Operating Systems Notes

SubmergedDuck

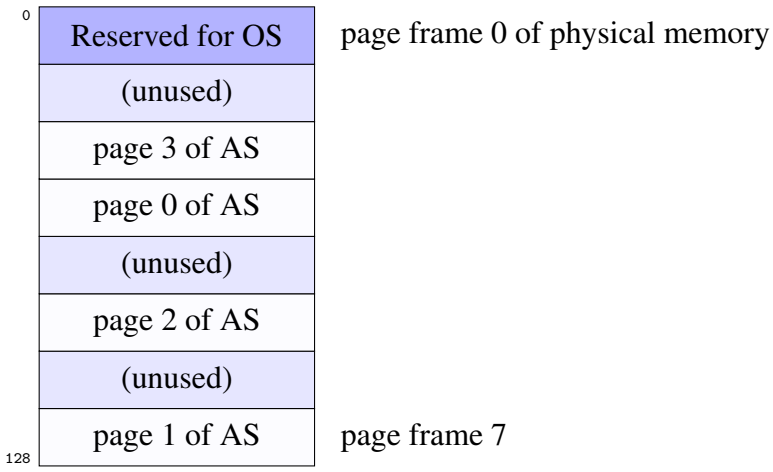
November 4, 2025

1 Page Tables: OSTEP

Paging. When memory is split into fixed-sized pieces. Inside one page, the offset tells "which byte within this page."

Page. A fixed size unit in a process's address space.

Fig 1. 64-Byte Address Space in 128-Byte Physical Memory. TBC.



Notice how each page here is $128 \div 8 = 16$ bytes each. Notice also how the address space is can be scattered throughout physical memory.

Free List. When an OS wishes to place a process's address space into physical memory, it keeps a free list of all free pages/page frames of the physical memory.

Page Table. A data structure to record each where each virtual page of the address space is located in physical memory. Each process has their own page table.

Address Translations. What the page table stores for each virtual page of the address space.

> **Eg.** Virtual Page 1 $\rightarrow$ Physical Frame 7
>
> **Eg.** To translate a virtual address to a physical address, we keep the offset (the right-most bits), but change the VPN to PFN. Using Fig 1, since the virtual page number is 1 (01 in binary) and the physical frame number is 7 (111 in binary) then our physical adress would be 111 | [offset] while our virtual adress would be 01 | [offset].

Offset. Indicates which byte a virtual address is on a virtual page.

Page Table Entry (PTE). A 20-bit VPN implies there are 2^{20} address translations for each process. We need 4 bytes per PTE to hold the physical translation plus other stuff. So for a 32-bit address space there are $2^{20} * 4$ bytes = 4MB of memory needed for each page table.

Note. So the number of page table entries is equal to the number of address translations. The number of address translations is related to how many bits a VPN has.

2 Page Tables: Other

Offset Bits. How many bits to count all bytes in one page.

Swap Offset. The index of the page-sized slot in the swap area (file/partion) where the whole page was written when it was swapped out.

> **Eg.** A 4KB page has 4096 bytes. Since $4096 = 2^{12}$, the page offset uses 12 bits.

Virtual Page Number (VPN). The index of a virtual page. These are the high bits of the virtual address.)

> **Eg.** Given a 32-bit virtual address and 4KB pages, there would be 20 bits left for the VPN (cause $4KB = 2^{12}$ bytes). This implies that there are 2^{20} virtual pages if there are 20 VPN bits.

Page Table Entry (PTE). One record in the page table that maps a virtual page to a physical frame and bits like valid/present, R/W/X permissions, dirty, accessed, etc. A single second-level page table holds 1024 PTEs.

> **Note.** Each page table entry has a Present(P) bit saying whether that virtual page is currently in RAM. $P = 1 \Rightarrow$ the mapping is valid and you can use the page table number. $P = 0 \Rightarrow$ otherwise (could be swapped out or not mapped.)

Virtual Page. A fixed-size chunk (eg. 4KB) of a process's virtual address space.

> **Eg.** With 4KB pages, every virtual address splits into a virtual page number (which page) + offset (which byte inside that page)
>
> **Eg 2.** Let virtual address = 0x12345ABC. Then the offset would be 0xABC (low 12 bits) and the VPN would be 0x12345 given 4KB pages.

Single-level Page Table. One big array indexed by the VPN; each slot is a PTE that says where that virtual page lives (which physical frame).

Second-level Page Table. Splits the VPN into PDI | PTI (10 | 10).

> **Eg.** There are 2^{20} page table entries, each entry is 4 bytes. The total size of the single page table would be $2^{20} \times 4$ bytes.

Physical Frame. A 4KB chunck of RAM (or same size as page) where a virtual page's data lives when it's in memory.

Physical Frame Number (PFN). The index of a 4KB physical frame in RAM. After page-table lookup, the PTE gives you the PFN plus status bits. Eg. Given 4KB pages, VA = [PFN | 12-bit Offset]

MiB. A mebibyte. 1 MiB = 2^{20} bytes

Page Directory Index (PDI). The first 10 bits of virtual address (VA). Selects which page table to use.

Page Table Index (PTI). The next 10 bits after PDI of VA. The Selects which PTE inside that page table.

Bit Shifting

```
#define PAGE_SHIFT 12 // 4KB Pages
uint64_t physical_address = ((uint64_t)pfn << PAGE_SHIFT)|
   offset;
```

Translation Lookaside Buffer (TLB). A tiny, very fast cache inside the MMU that stores recent virtual page number (VPN) to physical frame number translations (plus R/W/X/valid bits).

> **Hit.** MMU find the mapping in the TLB so it can form the physical address without reading the page tables from memory. Memory Accesses Needed: 1.
>
> **Miss.** MMU (or OS) must walk the page tables in memory to find the PTE, then it can access the data. Memory Accesses Needed: 3.

Memory Management Unit (MMU). The hardware between the CPU and RAM that translates virtual addresses to physical using TLB and page tables. Also enforces R/W/X permissions and raises page faults if needed and updates accessed/dirty bits.

Accessed Bit (A/R). Set by the hardware on any read or write to the page. OS uses it to tell which pages were recently used (eg. for replacement).

Dirty Bit (D/modified). Set on any write. If a dirty page is evicted, it must be written back to disk; if clean it can be dropped.

Dirty Page. A memory page that's been modified (written to) since it was loaded (its dirty bit is set.) If the OS evicts it, it must write it back to disk; a clean page (non-modified) can be dropped without writing.

