Operating Systems: Study **All** the Things

Three [4] Easy Pieces

Persistence Saving data across computer power cycles; *non-volatile* memory can use magnetic or optical media or solid-state memory that remembers its state.
Virtualization Emulating part (or all) hardware in system software to protect the actual hardware from direct access by running software. Can apply to the *processor* and the *memory*.
Concurrency Safely sharing resources across multiple processes that are running "at the same time", typically by ordering their instruction interleaving such that they behave as if they were run serially.
Abstraction Dr. Ladd's *Fourth Easy Piece* The operating system provides a facade over heterogeneous hardware, an abstract interface that hides the complexity of the underlying systems.

0.1 Mechanism/Policy

Mechanism v. **Policy** — Remember that mechanism is *how* the system can do something and policy is deciding *when* or *why* to do it.

Computer Hardware

Computer Hardware
1. CPU
2. RAM
3. Disk(s)
4. I/O

Memory Hierarchy
1. Registers
2. RAM
3. Disks
4. Network

Von Neumann Architecture — CPU runs a *fetch-decode-execute* cycle.
Program counter keeps track of next instruction.
Instructions can manipulate register contents **or** load register from memory **or** store register to memory.
All data manipulation is at the CPU.

Hardware Assists for Modern OS

Privilege Bit (*system bit*, *kernel bit*) The *privilege bit* is a CPU flag bit that determines whether the CPU is currently executing on behalf of the *system* or the *user*.
Privileged Actions

- Direct memory access (bypass memory address translation and page table).

- Execution of CPU instructions deemed *privileged*: setting CPU flags, installing interrupt handlers, I/O, etc.
- Changing the CPU flags

The *user* program runs with **limited-direct execution**: non-privileged instructions run directly in a translated memory space while privileged instructions require a `syscall` (software interrupt) into the operating system for the system to perform (after checking that it is permitted) the operation.
Interrupts The fetch-decode-execute cycle can be interrupted *asynchronously* by an *interrupt signal*; the `syscall` instruction will also generate an interrupt when run. Each interrupt has a number indicating its source (so the power switch, `syscall`, a timer expiring, and the network controller signaling it is ready can be told apart) and that number is used as an index into the `ISV` *interrupt service vector* (remember: vector = array), a vector of machine **addresses** where the OS has put code to handle each kind of interrupt.
Syscall (*trap*) an instruction that generates an interrupt from user-space code. The only way to elevate privileges from user-space to system-space.
CPU registers are used to pass requested function and required parameters to the *syscall handler* which is just an *interrupt handler* for the specific syscall interrupt.
k-stack The *k-stack* (kernel stack) is one way for the OS to capture the context of the interrupted process. This **hardware assist** sees the CPU copy full register contents onto the k-stack when the interrupt is taken; the interrupt service routine can run and either return from interrupt (back to the same process) **or** save the contents of the k-stack, put the context of a different process there, and return from interrupt into a different process.
Note that returning from an interrupt uses the k-stack as well, restoring the state on the k-stack onto the CPU before dropping from kernel to user mode and starting the instruction addressed by the PC.

Speed/Safety Spectrum

Unlimited Direct Execution: *Fast* execution with no protection of one process from another.
Limited Direct Execution: Compromise between speed and safety
Virtual Execution: *Safe* execution with every instruction run in software.

Persistence

Interacting with Peripheral Devices

Polling While the process requesting the I/O is running, the CPU continuously reads (*polls*) the device's interface to see if the operation in finished. When it is finished, the processor can return from the `syscall` to the process's user code (without a context switch).
Interrupts Since the CPU resource is busy doing nothing, better use can be

made by *blocking* the process that initiated the I/O until the device reports that it is done. The CPU is free to switch to other, non-blocked, processes.
To signal that it is done, the device raises an *interrupt.* An interrupt (just like the timer interrupts of multiprocessing) causes the CPU to jump to operating system code based on the *interrupt number* and the *interrupt service vector* (ISV). The handler determines which blocked process can be made ready and moves the PCB from the waiting to the ready queue.
Hybrid Benefit of polling: no context switch if wait is short; cost of polling: running the CPU w/o making progress. Best of both worlds? Poll for a little while; if not done, block on the device. Short I/O does not pay a context switch; long I/O does not hold the CPU.

Question: Using *hybrid* I/O device handling above, is the amount of time that the OS polls for a given process *mechanism* or *policy*? Must the length of polling time be the same for every process?

