Commensurate 相称的
provision 规定、供应
Explicitly 明确地
Implicitly 隐含地
Obliviously 没有意识到地
absence 缺席、缺乏
enforce 强制执行
canonically 标准的、规范的
inclusively 包括地、含括地
precede 在...之前、优先于
intermittently 断断续续地
Legitimate bug: real bug, not false positive 合理的错误：真正的错误，不是误报
contrived 做作的、牵强的
preclude 排除、妨碍
decree 法令、法规
anticipation 预期、预料
entitle 使有权利、给予资格
Unreserved 未被预定的、未被保留的
Resort 手段、诉诸
Reclamation 回收、取回
exacerbated 使恶化、加剧
susceptible 易受影响的、易受感染的
revocation 撤回、废除
tentatively 暂时地、试验性地
coalescing 合并、联合
inflate 膨胀、扩大
speculatively 推测性地、投机地

Upcalls go from the kernel into applications (kernel notifies user).

An uniprogrammed machine is a type of computer architecture in which the processor is dedicated to running a single program at a time. This means that the entire system resources, including the processor, memory, and input/output devices, are devoted to executing a single program until it completes or is preempted by the user or system.
In an uniprogrammed machine, the operating system provides little or no support for multitasking, and the program must be designed to explicitly manage its own resources, such as memory and input/output operations. This can result in inefficient use of resources, as a program may spend a lot of time waiting for input/output operations to complete, during which the processor and other resources are idle.
Uniprogrammed machines were common in early computer systems, but have been largely replaced by multitasking operating systems and hardware architectures that support simultaneous execution of multiple programs or threads.

Critical sections are pieces of code where some invariant may not be true while some structures are being updated. The code in the critical section may need to be protected (for example, from other threads observing it). Critical sections are often protected by locks, or by disabling and re-enabling interrupts (or a combination). Code in a critical section has more "control" of the processor because it would be bad to preempt it while in the critical section.

Forkable procedures are procedures that can be executed by a process, and then later forked to create a new process that executes the same procedure from the beginning. These procedures are also referred to as reentrant or thread-safe procedures.
Forkable procedures can be run sequentially because they are designed to be executed in a non-concurrent environment, where only one thread of execution is active at any given time. When a forkable procedure is executed, it performs its operations and returns control to the calling process, allowing the process to execute other procedures or tasks.
If a process decides to fork a new process that executes the same procedure, the new process will start executing the procedure from the beginning, just like the original process did. Since the procedure is designed to be reentrant, it can safely be executed by multiple processes without interfering with each other.
Therefore, multiple instances of the same forkable procedure can be executed sequentially, one after the other, without any issues or conflicts.

incorrect to call a sequential procedure with fork because the behavior of the sequential procedure may depend on its state, which may not be correctly preserved across the fork. When a process forks, the child process receives a copy of the parent's address space, which includes the procedure's state. However, if the procedure maintains any state outside of its own stack, such as static or global variables, file descriptors, or mutexes, that state may not be correctly preserved across the fork. As a result, the child process may exhibit unexpected behavior or even crash when executing the sequential procedure. Therefore, it is important to ensure that procedures that are used with fork are designed to be fork-safe, meaning that they do not rely on any state that may not be correctly preserved across a fork.

Yes, wait operation in a monitor needs to be locked.
In a monitor, only one thread can be active at a time and has exclusive access to the monitor. When a thread invokes the wait operation on a condition variable, it releases the lock on the monitor and blocks until another thread signals the condition. When the condition is signaled, the waiting thread is awoken and attempts to re-acquire the lock on the monitor.
If the wait operation is not locked, it may result in race conditions and data inconsistencies. For example, if one thread is waiting on a condition while another thread modifies the shared data without holding the lock, it may cause data corruption or incorrect results. To avoid such issues, it is important to hold the lock before calling the wait operation and ensure that any modifications to the shared data are made while holding the lock.

The monitor provides mutual exclusion for a section of code, meaning only one thread can be executing within a monitor at a time. The monitor entry call requests entry into the monitor and if the monitor is busy, the thread is blocked until it can acquire entry.
If the monitor entry call is not locked, multiple threads may simultaneously try to enter the monitor, causing race conditions and undefined behavior. Therefore, it is important to lock the monitor entry call to ensure only one thread can request entry at a time. Once a thread enters the monitor, it will automatically release the lock, allowing other threads to enter the monitor.