Page Fault. The PTE says the page isn't in RAM (invalid/not present), thus trapping to the OS to bring the page in.

3 Lecture 14/15: Page Replacement Algorithms

Policy Decisions for Virtual Memory Management
There are three types of policies we need to know.

Fetch Policy. Policies that we use to decide when to fetch a page.

Placement Policy. Policies that we use to decide where to put the page.

Replacement Policy. Policies that we use to decide what page to evict to make room.

Page Fetch Policies. There are two types of fetch policies.

1. **Demand Paging.** This is a when we only fetch a page when we actually reference it.
2. **Prepaging.** This is when we fetch a page, we will fetch a little bit more, so when the next page fault comes around the data/content that we need for that page is already there. This is so we don't have the issue of "stalling twice."

Page Placement Policies. Modern memory managemnent systems allow for any physical frames to hold any virtual page. This means that on most computers, any of the physical frames are equally good to hold any virtual page. This also means that memory management hardware can translate any virtual-to-physical mapping equally well.

Why would we prefer some virtual-to-physical over others?

1. **NUMA Multiprocessors.** NUMA stands for non-uniform memory access. When we have a NUMA processor, some memory locations results in faster memory access than others. Eg. Any processor can access entire memory, but local memory is faster.
2. **Cache Performance.** You want to pick physical frames that does not result in bad behaviour. Thus we have to choose physical pages to minimize cache conflicts.

Note. Placement policy will have a much smaller effect than fetch and replacement polciies if memory is limited.

Page Replacement Policies. The main goal of a replacement algorithm is to reduce the **fault rate** (how frequent a page fault occurs). We want to evict pages that will never be used again in the future, but this is hard to do cause we don't know the future, so we will pick to evict the page that won't be used for the longest period of time.

How are replacement algorithms evaluated?

We take an input reference string (a list of addresses in the order that a program references them) and we count how many page faults occured, the lower the better.

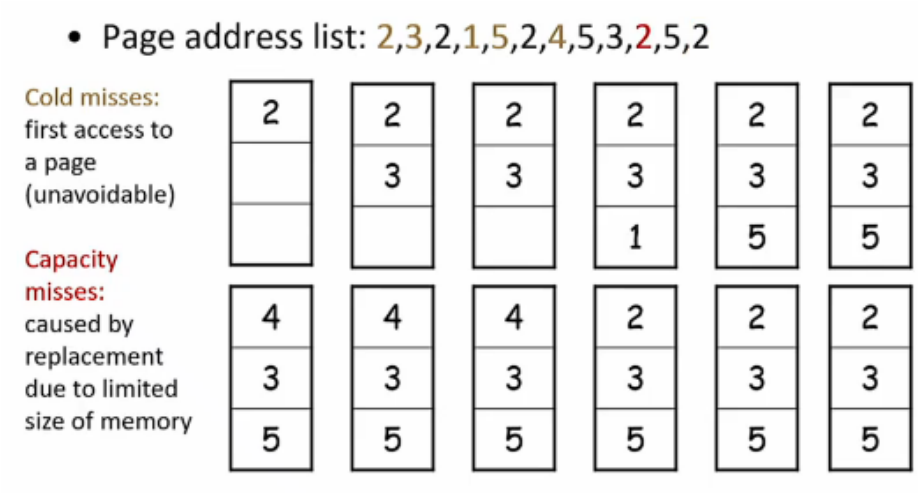
What does it mean to reference an address?

It means to use a variable or pointer to locate and access data in memory.

Timestamp: 5:19

Belady's Algorithm (OPT/MIN). This is known as the optimal **page replacement** algorithm, as it has the lowest fault rate for any page reference string. The idea is to replace the page that will not be used for the longest period of time, but the **problem** is we have to know the future perfectly.

> **Note.** This algorithm is not intended as a practical algorithm that we can use. We only use it as a "yardstick" to compare other algorithms.
>
> **Yardstick.** If Belady's/optimal is not much better, then the algorithm is pretty good, else the algorithm needs some work. Random replacement is often the lower bound (cause it's least optimal).
>
> **Modelling Belady's Algorithm**
>
> 1 **Cold Miss.** This is the very first access to a page, and it is unavoidable. No page replacement algorithms can reduce the number of cold misses.
>
> 2 **Capacity Miss.** This is a miss cause by the page replacement policy due to the limited size of memory.



Timestamp: 11:38

First-In First-Out (FIFO). In this algorithm we would maintain a list of pages in chronological order, and on replacement we would evict the oldest page in the queue.

> **Pros.** Maybe the oldest page is not being used. This queue is possible to implement in real life, and it is easy to do so.
>
> **Cons.** No information about a page except its age in memory. This algorithm also suffers from Belady's Anomaly. The fault rate might actually increase when the algorithm is given more memory.

- Page Address List: 0,1,2,3,0,1,4,0,1,2,3,4

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest			0	1	2	3	0	0	0	1	4	4

3 frames, 9 faults — Smaller memory

Anomaly	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Oldest				0	0	0	1	2	3	4	0	1

4 frames, 10 faults — Larger memory

Conditions for anomaly exist at this point.
Page 0 is in Smaller memory, but not in Larger.

Belady's Anomaly. It is when the fault rate may increase when an algorithm is given more memory.

The 2 Conditions for a Belady's Anomaly.

1 When the larger memory contains at least 1 page that is also in the smaller memory.

2 When the smaller memory contains at least 1 page that is not in the larger memory.

Page Fault. When a page is not in memory. A page fault can signal a page replacement algorithm.

Least Recently Used (LRU) The idea is that we can't predict the future, but we can guess based on passed experience. We evict the least recently used page.

Pros. This does not suffer from Belady's anomaly. Any program with high temporal locality will work well with this algorithm. Eg. code pages or stack pages are used more frequently

Cons. Vulnerable to scanning reference patterns. A program with poor temporal locality will not work well with this algorithm. Eg. A program w/ repeated sequential array access larger than avaialable memory. Also this is not easy to implement in reality.

Temporal Locality. This refers to the same page/address being accessed frequently.

Implementing Exact LRU Exact means if we were to implement this on a real computer.

Option 1. Have a timestamp so every time we reference the memory, we update the timestamp. We then evict pages with the oldest timestamp, but we'll also have to go through each page in to find one with the oldest timestamp (**Problem:** $\mathcal{O}(n)$+ larger space requirement). We'll need to make the page table entry (PTE) large enough to hold a meaningful timestamp. This may double the size of our page tables and TLBs.