Spinning Platter Drives

A *spinning* disk drive has

Platters Spinning, round, flat disks covered with magnetic media. Drive has one or more; they can be single or double sided.
Tracks Each platter is divided into concentric rings for storing data. A *head* is positioned over a track to read or write bits on the magnetic medium passing below it.
Sectors Each track is divided into some number of arcs. The magnetic medium in each arc can store bits when influenced by a strong magnet in the write head. Different tracks can have different numbers of sectors: each sector holds the same number of bits so outer tracks can fit greater numbers of sectors because the perimeter is greater.

Each sector has a unique address: [platter, track, sector]: platter picks which read/write head (there is at least one per platter) to use; track indicates how far from the center of the disk the sector lives; sector number indicates the angle around the circle of the track the data begins.

A sector is often between 512B and 4KB in size. There are file systems with variable size sectors and some with very, very large sectors.

Fragmentation

A *file* stores *bytes* into one or more *sectors* on a drive. Assume file sizes (in bytes) are randomly distributed (probably true for a general-purpose file system). The last sector of the file "wastes" half of the sector, on average. The amount of this *internal* fragmentation (internal to the file; out of sight of the file system) depends on the size of sectors.
Question: What is *external* fragmentation which is visible to the file system?
Address Translation Just as the OS can translate addresses inside a process, so modern disks translate their internal addressing to provide a simple,

consistent view: a drive is just an *array* of *sectors*. Each has a simple integer index.

On-disk Data Structures

A **file** is an extensible array of *bytes* when viewed from a user program. There is an *offset* for reading the array and another for *writing* the array. A read past the end of the array will return end of file and a write past the end will extend the array to fit the new data.

What is a *file* to the **file system**? A *named* sequence of sectors along with some *metadata*.

A sequence of sectors must be stored *on the disk*. This could be done with linked lists (remember how DOS sped this up?) or a tree.

Unix (our Very Simple File System) uses an *inode* that contains

Type (D, F, ...)
Reference Count
Size (in Bytes)
Direct[n]
Indirect1
Indirect2
...

This is a *skewed tree*.

Question: What does it mean to be a skewed tree? Draw a picture. Much more important: Why? What did designers know about access patterns and file statistics before settling on this?

A **directory** is just a special kind of file. It has the same kind of inode with a different type field. The content of the directory is *structured*. It is a sequence of `name`/`inode#` pairs.

Directory traversal: To translate from a path, `/home/laddbc/review.txt`, to an inode: OS must know the inode number of the root directory in the file system. This is part of FS design and can be set up when the FS is mounted.

Each directory's *inode* is loaded (starting with the known location of the root directory). Then the *data* for that folder is loaded and searched for the next *path component's name*. The path component's name is associated with the inode for that component. If the component is a directory, recur. If not, it is the inode for the file named in the path.

Path	Inode	Data	
`/`	√		`/` inode
		√	`/` contents: `home`:inode
`/home`	√		`/home` inode
		√	`/home` contents: `laddbc`:inode
`/home/laddbc`	√		`/home/laddbc` inode
		√	`/home/laddbc` contents; `review.txt`:inode
`/home/laddbc/review.txt`	√		`/home/laddbc/review.txt` inode

The final inode can be stored in the *file control block* that the internal **file descriptor** can refer to while the file is open. Note how all three names are

relate.

Path The hierarchical name leading across directories to the file. A file can have more than one simultaneous path name, its path name can change over time, and it is possible to remove this name from a file without destroying it (if the file is open).
File Descriptor Index of an internal (OS) data structure for an open file inside a process. The same file can have multiple file descriptors associated with it. File descriptors do not persist after the end of a process.
Inode An on-disk data structure that is allocated when the file is create and remains the same (the same inode) throughout the life of the file. The inode contains addresses for data segments of the contents and often counts the number of references (from other types of names) are made to the inode.

Bit-Maps Bit maps are compressed Boolean arrays. They are quick to load, store, and search. They are small(-ish) and are used to efficiently search for free *inodes* in the inode table and *data blocks* in the block store.