I'm confused about why explicit page faults in a guest OS will have less overhead. My understanding is that a page fault in the guest OS (which will cause an update in the guest page table or TLB) will be intercepted by the ESX server so that pmap and shadow table are properly updated. If this is correct, then it also requires execution within the kernel and should also block the guest OS.
My understanding is that the process you mentioned will only happen on implicit page faults, because the page should exist and the ESX server needs to retrieve it. For explicit guest OS page faults, they can just be forwarded to the guest OS to handle, which is less overhead.

I think the difference is that outside blocks explicitly mark the procedure's intention regarding the lock/the invariants. Without outside blocks, you can't know whether a call outside the monitor will release the lock, so you have to protect against the possibility that it will. As a result, every time you make a call outside the monitor, you need to restore the invariant, even if it's unnecessary (maybe the call doesn't end up using WAIT at all).
Outside blocks let the callee specify their intention regarding the lock (should it be released or not). Perhaps some/most calls outside the monitor won't release the lock, and therefore don't need the caller to restore the invariant before making the call.
In other words, outside blocks allow the programmer to limit the amount of "cumbersome" invariant restoration they have to do when making an outside monitor call to just the locations where it is really necessary.

An invariant is some property of the program that is always true, or at least is always true during certain parts of the program. Often invariants are properties of data structures that are preserved across operations on those data structures. For example in a tree data structure you might have an invariant that there is never a cycle among the nodes. Or in a linked list, the head points to the first element, and each next pointer points to the next element.
In concurrency, locks protect invariants. When updating a data structure, you might have to briefly break an invariant: inserting into a linked list might involve a brief moment where the invariant is not true (because the pointers of nodes within the list take multiple instructions to update, so mid-update the invariant might not hold). Without a lock, another thread might be able to observe the list when the invariant is not true, which is bad because code is usually written with the assumption that the invariant is true.
Wikipedia gives a general definition: https://en.wikipedia.org/wiki/Invariant_(mathematics)#Invariants_in_computer_science.
I also recommend reading the start of chapter 6 of the xv6 book (particularly section 6.1): https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf, which discusses locks and invariants (the linked list example above is a shortened version of their example).

For the superpages paper, it is mentioned that if we need to allocate a 511kb data segment, we should reserve a 512 kb superpage? But in 4.2 it's mentioned that reservation superpage does not reach beyond the end of the object? What would be the case if we allocate a 65kb data segment?
Everything has to operate with at least base page granularity, so the paper means it won't choose a larger size as long as it won't reach beyond the end of the object in base page units, so it is allowed to be beyond the end of the object, up to the size of a base page (8K in this case).
Since 512K is within 8K of 511K, it is allowed. Essentially you can think of the memory object size as being forced to round up to the nearest multiple of 8K.

In software engineering, double-checked locking (also known as "double-checked locking optimization"[1]) is a software design pattern used to reduce the overhead of acquiring a lock by testing the locking criterion (the "lock hint") before acquiring the lock. Locking occurs only if the locking criterion check indicates that locking is required.

The pattern, when implemented in some language/hardware combinations, can be unsafe. At times, it can be considered an anti-pattern.[2]

It is typically used to reduce locking overhead when implementing "lazy initialization" in a multi-threaded environment, especially as part of the Singleton pattern. Lazy initialization avoids initializing a value until the first time it is accessed.

```
Singleton* Singleton::GetInstance() {
  Singleton* p = s_instance.load(std::memory_order_acquire);
  if (p == nullptr) { // 1st check
    std::lock_guard<std::mutex> lock(s_mutex);
    p = s_instance.load(std::memory_order_relaxed);
    if (p == nullptr) { // 2nd (double) check
      p = new Singleton();
      s_instance.store(p, std::memory_order_release);
    }
  }
  return p;
}
```

These are some properties of volatile.

- The volatile keyword cannot remove the memory assignment.
- It cannot cache the variables in register.
- The value cannot change in order of assignment.

The function of "volatile" in C++ has nothing to do with threading. Remember, the purpose of "volatile" is to disable compiler optimizations so that reading from a register that is changing due to exogenous conditions is not optimized away. Is a memory address that is being written to by a different thread on a different CPU a register that is changing due to exogenous conditions? No. Again, if some compiler authors have chosen to treat memory addresses being written to by different threads on different CPUs as though they were registers changing due to exogenous conditions, that's their business; they are not required to do so. Nor are they required -- even if it does introduce a memory fence -- to, for instance, ensure that every thread sees a consistent ordering of volatile reads and writes.