Option 2. Keep pages in a queue. On reference, move the page to the front of the queue. On eviction, replace the page at the back of the queue. (**Problem:** Need costly operation to manipulate queue on every single memory reference. This requires a lot of hardware support, unless we want to fault into the OS the manipulate the queue. There's way too much overhead to figure out how to manipulate a queue on the CPU, no common CPU architectures have this support.)

Takeaway Note. Implementing exact LRU algorithm is not possible with the hardware support that we currently have.

Regarding FIFO. You only have to update the queue wait you bring a page or evict, instead of on every single memory access (that's why it's possible and easy to implement). While with an LRU, you have to update the queue everytime you access memory.

Timestamp: 25:10

Stack-Based Algorithms Algorithms that does not have Belady's Anomaly.

Eg. Least Recently Used (LRU)

Scanning Reference Patterns. Eg. Iterating through a very large array.

Comparing Replacement Policies. Follows the 80/20 "workload" rule. 80% of memory accesses are made to 20% of the "hot" pages (hit). The remaining 20% of references are made to the remaining 80% of the pages ("cold" pages).

Notes on the 80-20 Diagram. OPT is Belady's alorithm. It appears that the FIFO algorithm appears to be just as bad as the random algorith. The LRU algorithm is better than a random page replacement policy depending on the locality of the workload.

Notes on the Looping-Sequential Diagram. Refers to things like iterating through an array. LRU and FIFO performs worse than RAND and OPT. For both LRU and FIFO, if the array size is bigger than the amount of memory, the hit rate goes to 0.

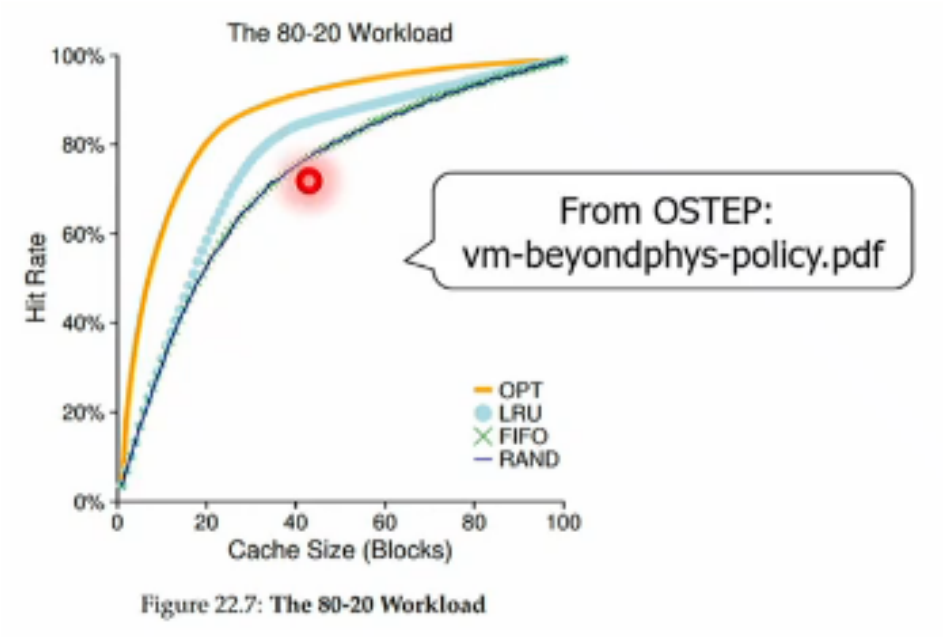


Figure 22.7: **The 80-20 Workload**

Figure 1: 80-20 Workload Diagram

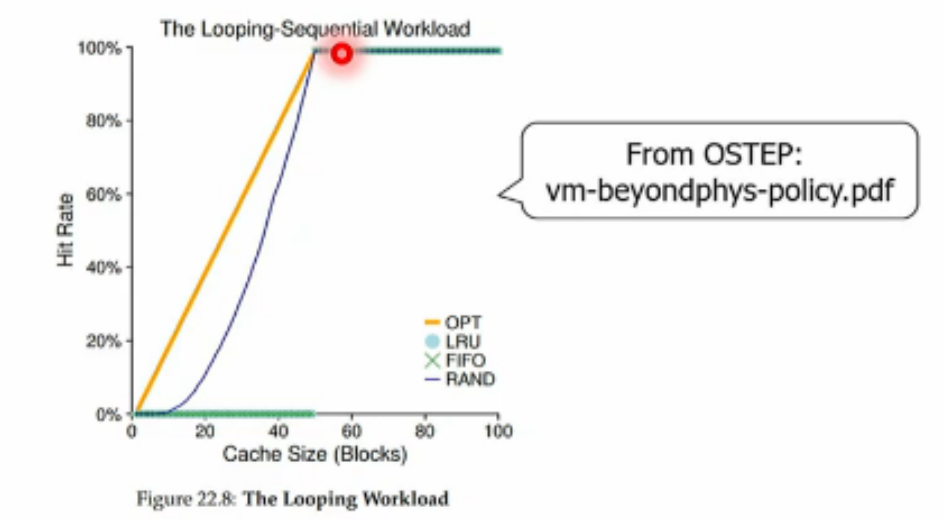


Figure 2: Looping-Sequential Workload Diagram

Timestamp: 35:34/48:15

Approximating LRU. Since exact LRU is too expensive to implement, we'll have to try something else. We make use of the PTE reference bit (R bit) instead. All R bits are initially 0, and as the processes executes, the R bit is set to 1 for pages that are accessed.

> **Problem.** We now lose the information of the exact order of references, which is bad if eviction happens infrequently (ends up degrading to FIFO). Let's say before eviction, we've accessed a lot of pages, and all of their reference bits have been set to 1, so we're basically ending up with a random eviction.
>
> **Possible Solutions to the Problem.** We try to have a period/timer where we can set all reference bits to 0, or, we can have a larger amount of reference bits (Eg. instead of just 1 bit, we can use 4/5 bits).

Second Chance Algorithm. Same as FIFO, except we look at the reference bit for the page we're about to evict. If the R bit is 0, we replace the page, if the R bit is 1, we clear the R bit and re-queue the page as youngest—then we try finding a page again to evict. The idea is that pages that are used often enough to keep reference bits set will not be replaced.