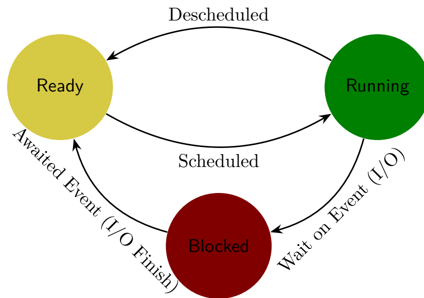
Virtualization

Virtualized Processing

Process Abstraction

Program Binary-encoded instructions in a *file* on *disk*.
Process Program file contents loaded into RAM along with *context*

Context of a process is the setting of all CPU and memory subsystem registers. The idea is that if the OS captures the context of the running process, the CPU can do "something else" and then, at some time in the future, put the saved context back and the same process would continue running, unaware that it had been paused.
Context Switch is changing from the context (execution) of Process A by *saving* its context to the context of Process B by *loading* its context from a saved copy. This is how two (or more) processes can share a single CPU concurrently.
Process Data Structures The OS keeps track of a process in a *process control block* (PCB); the PCB for all processes as stored in an OS-level vector that is indexed by the *process identifier* (PID) of each process.
In the PCB the OS keeps the *context* of the process, memory configuration, file control blocks, etc.
Process state is one of the things tracked in the PCB (the state is also indicated by where the PCB resides: Running, in a ready queue, or in an I/O queue):



Question: What happens, at the CPU register level, when a timer interrupt occurs, triggering a context switch between process S and process T? Include the *k-stack*, the *privilege bit* and when it is set/cleared, and one or more *PCB* in your answer. What state(s) are S and T in *before* and *after* the switch?

Where, in this diagram, are the *scheduling queues*? That is, where do processes that **could** make progress but are not on the CPU wait?

Starting Processes

`fork` Copy the current process (with a new PID and PCB). Return PID of child (to parent) or 0 (to child).

`exec` Load the named file/program over the current process, resetting all registers and memory subsystem settings to process start values.

Scheduling

Scheduling Metrics Which is important depends on the type of *workload* the system expects.

Turn-around Time $T_{\text{turn-around}} = T_{\text{completion}} - T_{\text{arrival}}$; batch workloads

Response Time $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$; interactive workloads

Scheduling Policies

First In, First Out
Shortest Job First
Shortest Time to Completion
Round Robin

Question: Which of the given policies do not require a mechanism for *preemption*? Explain why the others do require that mechanism.

Multi-Level Feedback Queue with a quantum length q. Rules when picking the next job:

1. On arrival, job J is put in Q_{max}.
2. If job K surrenders the CPU before its quantum is done, it goes to the end of Q_P; its priority remains unchanged.
3. If job K uses its whole quantum, it goes to the end of Q_{P-1}; its priority is lowered.

4. Next job to schedule is the first job on Q_X where X is the highest priority of a non-empty ready queue.
5. After the *boost period*, B, $B \gg q$, all jobs are reenqueued into Q_{max}.

Question: Why is rule 5 necessary? What if $\neg(B \gg q)$?

Question: Consider an MLFQ with: 3 queues, quantum of 1, allotment of 1, and a boost time of 25. The queues are numbered, in decreasing priority order: 3, 2, 1. Given the following list of jobs:

Job	Arrival	Run Time
A	0	4
B	2	4
C	4	1
D	6	3

Mark the quanta in which each job executes in the following table, using as the marker the *number* of the queue the job was scheduled off of.

So, for example, if Job D came off of queue 2 in the first quantum, there would be a "2" in row D and column 1 (the quantum starts at 0, ends at 1).

Before you mark anything:

How many non-empty squares will there be in each row?

How many in each column?

How many total in the table?

	Quantum Ending													
Job	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A														
B														
C														
D														

How many columns? **12 (total quanta for all programs)**

How many non-empty squares will there be in each row? **Run time of program in the row.**

How many in each column? **One, there is only one CPU being scheduled.**

How many total in the table? **12, one per column or one per quantum where a program is running.**

	Quantum Ending											
Job	1	2	3	4	5	6	7	8	9	10	11	12
A	3	2				1				1		
B			3	2					1			1
C					3							
D							3	2			1	

Virtual Memory

Memory Translation

Address Spaces

Virtual The **process** address space. *Always* runs across the exact same range (*i.e.* `0x00000000 - 0xFFFFFFFF`). Permits the compiler/linker to ignore location in *physical* address space.

Physical The **hardware** (and **operating system**) address space. Runs from the first byte in a **RAM** chip to the last byte in the installed memory. Addresses also start from 0 and count up, by byte.

Loading Policies

Singleton Load exactly one process at a time, into address `0x00000000`. *Pros:* no address translation, no CPU modification, easy compilation; *Cons:* maximum of one running process.

Static Relocation Let the compiler target *some* range of physical memory when executable is created. Load the program into that location. Multiple programs, with non-overlapping address usage, can share the physical address space. *Pros:* no address translation and no CPU mods; *Cons:* each process has a single place to load or must be recompiled (or stored multiple times).