TOCTTOU (Time-of-Check to Time-of-Use) is a type of software bug that arises when a program checks the state of a resource (such as a file or network connection) and then makes a decision or performs an action based on that state, but the state of the resource changes between the time it is checked and the time it is used.
This can lead to unexpected behavior, security vulnerabilities, or data inconsistencies. For example, if a program checks the existence of a file and then opens it for writing, an attacker could create or delete the file between the check and the open, leading to unexpected behavior or security issues.
To avoid TOCTTOU bugs, it is important to use atomic operations that ensure that the state of a resource is consistent at all times, or to use techniques such as file locking or transactional memory.

I'll give a hint, and then you can follow up if you have more questions.

Think about what memory each function uses. Is it possible that they might end up accessing the same memory, without Eraser knowing that the variable associated with that memory has changed?
Is it related to the stack being used for recursive function call?.
Yes that's about right. The stack memory from foo could become marked as shared-modified after it is passed into do_work, and then that same memory could be reused by fact later in the program. Since Eraser doesn't know that the memory represents a different variable at that point, Eraser will emit a warning (false positive).

Does that mean this is a real bug (i.e., not false positive)?

Is the answer the ambiguity of the write between Read-Shared to Shared-Modified as discussed in lecture? Don't know how to approach this question.

Yes that's right. The description says that a write from a new thread will change the state from shared (or exclusive) to shared-modified. The figure says that any write will change the state from shared to shared-modified. The figure is correct (why?).

In class we also discussed that post-initialization, a read of the variable would also cause a transition from the Virgin state to the Exclusive State , whereas in the figure it is only specified to be a write which would cause the transition to the Exclusive state. Would this be considered another contradiction?

Mesa allocates stack frames from the heap, so it can add stack space on-demand rather than allocating a large stack per-thread ahead of time. This means a process that doesn't need much stack space won't waste any because it will only end up allocating as much as it needs. See page 14 for details.
in current systems that have pagination, is this still an issue? Or can the stack just grow upward as needed.
I think this is less of an issue now. Modern systems will allocate physical pages for the stack on-demand, though they must still reserve the virtual memory for the full size of the stack (cannot allocate virtual pages in those slots). That's not a problem because the virtual address space is so big. One drawback is that modern systems cannot shrink the stack once it has grown (though managed languages like Go have green threads with stacks that can shrink).

In modern system: The main difference between heap allocation and stack allocation is in the way the memory is managed. Heap allocation involves manually requesting and releasing memory, which can lead to memory leaks or fragmentation if not managed properly. Stack allocation, on the other hand, is managed automatically by the compiler, and the memory is automatically released when the function or block of code exits.
Another difference is in the speed of allocation and deallocation. Heap allocation is generally slower than stack allocation because it requires the operating system to allocate and manage the memory. Stack allocation, on the other hand, is faster because the memory is already reserved and ready for use.

"Yielding the CPU" refers to the process by which a currently executing process or thread gives up its control of the CPU to allow other processes or threads to run. When a process or thread yields the CPU, it essentially signals to the operating system that it is willing to give up its time slice and allow another process or thread to run.

When a thread yields, it moves from the running state to the ready state and allows the thread scheduler to select another thread to run.

An invariant is some condition that is true. Establishing the invariant means changing something to make that condition hold, and assuming an invariant means assuming that the condition holds.
So in the context of Mesa monitors, there is some invariant that is always true of the monitor data, except for when a process is executing in the monitor. The user can always assume the invariant is true when not in a monitor procedure, and the monitor code must make that invariant true again before exiting the monitor.
This explanation about critical sections is kind of related: https://edstem.org/us/courses/38501/discussion/3065321?answer=7008728
Critical sections are pieces of code where some invariant may not be true while some structures are being updated. The code in the critical section may need to be protected (for example, from other threads observing it). Critical sections are often protected by locks, or by disabling and re-enabling interrupts (or a combination). Code in a critical section has more "control" of the processor because it would be bad to preempt it while in the critical section.
The locks on monitors ensure that when a monitor procedure is called, there is only one process executing there. That's because the invariant could be false within the monitor, and an external process wouldn't know. Locks ensure that only a single process is in the monitor, knows the state of the monitor data, and re-establishes the invariant before exiting the monitor.