> **Note.** The second chance algorithm can be implemented in practice because the FIFOQueue is easy to implement. *In worst case scenario, this algorithm degrades to FIFO.
>
> **Not Recently Used (NRU) Algorithm.** We can combine the reference bit with the modify bit to create 4 classes of pages (Eg. prefer clean pages). This variant is called the not recently used algorithm.

Clock Algorithm. A practical implementation of the LRU algorithm. The idea is to replace a page that is "old enough" by arranging the physical frames in a circular list. Then you use a clockHandle (a cursor) to go through the circular list. As you go through each page, if the reference bit is off, then we evict the page. If the reference bit is on, then we turn the reference bit off, and check the

next page.

Cons. When the memory is large, the accuracy of the information degrades because we're not looking at the reference bits enough. *In worst case scenario, this algorithm degrades to FIFO.

What's the minimum age of a page if R bit is off? It would be the size of the clock/queue?

Recency-Based PR Policy. Replacement policies based on recency: Eg. Clock, Second-Chance, LRU.

Frequency-Based PR Policy. Also known as counting-based page replacement policies. The idea is that we will count the number of uses for each page.

Pros. An easy algorithm to implement, making it more likely to have available hardware support.

1 **Least-Frequently-Used (LFU).** We replace the page that's used least often. *Problem: pages that are heavily used at one time tend to stick around even when not needed anymore.

2 **Most-Frequently-Used (MFU).** We evict a page that's accessed a maximum number of times. *Problem: favours new pages.

Note. Neither of these 2 algorithms are common as they are poor approximations of Belady's/Optimal cause they don't respect temporal locality.

Scan-Resistant Algorithms. The idea is that we combine recency and frequency so that single-use pages can be replaced quickly (Eg. LRU/k, 2Q, ARC, LIRS, S2Q).

4 Lecture 12: Paging and Address Translation

Compaction. The idea that we have something called a physical memory (RAM) and as we are assigning physical memory to processes, we can end up with holes inbetween allocations of memory. The purpose of compaction is to eliminate the small fragments in physical memory by relocating allocated space by consolidating them so we get 1 big free space in physical memory that we can use for other processes.

> **Note.** Compaction is not possible with the C memory allocation library. The C library uses something called dynamic partitioning. It's not possible because once malloc() returns an address to you, that address would be out of reach by the malloc library—this means that the malloc library cannot change the address that was returned and update it to something else.
>
> **Dynamic Partitioning.** Opposite of fixed partitions. Video

Dynamic Relocation. Also known as execution-time binding of addresses. The purpose of this is to be able to do **swapping** and **compaction** at runtime. Dynamic relocation allows us to take a virtual address that the user process is using as its executing instructions and translating them to physical addresses.

> **Note.** This allows the OS to change the mapping between virtual and physical addresses whenever it wants—thus able to move data from one region of memory to another region of memory.
>
> **Regarding Compaction.** We are able to change some of the virtual-to-physical address translations.
>
> **What's the minimum requirements to relocate fixed or dynamic partitions?**
>
> All memory used by the process must be contiguous, which means no holes.

Relocation Registers. What some CPU architectures, commonly, the older ones have. The basic idea is to translate physical addresses by doing an add opertation to the virtual address to get the corressponding physical address.

> **Eg. Translating Addresses w/ Reloc. Reg.** Let's say base address is 10, and we want to translate the virtual address 5. Then the physical address you get is 10+5 = 15.
>
> **Note.** We also have to check that the base address is within the limits of a process's address space to protect overwriting memory that we lost to other processes. Sometimes the generated physical address does not even exist, eg. -200 physical address.
>
> **Base and Limit Reg.** On CPUs that have a "base" and "limit" registers, the MMU uses these registers to store the base address and also a limit. These registers are updated whenever we do a kernel level context switch.
>
> 1 **Base.** When we restore registers for the next process, we load the **base** register with the starting address of the physical address of that process.

2 **Limit.** Limit register is set to the last logical (virtual) address of that process. Purpose is to cap what range of physical addresses are valid.

3 **Load/Store.** When we execute any memory reference instruction like load and store, the hardware will add the base address to the logical address and we would compare that against the limit address. If the physical address ends up being smaller than the base or it goes past the limit, then we trap into the kernel for illegal address exception.

```
if (addr < base || addr >= (base + limit)){
// Trap: Illegal Address Exception (*how segfault works)
}
```

Kernel-Level Context Switch. When we switch from one process to another.

Logical Address. It's the same thing as a virtual address.

Timestamp: 8:22

Problems w/ Partitioning. Fixed partitioning can cause internal fragmentation and need for overlays. Dynamic partitioning can cause external fragmentation and problems managing available space.

> **Note.** Partitioning in general is not used in any modern systems—the common problems is because processes must be allocated to contiguous blocks of physical memory. That's why we use paging instead of partitioning.

Paging. This method eliminates external fragmentation because we will never have a situation where there is an unused hole in physical memory that we can allocate to a process as contiguouity is no longer required. Internal fragmentation is still an issue, but not that bad because its reduced to at most a fraction of one page per memory region that you have.

1 First, we logically partition physical memory into equal, fixed-size chuncks (eg. 4KB). Each of these chuncks are called **page frames.** The size of each frame is called a **page size**.

2 We then divide up a process' virtual address space into chunks of the same size as a physical frame.

Random Access Memory (RAM). It doesn't matter which address you're accessing, the speed of accessing the address will always be the same.

Timestamp: 23:40

5 Lecture 15/16: Advanced VM Functionalities

Simplified 2Q Algorithm (S2Q). This is an algorithm that combines recency and frequency to combat scanning patterns of an array. This is a type of scan-resistant algorithm to get rid of one-time use pages quickly from our main memory.

What do we do each time a page is accesed? Every time we reference a page, if the page is already in queue, we promote the page. If the page is not in queue, we add it to the A1 queue. If we see a page again, we move it to the most recently used position of the AMQueue, so it gets evicted last (lowest priority to get evicted).

A1Queue. When we have seen the page for the first time, we put in this queue. Pages in this queue can be evicted, so sometimes we put pages that have already been seen, but are already evicted in this queue.

AMQueue. This is the main queue. This main queue is an RUQueue where we keep most of the pages that are multi-use. Eg. stack pages or code pages

What do we do when we want to evict a page? The basic idea is that if the size of A1Queue is over a particular threshhold that we set, then we evict the oldest page from A1Queue. If the A1Queue is below the threshold, we kick out the least recently used page from AMQueue.

Note. In A3, the threshhold is the size of the memory divided by 10 rounded down.