Dynamic Relocation OS **transparently** translates addresses based on where the process was loaded. *Pros:* easy compiling, can load one executable into any available memory, multiple processes in memory simultaneously; *Cons:* need base/bounds (or similar) registers [**Hardware Assist**], each memory access is translated (and bounds checked), dynamic memory management in the OS, must allocate all process memory contiguously.

Segmentation Multiply the base/bounds concept by having a base/bounds per process *segment*. *Pros:* easier to fit into fragmented memory, uses existing link specification; *Cons:* more registers (this is context).

Paging Allocate all memory in small, identically sized blocks (*pages* of virtual memory into *frames* of physical memory). *Pros:* no need to track size of allocation, no need to find contiguous blocks of RAM, can be swapped out to secondary storage (because of one size this is easier), no need to load entire segments into memory; *Cons:* too many registers (need a *page table* per process that is stored in RAM), bookkeeping on what frames to free on a page fault is non-trivial.

Paging and segmentation can be combined.

Paged Virtual Memory

A *page table entry* (PTE) contains *metadata* about the page and the *frame* address where it is (if it is in memory) or the *swap* address where it is if the OS has swapped it out.

Metadata can include the *valid* bit (is this page a proper part of the process's address space), the *present* bit (is this page resident in memory), the *dirty* bit (has this page changed since it was loaded), and the *reference* bit (has this page been accessed by the process since the OS reset reference bits). The address field in the PTE can contain a *frame number* (if page is *present*), a *swap page number* if the page was evicted to the swap, or (effectively) `null`. The size of the

PTE is determined by taking the number of bits of address, adding the number of bits of metadata, and rounding up to the next byte size.

The *page table* is an array of PTE. There is one PTE per page in the *process address space*. The size of the page table is the product of the number of PTE and the size, in bytes, of each PTE. The context of the process includes a *page table base register* (PTBR), the physical address of the beginning of the page table.

To translate an address: Split address into page number (**P**) and offset into page (**o**). Use **P** and size of PTE to calculate array offset into page table of proper PTE. Add array offset to PTBR and fetch PTE. Assuming no page fault, extract frame number from PTE (**F**) and prepend **F** to **o** to get physical address.

On page fault (page not present): OS figures out where data for the page lives: in the program file (if it is part of the immutable `text` or `data` segments), in swap (if it is a mutable page that had been allocated earlier), or should be newly allocated (it is in the free space between segments). If it is on disk, schedule the read and block the process on the completion. If it is a new allocation, grab a frame, zero, and update the PTE (into page table).

Once page contents are read, pick a victim frame and put the data in it. Update PTE for this process and unblock it and also update page table of process from which the frame was taken. Once page is full and PTE updated, make process ready and it will restart the instruction.

Victim is selected by some policy. One good one is *least-recently used* (LRU).

Question: The table below describes a *paged* memory system. Fill in the empty boxes to complete the description and answer the questions following the table.

Metadata (b)			
valid	present	dirty	reference

	Address Space Size (B)	Bit Width Address	Page/Frame Size (B)	Bit Width Offset	Number of Pages/Frames	Bit Width Page/Frame Number	Page Table Entry Size in Bytes
Virtual	1M						
Physical	32M		4K				

a. Explain why the number of bits in the *page numbers* and the number of bits in the *frame numbers* **are/are not** the same.

b. How **many** entries in a *page table*?

c. **Size** (B) of *page table*?

   Process Q's page table is in memory beginning at `0x1000`. The process's page table begins with the following entries (spaced as closely together as possible on whole byte boundaries); all numbers are in hex.

Page Table

Meta	Frame
3	000
C	0F0
A	00E
E	D00
E	0CC
F	B0B
B	AA0

```
03FF8       movl  0x05010, %eax
        top:
03FFC       cmpl  $0x21, %eax
04000       jg     bottom
04004       movl  0x05010, %eax
04008       addl  $0x01, %eax
0400C       movl  %eax, 0x05010
04010       jmp   top
     bottom:
04014
```

Initially `RAM[0x05010] == 0`.

d. Which *pages* are invalid?

On our computer, each instruction takes exactly one (1) clock cycle. Consider the following code:

e. Give the *virtual* address trace for 12 clock cycles of this code.

f. Using the page table given above, give the corresponding *physical* address trace.

User Memory Allocation

`malloc` is part of the C Runtime-Library (CRT).

