BLOCKSIZE 512

struct inode {

short type; // File type

short major; // Major device number (T\_DEV only)

short minor; // Minor device number (T\_DEV only)

short nlink; // Number of links to inode in file system

uint size; // Size of file (bytes)

uint blocks[32];

uint indirect;

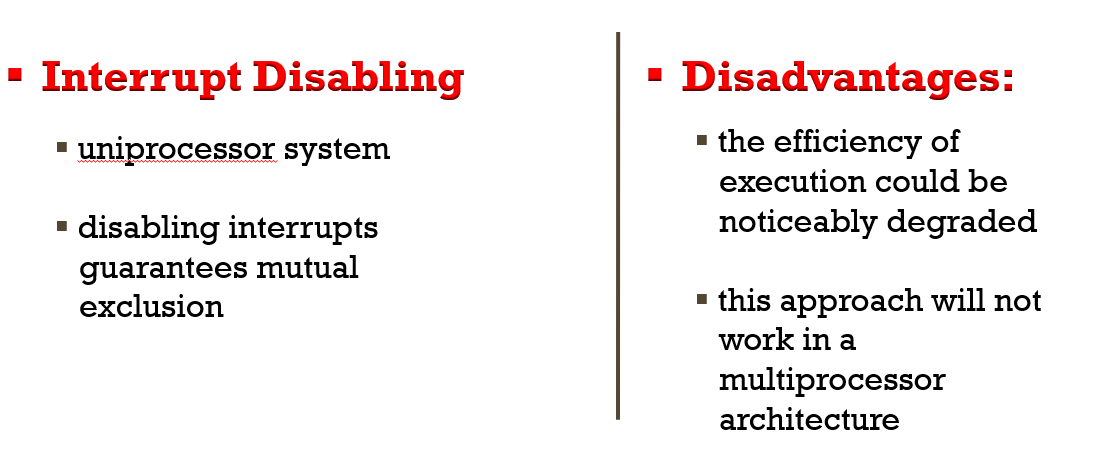
};

That is, it has 32 direct block pointers and one indirect block pointer. What size (in bytes) is the largest file we can create using this system?

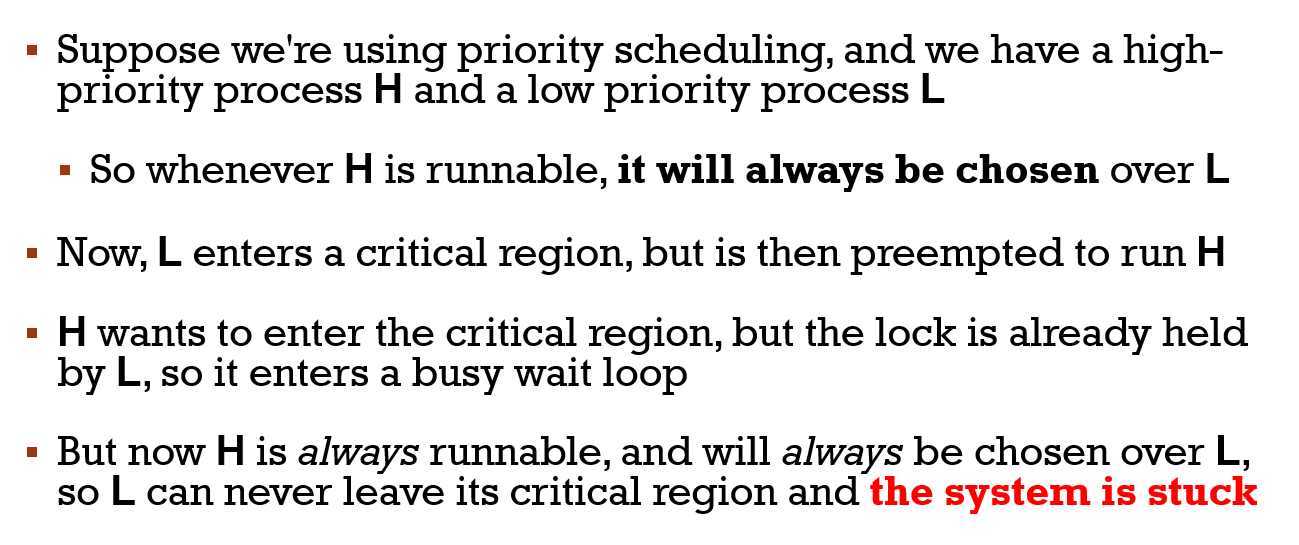
An int is 4 bytes. So assuming a 32 bit system (4 bytes per pointer) this would be 512/4 = 128 + 32 = 160 block pointers. 160 \* 512 = 80K.