Note. This algorithm is not practical for implementation because you're removing or inserting from a queue on every single reference. The reason why clock2Q algorithm is possible/practical is because we just have to update one reference bit on every single memory reference, and this is done by the hardware.

Note. The head for A1Queue is oldest page. The head for AMQueue is most recently used page.

Timestamp: 19:47 / 1:42:11

6 Lecture 11: Memory Management

Memory Management Unit (MMU). Memory system must see physical (real) addresses to fetch data or instruction. The MMU is where translation for virtual to physical addresses occur. For every virtual address, physical memory must be allocated.

> **Note.** We want to minimize wasted memory. We don't want to allocate physical memory for unused memory in the virtual address space of a process. Eg. Wasted memory is like a hello world program where there's so much space inbetween the heap and stack that's unused :(

Memory Management Goals.

1. **Virtualization.** Allow for virtualization of memory, the OS must figure out how to share the main physical real memory.
2. **Busy CPU.** We want to support enough active processes to keep our CPU busy.
3. **Memory Efficiency.** We want to minimize wasted memory.
4. **Overhead.** We want to minimize memory management overhead, which just means to minimize wasted CPU/memory. We want our data structure to track physical memory usage to be as small and as fast as possible.

The 5 Basic Requirements to Support User Processes

1. **Relocation.** Relates to the idea of information hiding or encapsulation. The goal here is we don't want to burden application programmers with is going on underneath memory management. Long-term scheduler may **swap** processes in and out of memory to relieve memory pressure. This is why we need address translation, so we can write our code into the virtual address space and translate it to the physical address space.
2. **Protection.** The OS needs to ensure that a process's memory is guarded from unwanted access by other processes, both intentional and accidental.
3. **Sharing.** Some processes may want parts of their virtual addresses to be shared and used by other processes. (Eg. The C Standard Library). We need to specify and control what sharing is allowed so a process couldn't corrupt a shared library forexample.
4. **Logical Organization.** This is about how a program uses memory space. To a machine, memory is just a 1-dimensional array of bytes, but to a programmer, we think about regions in memory (Eg. shared libraries, heap, text, stack, mapped files).
5. **Physical Organization.** We have to consider the combination of main memory and external storage devices like SSDs/hardrives as a two-level hierarchy. The challenge with this requirement is that the CPU can only access data that are in main memory (RAM) or registers.

Timestamp: 9:18

Swapping. When the OS is low on physical memory it may do something called swapping where the physical memory of a process can be written temporarily to disk to free up memory for other processes we use.

How does the OS meet the 5 requirements? Modern systems use virtual memory, a complicated technique requiring hardware and software support—based on a simpler technique: segmentation/-paging.

Address Binding Problem. This refers to the translation of symbolic names (Eg. int y;) into the final physical memory location. When we invoke the exec() system call, we load the program binary into a process's virtual address space, and the exec() system call will settup the registers and other parts of the PCB so that the process can become ready to run.

When do we assign memory accesses to variables?

> **Option 1: Compile Time Binding.** To do this, we need to know the information of the memory system that we want our program to run on because at compile time, we need to assign every variable that a process use to a fixed memory location. This is so when a process executes, it will use the exact physical memory locations. This is used in embedded systems, (Eg. Fridge). This is also a type of absolute code since the binary contains real addresses. **Note.** No relocation is possible, meaning you can't run more than one instance of a program.
>
> **Option 2: Load Time Binding** Also known as **static relocation**. In this option, the compiler translates symbolic addresses into relocatable addresses within a source file (compilation unit).

Absolute Code. If we use absolute code, we can't run more than one instance of a program because a program will overwrite main memory. (Eg. can't run more than 1 calculator program in your desktop at a time.) You can load in multiple different programs though.

Timestamp 25:11 / 1:04:54

7 Lecture 9: Scheduling

Process Scheduling. The allocation of processors (CPU) to processes/threads overtime.

> **Note.** This is the key to multiprogramming: can increase CPU utlization and system throughput by overlapping I/O and computation and make progress on multiple tasks concurrently. (Even on a single-core)
>
> **Throughput.** The amount of work/items/material passed/done by a system/CPU/process.

Timestamp: 5:05

Scheduling Building Blocks

> **Target.** Kernel-level threads.
>
> **Mechanisms.** The mechanisms behind sheduling involves the OS trackings various threads by using thread states and thread queues. Eg. Ready queue, block queue, running/execute queue.
>
> **Policies.** When we are talking about scheduling, we're talking about the policies of CPU virtualization—which thread to run next and when?
>
> **Metrics.** We need metrics to evaluate the effectivness of the scheduler and scheduling algorithms that we choose.
>
> 1 **Performance.** We look at CPU utlization, where we make sure that the CPU is kept busy when there's work to be done. We also look at throughput, where we want to maximize tasks completed per unit time.
>
> 2 **Fairness.** Ensure that each thread gets a reasonable share of the CPU time, preventing starvation.

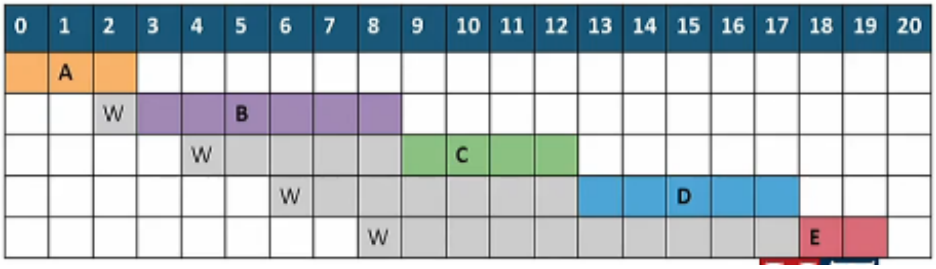
A Simple Policy. Assume that tasks run to completion when they are first scheduled, meaning we never context switch unless they perform a blocking call. Once a thread is done, they make the exit() system call.

> **Goal.** Minimize average wait time.
>
> **Arrival Time.** When a thread becomes ready to run.
>
> **Service Time.** The amount of time required to run a thread to completion.
>
> **Wait Time.** The time between a thread's arrival and being scheduled.
>
> **Turnaround Time.** The time between a threads arrival and completion— Wait Time + Service Time = Turnaround Time.

First Come First Served (FCFS) Schedules the first thread that's in the wait queue. Uses a first in first out (FIFO) queue.