One way of keeping track of free memory is with a *free list*: a linked list where each node is a block (of variable size) with two fields placed at the beginning of the block: `size` and `next`. `size` is the size, in bytes, of this node and `next` is the address of the next node in the free list.

Question: Is `next` a *physical* or a *virtual* address?

Allocation is done by traversing the free list for a node large enough to service the `malloc` call, along with any bookkeeping fields. Consider the **best-fit**, **first-fit**, and **worst-fit** policies for allocating memory.

Question: Describe *internal* and *external* fragmentation in terms of `malloc` that allocates only 1KB and 2KB blocks. What could possibly go wrong?

a. When could *internal* fragmentation cause there to be enough "free" memory to exist for a 1.5KB allocation yet the system be unable to fulfill it?

b. When could *external* fragmentation cause there to be enough "free" memory to exist for a 1.5KB allocation yet the system be unable to fulfill it?

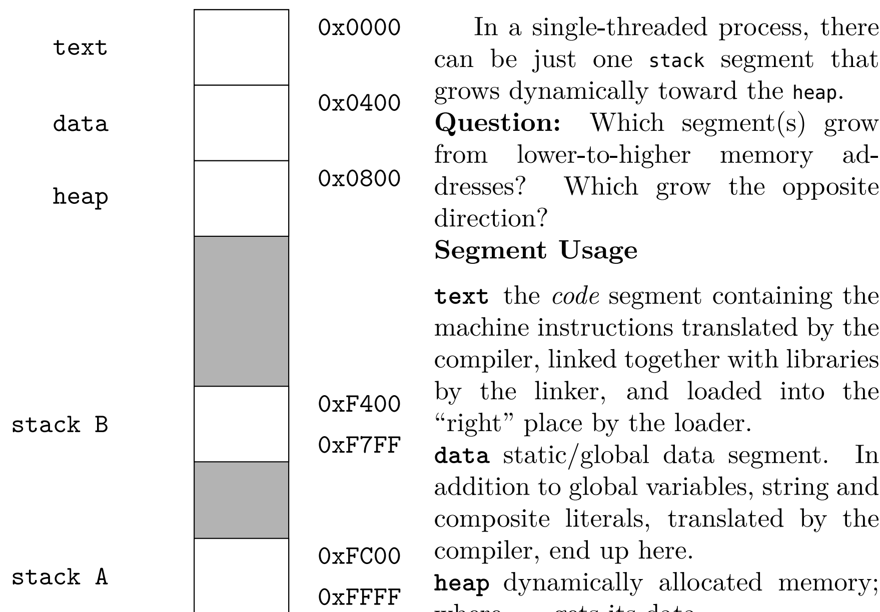
Concurrency

Concurrency Terms

critical section a piece of code that accesses a *shared* resource.

race condition when multiple *threads of execution* enter their [same resource] critical region at roughly the same time; multiple updates to the shared data structure leads to potentially broken execution.
indeterminate programs are programs with race conditions; results of the computation depends on ordering of threads and can vary from run to run.
mutual exclusion structures protect critical regions, permitting at most one thread into the critical section. This eliminates race conditions, making the program determinate at the potential cost of parallel execution and thus speed.

Process Segments



Addresses assume 64KB address space and exactly 1KB in use by each segment. Each thread has its own, independent `stack` segment. This picture shows a stack segment for thread A and for thread B. The stack pointer (and, perhaps, stack origin) is part of the *thread context* and must be saved in the *thread control block* (TCB).

In a single-threaded process, there can be just one `stack` segment that grows dynamically toward the `heap`.
Question: Which segment(s) grow from lower-to-higher memory addresses? Which grow the opposite direction?
Segment Usage

`text` the *code* segment containing the machine instructions translated by the compiler, linked together with libraries by the linker, and loaded into the "right" place by the loader.
`data` static/global data segment. In addition to global variables, string and composite literals, translated by the compiler, end up here.
`heap` dynamically allocated memory; where `new` gets its data.
`stack` dynamic allocation of local variables for function calls; the stack is made up of *activation records*, one for each currently active function.

Question: Which segment(s) are loaded from the *program file* (statically)? Which segment(s) have their contend generated purely dynamically?

Concurrency Primitives

Locks A `lock` (`Mutex` in Rust) is an ADT supporting

`lock` function permits one thread past; all other threads *spin* (keep checking

the lock status) **or** are put in the `Blocked` state with an `unlock` operation serving to signal the thread to return to `Ready`.
`unlock` function releases some waiting thread to continue past its call to `lock`. Lock is cleared if no thread is waiting.