**I/O Management, Disk Schedule and Drivers**

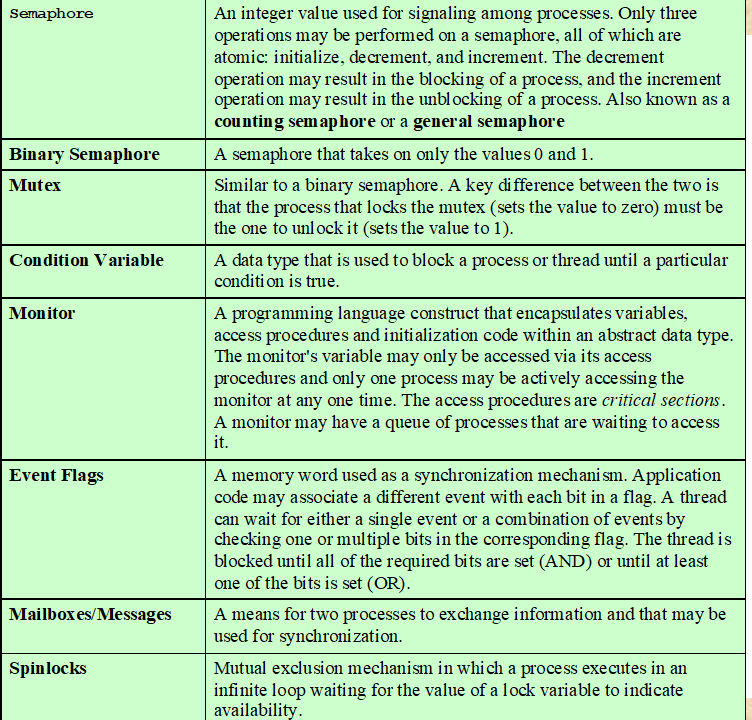
* Drivers and I/O
  + Code for handling interactiosn with hardware peripherals makes up a significant portion of most OSes
* Two categories of I/O Devices
  + Block Devices – Random Accewss
  + Can ask for any block of data with equal ease
  + Character Devices-Only support reading/writing streams
  + Not structured into blocks
* Device controllers-
  + Hardware peripherals tend to have both mechanical and electronic components (the device controller)
  + OS communicates with electronic components which controls the mechanical bits
* Device Drivers-Low level programs written b y hardware manufacturers which are loaded as kernel modules and provide the kernel with the knowledge on how to control the devices
* Device controllers – (hardware) understands software input. It translates software input into something a hardware device understands. Controllers sometimes have their own memory and their own CPU
  + Device drivers are purely software.
  + Device drivers communicate between the operating system and the physical device controller
* Firmware-Device controllers also run software and have their own operating systems
* I/O techniques:
  + Port I/O-
    - Fixed number of port addresses
    - Specialized CPU instructions (in and out) read and write bytes to these ports which are connected to hardware devices
    - Separate address spaces just for I/O
    - Need to have extra CPU instructions
    - Not generally accessible from C/C++ code
  + Memory Mapped I/O
    - Only one big address space
    - Certain portions of physical memory are assigned to various hardware devices
    - Reading from / writing to that address has the effect of receiving /sending data tot the peripheral
    - Sacrifices some physical address space
    - Can interact badly with memory caching
    - Each address must be checked to see if it is I/O
  + Interrupt-driven I/O
  + Direct Memory Access (DMA)
    - Ask the device to do a bulk transfer to memory
    - Goes directly from the device to RAM so the CPU can run concurrently
    - Communication between devices and RAM is mediated by the DMA controller
    - Uses programmed I/O to configure the DMA controller
    - DMA controller independently fetches bytes from the disk into main memory and then notifies the CPU when the transfer is complete
* Interrupt-When hardware devices need attention for any reason they raise an interrupt
  + Interrupts are implemented by asserting a signal on some bus line which is detected by the interrupt controller
  + Queues interrupts and delivers them to the CPU
  + Handles each one and then tells the interrupt controller it has acknowledged the interrupt
* Interrupt Handling-Acknowledges the interrupt, allowing the interrupt controller to deliver the next one
* Interrupt Precision-When an interrupt occurs, we may have multiple partially completed instructions pending
  + An interrupt is precise if
    - The program counter is saved somewhere
    - All instructions before the program counter have finished
    - All instructions after the program counter have not yet started
    - Execution state of the instruction at program counter is known
  + If not satisfied, the interrupt is imprecise
    - It’s the OS’s job to roll back any partly finished instructions before resuming
    - The architecture’s interrupt handling will provide lots of information about the half-completed state at every interrupt
* Device independence-Programs shouldn’t have to care which specific device is used for input/output
* Uniform naming – Names for I/O resources should not be tied to a specific hardware device
* Error handling – Handle errors at the lowest level possible, avoid bothering the user unless necessary
* Make I/O appear synchronous to the user
* Handle resource sharing for devices that support it
* Programmed I/O-CPU does all the work
  + Poll the hardware, send or receive the data, and repeat until all data is sent or received
* Interrupt-Driven I/O-Instead of polling while waiting for hardware to be ready, we could ask the hardware to tell us via an interrupt
* DMA I/O-Program the DMA controller for a bulk transfer
* Interrupt Handlers
  + Save additional registers
  + Set up kernel context
  + Set up a stack for the handler
  + Acknowledge the interrupt
  + Copy saved registers into the process descriptor
  + Run the interrupt handler
  + Choose the next process
  + Switch context to new process
* Interrupt handlers need to do the minimum possible work needed to acknowledge the interrupt
* Queue the remaining work to do later
* Device drivers do the actual work of interacting with the device controller
  + Initialization-Talk to the controller to get device into a working state
  + Control – Take commands from higher levels in OS and translate them into hardware requests
  + Multiplexing – Check whether device is in use and defer until later if so
  + Power management – Put the device to sleep when it’s not being used
* Buffering – Don’t give data back in small pieces
  + Store it up in some buffer until enough has accumulated
  + Stored in Kernel Space with two buffers
    - A process can transfer data to or from one buffer while the operating system empties or fills the other buffer
* Circular / Ring buffer – Keep a buffer and two pointers
  + One buffer is the free slot which can be written there and advanced to the buffer
  + The other is the first unprocessed item in the buffer; can read from there and advance the pointer
  + When either pointer reaches the end of the buffer, wrap around to the start
* Threads
  + Share memory
  + Preemptive thread scheduling is a problem
  + Individual processes have little control over order in which processes run
* Preemptive scheduling introduces non-determinism
* Race Condition-Two or more processes run in parallel and output depends on order in which they are executed
  + Preemption at the wrong time
* Synchronization-If two threads, processes, interrupt handlers, etc. Are going to have conflicting accesses, force one of them to wait until it is safe to proceed
  + Memory: Multithreaded application
  + OS Object-Two processes that read/write same system file
  + Hardware devices-Two processes that both want to burn to a DVD
* Atomic operations – Series of operations that cannot be interrupted
* Mutual Exclusion-Making sure that only one process has access to the shared resource at a time
  + Majority of the time, programs do things that don’t require synchronization; the things they do only affect non-shared resources
  + Critical region-Shared memory or file that had to be accessed by only one program
* Peterson’s Algorithm-
  + Before using shared variables, each process:
    - Calls enter\_region() with its own process number, 0 or 1, as a parameter. This will cause it to wait until it’s safe to enter
    - After it h as finished with shared variables it calls leave\_region() to indicate that it is done and to allow the other process to enter
  + Initially, no process in the critical region
  + Process 0 calls enter\_region()
  + Indicates interest by setting the array element to 0 and also the turn to 0
  + Each process indicates its interest by setting its entry in the “interested” array
  + Sets the global turn variable to its own process number
  + Loop until turn indicates that it’s our turn and we see that the other process is no longer interested