Memory Resource Management in VMware ESX Server
==

Reminder about midterm in one week
  No electronics--must bring printed copies of papers

```
Three kinds of memory addresses involved in virtual machines
  * Guest virtual addresses
  * Guest physical addresses ["physical" addresses in paper]
      Would be plain physical addresses if not in VM
      Only have meaning in the context of a specific VM
      Can be paged out to disk and not exist in RAM
      Can be mapped to any host physical address by VMM
      Can even map multiple guest physical pages to same host physical page
  * Host physical addresses [machine addresses in paper]
      Global across all VMs
      Never seen by guest OSes
      Correspond to actual bits in DRAM chips

These are defined by several data structures
  A guest operating systems running in a VM maintains a *primary* page table
    Maps from *guest virtual* addresses to *guest physical* addresses
  The VMM has per-VM data structure called pmap in this paper
    Maps from *guest physical* to *host physical*
    (Not to be confused with pmap machine-dependent layer in the BSD kernel.)
  In addition, the VMM maintains a *shadow* page table
    Maps directly from *guest virtual* addresses to *host physical*
    Hence is a function of guest PT and pmap
    Shadow PT is the only PT seen by hardware (assuming no nested paging)
    Accessed/Dirty bits authoritative (VMM must copy back to primary PT)
```

Can be computed lazily on the fly, so use shadow PT as a cache
Monday lecture will deal with keeping shadow PT in sync w. primary

Who sees what?	Guest-PT	pmap	Shadow-PT
Guest OS	Yes	No	No
VMM	Yes	Yes	Yes
Hardware	No	No	Yes

Unfortunately, today's paper uses an older terminology:
Ed Bugnion, who coined older terms, says he now prefers newer terminology
Let's translate the following terms as we read the paper:
Virtual address -> guest virtual address
"Physical" address -> guest physical address
Machine address -> host physical address
Similar, we have VPN, PPN, MPN for virtual, physical, machine page number

Note that VMWare workstation (or kvm, virtualbox, etc.) has 4th address type:
Host virtual memory is memory in processes running in host OS
That's because VMWare workstation runs as a process on an existing OS
Re-uses the host OSes device drivers, networking stack, etc.
E.g., can run emacs and vmplayer side-by-side, where emacs uses host VA
VMWare ESX is different in that it replaces host OS
Directly accesses NICs and other hardware
No host networking stack to worry about

In ESX, what are three basic memory parameters for a VM? min, max, shares
min - VMM always guarantees this much machine memory, or won't run VM
Actually, need min + ~32MB overhead
max - this is amount of guest physical memory VM OS thinks machine has
Obviously VM can never consume more than this much memory
share - how much host phys. memory this VM should have relative to other VMs
Big question addressed by this paper: Memory management when over-committed

Straw man: Page host physical mem to disk with LRU. Why is this bad?
OS probably already uses LRU, which leads to "double paging" problem
OS will free and re-use whatever guest physical page VMM just paged out
Also, performance concerns limit how much you can over-commit
[Can't modify OS bcopy to use many of Disco's tricks]
Goal: Minimize memory usage and maximize performance with cool tricks

Straw man 2: Give each OS a huge max memory, so guests never page
Avoids double-paging
VMM still doesn't know which pages are important
E.g., VMM must page out deleted file blocks lingering in buffer cache