Task	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Total run time: 20
- Total wait time: 23
- Average wait time: 4.6



Non-Preemptive. Once a task is scheduled in the CPU, it's going to complete with no interruption.

Con. The average wait time w/ FCFS is often quite long due to the convoy effect. Not great if you have a lot of short tasks and 1 big task paired together. If there's a **big variability** in the service time for each task, then this algorithm is pretty bad to use.

When to use?. When there's not a lot of variability in the service time for each task.

Shortest Job First SJF The next thread that is scheduled to run is the thread with the shortest service time.

Convoy Effect. It's when all other tasks wait for the one "big" task to release the CPU.

Timestamp: 19:22

8 Lecture 13/14: Translation Lookaside Buffer (TLB)

Segmented Paging. The idea that each region of the address space has its own page table.

Eg. The code region has its own page table, the stack has its own page table.

The CONS of segmented paging.

1 **Page Table Waste.** Number of valid page table entries can change over time. Eg. sbrk() allows you to adjust the size of the heap, later on we can choose to do a negative sbrk(), shrinking the size of the heap. Eg. We can have a large but sparse heap.

2 **Inflexible.** The number of regions we can have is limited by the hardware.

3 **Hardware.** We will need multiple page tables: stackPT, heapPT, dataPT, and code PT. For each region, the hardware will need to use two registers: base and limit. The limit register is used to tell us how big the segment is. Issue: we don't knowhow many regions are in a process. The hardware needed for segmented paging does not exist on any modern CPUs.

4 **External Fragmentation.** Each page table must be contiguous is some sort of memory (kernel virtual memory or physical memory), but with segmented paging, finding free space for page tables in memory is difficult—difficult to do compaction in the kernel/-physical address space.

Note. We should avoid using contiguous allocations for page tables to avoid dealing with external fragmentation. Therefore segmented paging is a BAD IDEA.

Sparse Heap Page Table. A heap page table with a lot of invalid page table entries, resulting in page table waste.

Hierarchical Page Table. Based on the idea of adding multiple levels of indirection, which is based on the intuition that we don't want to allocate memory for parts of the page table that's not in use.

Note. Also known as a multi-level page table, and is where the virtual address is broken into N + 1 pieces. The page frame is located at the bottom level page table.

Two Level Page Table.

Page Directory Base Register (PDBR). Points to the the address where the page directory, or the top level page table is located.

Page Directory. The top level page table containing pagetables. Each page table in the page directory points to a secondary page table, or a bottom page table.

Page Directory Index. The address of a page table in a page directory.

Secondary Page Table. The bottom level page table of a two level page table.

Page Table Index. The address of a page frame in the secondary page table.

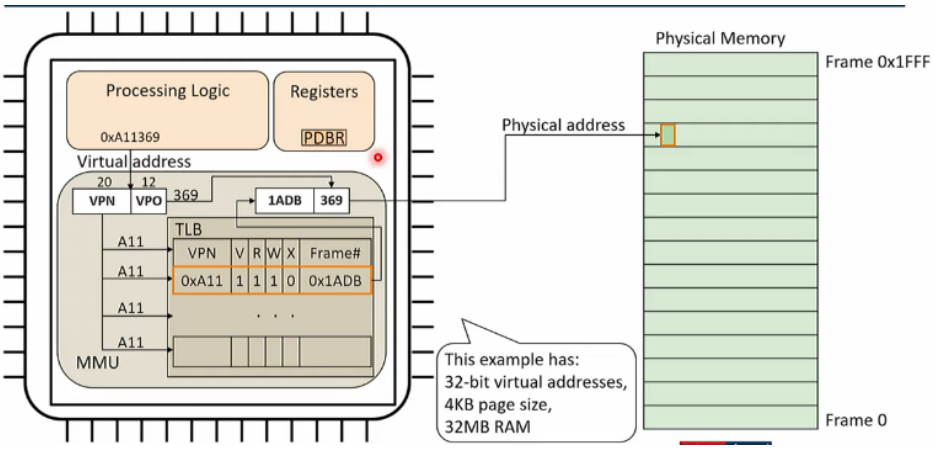
Timestamp: 30:31

9 Lecture 14: Paged Virtual Memory

Translation Lookaside Buffer (TLB). The input to the TLB is the VPN, and the output is the PFN. This is assuming that the valid bit is set in the PTE and that the operation (R/W) is permitted.

> **Note.** The TLB will return an error if we try to write to a page that is set to read only.
>
> **Note 2.** Because the TLB is very small we have to use it effectively, and so we need to do address translations most of the time—get more hits. Doing translations with a page table is much slower than doing it in the TLB. Eg. w/ a 4-level page table, we need 5 memory accesses, and if each memory access is 100 machine cycles, thats 500 machine cycles compared to 1 machine cycle.

Address Translation w/ TLB.



> **Processing Logic.** Contains the arithmetic logic unit (ALU) to perform the fetch, decode, execute cycle.
>
> **Registers.** Contains the page directory base register (PDBR). The PDBR stores a pointer to the page directory that we need to find the page table in memory.
>
> **MMU.** Contains the TLB. The TLB will look up the VPN (0xA11 in this example) in all the TLB entries in parallel. The TLB will then use the frame number stored in the 0xA11 VPN TLB entry. In the diagram, since X = 0, if we try to execute an instruction located at the virtual address, a protection fault will occur.
>
> **What happens when we get a TLB miss?** We use the PDBR to find the page table and then walk the page table to find the page table entry (PTE). After finding the PTE we would update the TLB with a new entry.

Principle of Locality (Locaility of Reference). We get TLB hits most of the time because of this principle. When we look at addresses that are referenced by a process in execution, it turns out only a handful of pages are used at one time, and we only need to have fast translations for the pages that the running process is currently using.

Hits. Typically >99% of translations are hits.

Misses. When we miss a translation it's called a **TLB miss** or **TLB fault**.

There are 2 ways to handle TLB misses. Who is responsible for loading the missing translation into the TLB?

1 **Hardware Loaded.** In this scenario the MMU (Eg. Intel Pentium) is able to look up the page table in memory, find the entry that's requested, and load the entry into the TLB. Because the hardware needs to access the page table for the current running process, the OS has to settup the PTBR when swtiching to a new process. The OS is also responsible for building and maintaining page tables in memory. The CPU will read in the page table and perform TLB faults.

   Note. Hardware reads the page table, but doesn't write to the page table. We only write to the page table as an OS.