* Compare & Swap
  + Compare is made between a memory value and a test value
  + If the values are the same a swap occurs
  + Carried out atomically
* Busy Waiting-Whenever a process is waiting to enter a critical section, it sits in an infinite loop
  + This wastes CPU time
* Priority Inversion-



* Mutex-Instead of looping, yield the CPU



* Semaphore-Using a lock to solve the lost wakeup problem
  + Use a semaphore-An integer to coutn the number of wakeups pending
  + No way to inspect or manipulate semaphores other than these three operations
  + May be initialized to a nonnegative integer value
  + semWait decrements the value
  + semSignal operation increments the value
  + No way to know before a process decrements a semaphore whether it will block or not
  + There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently
  + You don’t know whether another process is waiting so the n umber of unblocked processes may be zero or one
* Strong Semaphore-The process that has been blocked the longest is released from the queue first
* Weak Semaphore-The order in which processes are removed from the queue is not specified
* Monitors-Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
  + Local data variables are accessible only by the monitor’s procedures and not by any external procedure
  + Process enters monitor by invoking one of its procedures
  + Only one process may be executing in the monitor at a time
* Synchronication-Achieved by the use of condition variables that are contained within the monitor and accessible only within the monitor
  + Operated on byu two functions
    - Cwait©-Suspend execution of the calling process
    - Csignal-Resume execution of some process blocked after a cwait on the same condition
* Barrier-Force a group of processes / threads to wait until all have finished
  + Mix of parallel and sequential steps