Locks require some form of *atomic update* machine instruction (hardware assist) such as *test-and-set*, *load-linked/store-conditional*, *compare-and-swap*, or *fetch-and-add*.
Condition Variables A `condition_variable` (`Condvar` in Rust) is an ADT using an external lock and a queue to protect a critical region without a spin lock. The queue holds threads in the `Blocked` state.

`wait(&lock)` thread must hold `&lock` on call (and will hold it on return). `wait` appears in a loop that tests the condition that would permit the thread to continue and is called when the condition is false. `wait` (atomically) enqueues the thread, releases the lock, and then does a thread context switch. When the thread is made ready (dequeued), the `wait` code acquires the lock and then return from `wait`. (And the condition is tested again.)
`signal(&lock)` While holding `lock`: if any threads await on the queue, make the first one `Ready`. Release `lock` and return.

A thread will only pass the `wait` loop when the condition being tested is true.
Semaphores Initialized with a count (maximum number of threads allowed into the critical region) and

`wait` decrements the counter and returns if the counter is non-negative (thread continues into critical section) or blocks on a queue otherwise.
`post` increments the counter and wakes one of the waiting threads (if there are any).

Again, `wait` is called in a loop.

Concurrent Algorithm Evaluation Criteria

Safety (or **Mutual Exclusion**) is the critical region actually *safe*? There must be no sequence of interleaving that permits mutual exclusion to be violated.
Fairness does the algorithm avoid *starvation*? Does a waiting thread enter the critical region **before** the thread that is in the critical region *reenters* it?
Performance what is the runtime impact of the primitive? Consider no-contention, two-thread contention, and multi-thread contention. With caches and multiple processors, new performance concerns arise.

Common Concurrency Problems

Producer/Consumer (Bounded Buffer) A set, P. of producer threads produce data; a set, C, of consumer threads use data; communication is through a data *buffer* with n elements.
If a producer makes data and there are no buffer slots, it must wait until one is free; if a consumer goes to take data and there is none available, it must wait

until one is available.

Reader-Writer A set of readers, R, and writers, W, share some data structure, D. Readers read but do not change D; writers read and *may* modify D.

If a only readers are in their critical regions, arriving readers may enter theirs. If a writer comes, it must wait until all readers leave their critical regions; if readers arrive after a writer begins waiting, they wait on the writer leaving its critical region.

Writers must enter their critical regions mutually exclusively with regard to both readers and writers.

Efficient parallelism of readers and mutual exclusion for writers makes this a challenge.

Dining Philosophers n philosophers sit at a round table with a bowl of rice in the middle and one chopstick between each pair of philosophers. Philosophers live life by thinking until they get hungry. When hungry, a philosopher must pick up both adjacent chopsticks before he can eat rice. When no longer hungry, the philosopher releases both chopsticks.

The problem is making a starvation-free algorithm for chopstick acquisition given that a philosopher cannot communicate with his neighbors and only knows the number on the chair on which they sit.

Atomic Instructions

Consider `SpinLock`:

```
class SpinLock {
  public:
  SpinLock();
  void lock() volatile;
  void unlock() volatile;
};

// -----------------------------------------------------------
// testAndSet(lockVar, newVal): atomically replace lockVar with newVal; return original value.
int testAndSet(int &lockVar, int newValue) {
  int original = lockVar;
  lockVar = newVal;
  return original;
}
```

Question: Implement `SpinLock` with `testAndSet`.

```
// -----------------------------------------------------------
// compareAndSwap(lockVar, expVal, newVal): atomically swap lockVar value with newVal
// IFF lockVar's value was expVal. Return lockVar's original value
int compareAndSwap(int &lockVar, int expVal, int newVal) {
  int original = lockVar;
  if (original == expVal)
    lockVar = newVal;
  return original;
}
```

Question: Implement `SpinLock` with `compareAndSwap`.

```
// ------------------------------------------------------------
// loadLinked(lockVar) fetch and mark lockVar, return value
int loadLinked(int &lockVar) {
  return [and mark] lockVar;
}

// saveConditional(lockVar, newVal) set lockVar to newVal IFF still marked by this thread
// return whether the value was saved
bool saveConditional(int &lockVar, int newVal) {
  if (lockVar is marked by me) {
    lockVar = newVal;
    return true;
  } else
    return false;
}
```

Question: Implement `SpinLock` with `loadLinked`/`saveConditional`.