2 **Software Loaded.** Gives us more flexibility because the OS gets to decide the page table format and the PTE format. When there's a TLB miss, we fault into the OS—the exception handler is responsible for looking up the page table to find the appropriate translation to load into the TLB.

   Note. The software/OS is both reading and writing to the page table.

Note. Software loaded TLB is much slower than hardware loaded TLB.

OS. Responsible for creating page tables.

Timestamp 6:37

Managing TLBs. *A context switch is 500x more expensive.

1 **OS.** OS ensures that TLB and page tables are consistent. When OS changes the protection bits of a PTE, it needs to update the copy of the PTE in the TLB (if it exists).

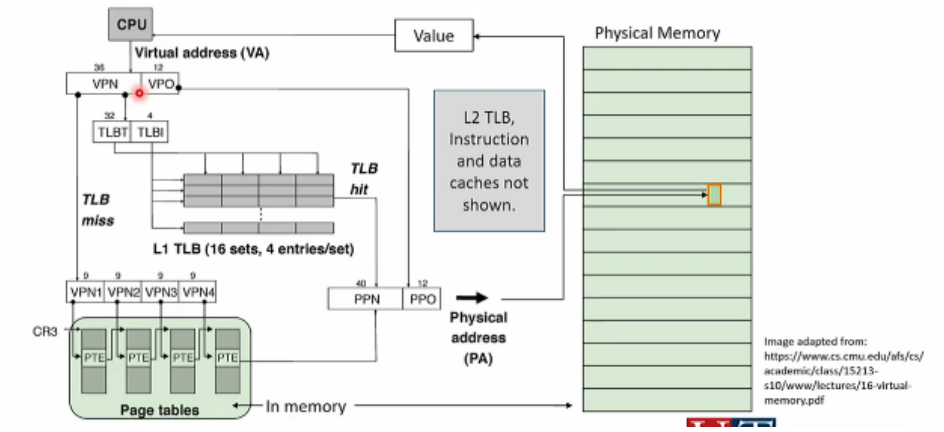
2 **Context Switch.** We reload the TLB on a process context switch, invalidating all entries—this make context switches expensive (cause there's going to be a bunch of TLB misses at the beginning).

3 **Cached PTE Eviction.** When a TLB misses and a new PTE must be loaded, a cached PTE likely needs to be evicted. Choosing which PTE to evict is called the TLB replacement policy. Implementing in hardware for hardware-loaded TLB is often simple.

Intel Core i7 Address Translation.

1 Look up page table and check the TLB for a matching entry at the same time. If we have a TLB hit, then we cancel page table lookup.

   Note. Looks like Intel Core i7 uses a 4-level page table.



Paged Virtual Memory. The idea of implementing paging. The OS uses main memory as a page cache of all data allocated in the system.

1 Initially, pages are allocated from memory. When memory fills up, allocating a page in memory requires some other page to be evicted from memory.

2 Evicted pages go to the **swap** space. Swap is a location on disk.

3 OS keeps track of use of each physical frame with a **coremap** data structure.

Swapping. The action of moving the contents of a page into disk.

Page Fault. This happens when the page being accessed is invalid (PTE is invalid). A **page fault handler** deals with page faults and tries to make pages valid.

1 **Minor Page Fault.** Happens when disk access is not needed. Eg. first access to a new stack or heap page.

2 **Major Page Fault.** Happens when disk access is needed. Eg. First access to a code or data region, access to evicted pages (remember that evicted pages go to swap)

Note. Page fault only happens when both the page table and the TLB are invalid for that address translation (TLB and PTE both have a valid/invalid bit).

Eviction. When OS runs out of memory, we need to evict something. A victim fram ei chosen from the **coremape**. The PTE of the page mapped to the frame is set to invalid, if the PTE is cached in TLB, we must also invalidate the TLB entry. The **swap location** of the page is then *stored in the PTE.

What happens when a process accesses a page that's been evicted? The invalid PTE will cause a page fault (trap), the trap will run the OS page fault handler, the handler uses the invalid PTE to locate page in swap (on disk). The page is then read into a physical frame, and the PTE is update to point to the page. The process finally resumes.

Note. The page fault is generated by the CPU.

Timestamp: 33:57

Locality

10 Lecture 8: Concurrency Bugs

The Three Common Concurrency Bugs.

1 **Atomicity Violation.** A bug caused by a lack of mutual exclusion inside the critical section. It happens cause the "desired serializability among multiple memory accesses is violated (eg. a code region should be atomic, but is not). Harder to find cause it does not cause a crash or hang.

Eg. Data in your application starts to corrupt slowly.

2 **Order Violation.** A bug caused by incorrect ordering of operations. Also harder to find because of the same reasons for atomicity violation bugs. Also data starts to corrupt slowly too.

3 **Deadlock.** How to spot: All of the threads would just stop working, when we gdb and take a look, all the threads are sleeping and waiting for some event to happen, but nothing happens.

Eg. Your entire website or application just freezes.

Serializability. This is a correctness guarantee that the execution of concurrent operations would be equivalent to some sequential execution of the same operations.

Note. Basically the guarantee that mutual exclusion provides.

Eg. T_A executes instructions $a_1, a_2, ..., a_n$ to modify shared data. These instructions are atomic. T_B executes $b_1, b_2, ..., b_n$ to modify the same shared data. The outcome of running T_A and T_B can only be $a_1, a_2, ..., a_n, b_1, b_2, ..., b_n$ or the otherway around (T_B first then T_A).

Atomicity Violation: Buggy Code [MySQL]

```
1          Thread 1::
2          -- Check if thd.proc_info is null
3          if (thd->proc_info) {
4              -- If not null, write into thd.proc_info
5              fputs(thd->proc_info, ...);
6          }
7
8          Thread 2::
9          -- Set thd.proc_info to null
10         thd->proc_info = NULL;
```

1 **What's wrong with this code?** An interleaving can happen, creating an atomicity violation. Line 3 -> Line 10 -> Crashed Line 5 (cause we can't pass null pointer) is the atomicity violation.

2 **How can we fix it?** Using the same atomic lock() and unlock() on both critical sections.

Timestamp: 13:35

11 Lecture 13/14: Page Table Design

Paged Address Translation

```
page_number = vaddr / page_size;
page_offset = vaddr % page_size;
frame_number = page_table.translate(page_number);
paddr = frame_number * page_size + page_offset;
```

Example of Address Translation. Suppose addresses are 16 bits, pages are 1024 bytes (1KB page size).

Offset. The offset will be 10 bits since each page has 2^{10} bytes.