What happens under memory pressure (Sec 6.3)?
System can be in one of four states: high, soft, hard, low
high - (6% free) plenty of memory
soft - (4% free) try getting OSes to give back memory (page as last resort)
hard - (2% free) use random eviction to page stuff out to disk
low - (1% free) block execution of VMs above their mem usage targets
Use hysteresis--exceed thresholds significantly before transitioning higher
How bad would oscillation be if you didn't do this?
Might be bad at 1% as you rapidly block/unblock VMs

```
    Maybe okay at 2% as long as OSes also freeing memory?

How to convince an OS to give back memory?  Ballooning
    Implement special psudo device driver that allocates pinned physical memory
    VMM asks baloon driver to allocate memory
    Baloon driver tells VMM about pages that guest OS will not touch
  How does balloon driver communicate with VMM?
    Section 3.6 says polls once per second to get target balloon size
    Could use any IO mechanism to communicate
      Access special guest physical address (handle via tracing), inb/outb
    Today would use "hypercall" (e.g., vmcall) instruction
    Could also conceivably use interrupts instead of polling
  What happens if balloon memory accessed?
    OS shouldn't touch private balloon driver memory, so VM probably rebooted
    Handle as hidden page fault that VMM satisfies with zero-filled page
    After this, assuming reboot, VMM needs to resynchronize balloon state
  How well does ballooning work (Figure 2)?
    Looks like only small penalty compared to limiting physical memory at boot
    Why the penalty?  OS sizes data structures based on physical memory size
      E.g., might have one "ppage" structure per physical page, will use memory
  Does Figure 2 show that ballooning is effective?
    Would be nice to have third bar with just random eviction paging
    As it is, don't know how much cleverness of technique is buying us

How to share pages across OSes?
  Use hashing to find pages with identical contents
  Big hash table maps hash values onto host physical pages:
    If only mapped once, page may be writable, so hash is only *hint*
      Hash table has pointer back to guest physical page
      Must do full compare w. other page before combining into shared CoW page
    If mapped multiple times, just keep 16-bit reference count
      If counter overflows (e.g., on zero page) use overflow table
  Scan OS pages randomly to stick hints in hash table
  Note:  Always try sharing a page before paging it out to disk
  How well does this work?
    Figures 4, 5 show significant memory savings
    Why is Reclaimed lower than Shared?  Still need one copy of data
  Does it matter for performance?
    p. 7 - tiny 0.5% average speed-up (when no memory contention)--why?
      Maybe improves cache locality
    So this probably means sharing is rarely harmful despite extra hashing
    But would be nice to see actual speedups under limited memory conditions

How does proportional share typically work (no idle tax)?
  Each VM has been assigned number of "shares" S by administrator
    and has been given some number of pages P by VMM
  Reclaim a page from OS with lowest ratio of "shares-to-pages" S/P
    E.g., if A and B both have S=1, reclaim from larger of the two
          if A has twice B's share, then A can use twice as much memory
    Can view S/P as "price" guest OS can afford to pay for a page
  Why is this not good enough?
    May have high-priority VM wasting tons of memory
    (This is reasonable:  high priority VM might sometimes not need all memory)
  Digression:  Simple tax arithmetic
```

  - [Ignore payroll, state, and local taxes.]
  - Suppose my income tax rate is tau (e.g., might be 24% in U.S.)
    - For each $1 gross I earn, I pay tau*$1 in taxes
    - So $1 gross = $(1-tau) take home
    - And $1 take home requires $1/(1-tau) gross pay
  - Let k = 1/(1-tau) be cost of a take home dollar (e.g., ~$1.32 for 24% tax)
- Idea: idle memory tax. substituting memory pages for dollars...
  - Any page you are using is fully "tax deductible" - just costs one page
  - If you aren't using a page, must pay fraction tau of it back to the system
    - So each idle page actually costs you k times the price of a non-idle page
  - Now how much can a VM afford to pay for each "take home" page?

$$\text{rho} = \frac{S}{(\#\text{ used pages}) + k^*(\#\text{ idle pages})}$$

$$\text{rho} = \frac{S}{P * (f + k(1-f))}$$

  - where f is fraction of active (non-idle) pages, and
    - k is "idle page cost", k = 1/(1-tau) for tax rate 0 <= tau <= 1
      - at 75% tax rate, k = 4
  - So reclaim from VM with lowest rho, instead of lowest S/P

How to determine how much idle memory?
- Statistical sampling: Pick n pages at random, invalidate, see if accessed
  - If t pages touched out of n at end of period, estimate usage as t/n
  - How expensive is this? <= 100 page faults over 30 seconds negligible
- Why not just have balloon driver report on memory utilization? (p. 8)
  - Different guests keep incomparable metrics
  - Guest mostly focused on per-process utilization, not system wide
- Actually keep three estimates:
  - Slow exponentially weighted moving average of t/n over many samples
  - Faster weighted average that adapts more quickly
  - Version of faster average that incorporates samples in current period
- Use max of 3. Why?
  - Basically, when in doubt, want to respect priorities
    - Spike in usage likely means VM has "woken up"
    - Small pause in usage doesn't necessarily mean it will stay idle
- How well does this do?
  - Fig. 7 (p. 9) looks good in terms of memory utilization
    - Why isn't gap wider, given 75% tax, could expect 4:1 ratio?
      - Because VM1 only using so much memory, so would be taxed on more
  - Would be nice to see some end-to-end throughput numbers, too, though

Why not set tax rate to 100%?
- Need "buffer" against rapid working set increases (p. 8)
  - Utilization of high-priority VM can spike after wake-up event
- Anecdote: Behavior when X server paged out by low-priority simulation

What is issue with I/O pages and bounce buffers?
- Better to re-locate pages that are often used for I/O
- Store stats in pmap to identify which guest physical pages should be remapped

What happens if a guest OS uses superpages?