Page Number. The page number will be $16 - 10 = 6$ bits.

Maximum number of pages. $2^6 = 64$

Size of the virtual address. $2^{16} = 65,536 = 64\text{KB}$

Note. Every single byte of a page needs an address.

Example of Address Translation in Code. Using bit-wise operators. Given 0x31DE virtual address, 1KB page size.

```
// Extract Page Number
VPN = vaddr >> 10; // Same as vaddr / 1024 but faster

// Get Frame Number From PageTable
PFN = process->page_table[VPN];

// Combine Frame Number w/ Page Offset
offset = vaddr & 0x3FF; // Same as vaddr % 1024
paddr = (frame << 10) | offset;
```

Bit Mask.

```
// Get the bit Mask
page_mask = (1 << offset_bits) - 1;
```

Timestamp: 17:03

12 Lecture 8: Deadlock

Deadlock Prevention. The main idea for dealing with deadlock is to break on of the four conditions to avoid deadlock.

The 4 Conditions. Mutual exclusion, hold and wait, no pre-emption, and circular wait.

1 **Breaking Mutual Exclusion.** We can break mutex with something called lock-free data structures. (Eg. Use hardware atomic Compare-And-Swap instruction.)

2 **Breaking Hold and Wait.** We can possibly break hold and wait by locking resources or using the trylock() function.

3 **No Pre-emption.** We can only break pre-emption if a resource has a rollback state, such as the CPU as you can save context, otherwise it's not feasible to break no pre-emption.

4 **Circular Wait.**

Breaking Mutex: Compare-And-Swap (CAS). Use CAS instead of locks to break mutex. If you need more than 1 address to update a value to, then a lock-free data structure is not possible.

```
1  int CAS(int *address, int expected, int new);
2
3  // Blocking Atomic Add
4  void AtomicAdd(int *val, int a){
5      lock(L);
6      *val += a;
7      unlock(L);
8  }
9
10 // Non-blocking Atomic Add
11 // If the address val, still contains the old value, then
      CAS will return true.
12 void AtomicAdd(int *val, int a){
13     do {
14         int old = *val;
15     } while (CAS(val, old, old + a) == 0);
16 }
```

Breaking Block and Wait. Hold and Wait policy tries to gather all needed resource at the same time, which can lead to long delays especially if one or more resources are not readily avaialble—leading to threads stalling, and the wait time to make any progress in the critical section can increase dramatically.

1 **Another Problem: Locked Resources.** We're forced to lock all resources at the beginning, before the resource is actually needed, limiting concurrency, and resulting in blocking many other threads.

2 **Another Problem: Resource Requirements.** We may not know all the resource requirements in advanced.

3 **Alternative.** Some thread libraries offer a trylock() function—you grab a lock if it's avaialable, else we try again later. Deadlock is avoided. Problem: We could have a possible livelock.

```
// How we can break hold and wait?

Record reci = get_record(db, pki) // Gets a database record
reci.lock(); // Lock resource pki

int i = get_value(reci, "i", int); // Gets shared int

// If int < 0, grab resource B (pkb), else grab A (pka)
Record recx = get_record(db, i<0 ? pkb : pka);
recx.lock(); // Lock resource either pkb or pka


int x = get_value(recx, "cost", int);
inc_value(recx, "views"); // Increment view of x
recx.unlock();
set_value(reci, "i", i + x); // Increment cost x by i
reci.unlock();
```

```
// Trylock() Function
// We will keep trying until we have both locks ready at
   the same time.
// Deadlock avoided.

ready = false;
while (!ready){
    lock(L1);
    if (trylock(L2) == -1) // If L2 is available
        unlock(L1);
    else{
        ready = true;
    }
}
```

Timestamp: 17:19

Livelock. A case where all of the threads are not blocked, but at the same time, no progress is being made.

Eg. Trylock(). In a trylock() threads can get stuck waiting to be ready/acquire multiple locks at the same time.

Solution. Using something called exponential backlog, where each time we want to try acquiring the lock again, we wait for a longer period of time to try again.

Breaking No Pre-emption. No pre-emption means we're force-taking resources from a thread. The main idea is that taking away a resource from another thread is only safe when we can rollback a resource to a previous/saved state.

When is it safe to take a resource away from a thread? It depends on the resource. We can take away the CPU, but not a data structure like linked list. The CPU is safe because we save the context when we take the CPU.

Note. Breaking pre-emption isn't really feasible, or it's too compelx to achieve.

Breaking Circular Wait. To break circular wait, we assign a **linear ordering** to resource types and require that a thread holding a resource of one type, R, can only request resources that follow R in the ordering.

Lock Ordering Eg. Must lock R_1 before R_2 before R_3.

Dining Philosophers Problems. You have a round table of philosophers, there's only 1 chopstick on each side of a philosopher, and to eat they need 2 chopsticks.

```
1  // Let each philosopher be a thread
2
3  void philosopher(int pid)
4  {
5      int left = pid;
6      int right = (pid + 1) % N;
7
8      while (true){
9          think();
10         lockAcquire(chopsticks[left]);
11         lockAcquire(chopsticks[right]);
12
13         eat();
14
15         lockRelease(chopsticks[right]);
16         lockRelease(chopsticks[left]);
17     }
18 }
19
20 // Would this result in a deadlock? Yes it would.
21 // Why?: Because each philosopher is able to acquire locks
   to chopsticks on their left. If there's 5 chopsticks, and
    5 philosophers, and each philosopher already acquire 1
   chopstick, then no philosopher can make progress eating
   as they can't acquire the chopstick on the right.
```

Lock Ordering: Dining Philosophers. A solution to prevent deadlock by preventing circular wait.

```
1   struct lock *chopsticks[N];
2   void philosopher(int pid){
3       int left = pid;
4       int right = (pid + 1) % N; // The "circular part"
5
6       // Breaking Circular Wait: Lock Ordering //
7       vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
8
9       // If left < right, take the left chopstick, else take
            right.
10      int first = (left < right) ? left : right; // L1
11
12      // If left < right, take right chopstick, else take
            left.
13      int second = (left < right) ? right : left; // L2
14
15      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
16      // Breaking Circular Wait: Lock Ordering //
17
18      while (true){
19          think();
20          lockAcquire(chopsticks[first]);
21          lockAcquire(chopsticks[second]);
22
23          eat();
24
25          lockRelease(chopsticks[second]);
26          lockRelease(chopsticks[first]);
27      }
28  }
```

Timestamp: 33:41