  If Shadow PT also used superpages, might complicate idle estimation
  But can shatter superpage in shadow PT w/o guest PT
    Might not even have available superpages for shadow PT in first place

Should shared pages cost as much as non-shared?
  Does no harm to treat them this way (consistent with shares policy)
  Could get more complications/artifacts from altering proportional share algo

How would you categorize this paper? (what is the meta-narrative)
  Experience? yes, but not great experience paper because not much data
  Big idea? sort of--statistically multiplex memory--but not super original
  Bag of tricks - really 3-4 ideas that work super well together
    Somewhat unusual compared to papers structured around a single main idea

Page revocation is a technique used in virtual memory systems to deal with memory pressure, where a page of memory is forcibly removed from a process's address space and returned to the free memory pool. This technique is typically used when the operating system detects that a page has not been accessed for a certain period of time, or when the system is running low on free memory and needs to reclaim memory from inactive or less important processes. Page revocation can be used in conjunction with other memory management techniques, such as paging and swapping, to optimize the use of memory resources and improve system performance. By reclaiming memory from inactive processes, page revocation allows the operating system to free up memory that can be used by other processes or applications, reducing the likelihood of out-of-memory errors or other performance issues.

A true page fault is forwarded to the guest OS, which might hande it by preempting a process and running another one. A hidden page fault blocks the entire virtual CPU, preventing the guest OS from doing anything useful.

Revocation

I was also thinking this was related to incoherence between the underlying system and the needs / behaviors of the user.
Yes, and I think that's why ballooning works better.
iirc, double paging was an issue from running guest LRU on top of host LRU?
Seems that the two problems are a little bit different. According to the paper, double paging refers to the situation when the meta-level policy and guest OS both want to page out the same page, this exact page will be faulted in right after it is paged out for the first time, and then immediately gets paged out again.
Double-LRU seems like a concrete approach to implementing meta-level paging. I might be wrong on this lol.

A more clear version of this question would be "give two places where scheduler activations block without notifying the user immediately" since usually even in cases where the it's not possible to send a notification, the kernel will produce the notification when it becomes possible (these are valid cases for the purposes of the question).
Some examples might be when the kernel removes a process's last processor or when a page fault in the user thread scheduler occurs. In both cases, the kernel can't generate a notification

until it reschedules the process (case 1), or until the page fault is handled (case 2). See page 64 of the paper for details.

When a process in an operating system attempts to access an address that is not in memory (a cache miss), the operating system performs a series of steps to retrieve the data from disk:

1. The operating system first checks if the address being accessed is valid or not. If it is not valid, the operating system raises an exception and terminates the process.
2. If the address is valid, the operating system checks the page table to determine if the page containing the address is in memory or not.
3. If the page is not in memory (a page fault), the operating system marks the page as invalid in the page table and initiates a page fault interrupt.
4. The interrupt handler in the operating system takes over, saves the state of the current process, and begins to service the page fault.
5. The operating system looks for a free physical page in memory to allocate to the page being faulted in. If no free pages are available, the operating system must select a page to evict from memory to make room for the new page.
6. If the page to be evicted has been modified (dirty), the operating system must write it back to disk before evicting it from memory.
7. Once a physical page has been allocated, the operating system reads the contents of the page from disk into the newly allocated physical page in memory.
8. The page table is updated to mark the page as valid and to map the virtual address to the newly allocated physical address.
9. The process is then resumed from the point of the page fault, with the required data now in memory and accessible to the process.

explain TLB miss, and how it is related with the process mentioned above

In computer architecture, a translation lookaside buffer (TLB) is a hardware cache that stores recently used mappings of virtual memory addresses to physical memory addresses. When a CPU accesses memory, it first looks up the virtual address in the TLB. If the mapping is found in the TLB, the physical address is retrieved from the cache and the memory access is completed. This process is known as a TLB hit.
However, if the virtual address is not found in the TLB, it results in a TLB miss. In this case, the CPU must perform a page table walk to find the physical address corresponding to the virtual address. This involves accessing the page table data structure in memory, which can be an expensive operation.
When the TLB miss occurs, the CPU raises a page fault exception and the operating system's memory management unit takes over. The memory management unit reads the page table data structure from memory, updates the TLB with the missing mapping, and resumes the interrupted memory access operation. This process is known as TLB refill.

I think the "initialization hack" means that we assume a variable X is not initilized until it is rd/wr by a second thread. However, this hack might lead to a bug.
In this question, we want to delete the "initilizasion hack", so we assume each variable is initailized when it is first rd/wr by the first thread. That is why we need both Write Exclusive and Read Exclusive state rather than a single Exclusive state in the original paper.