Design philosophy of operating systems (I)

Hung-Wei Tseng

Recap: von Neumman Architecture

By loading different programs into memory, your computer can perform different functions

509cbd23

00c2e800

Processor

Program

Instructions	Data
0f00bb27	00c2e800
00005d24	00c2f000
0000bd24	00000008
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Memory

Instruction	Data
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

What happens when creating a process

registers

R0
R1
R2
⋮
R3

code

static data

heap

Dynamic allocated data: `malloc()`

stack

Local variables,
arguments

Virtual memory

code

static data

program

The operating system
needs to track all of these for
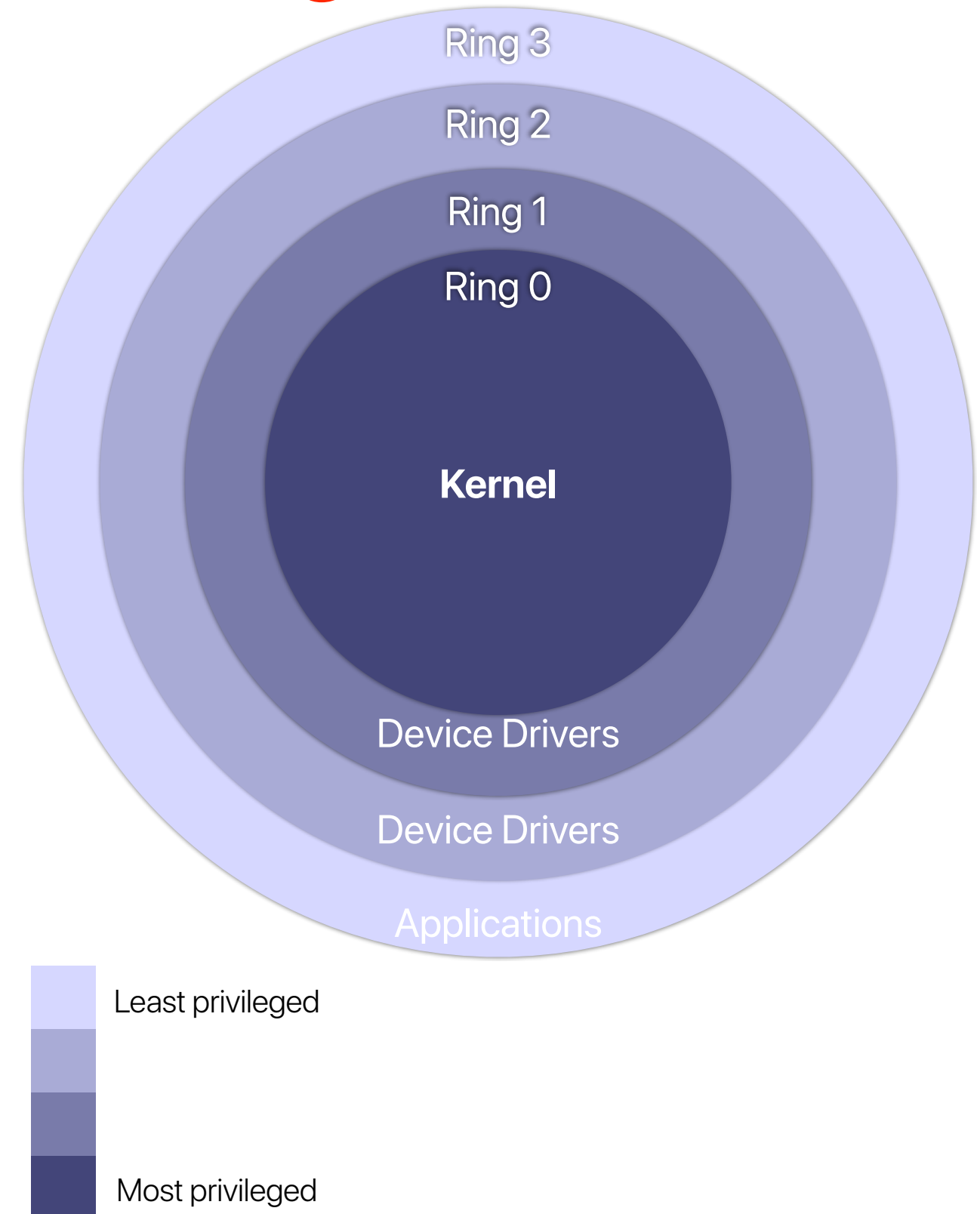each process!

Linux contains a .bss section
for uninitialized global variables
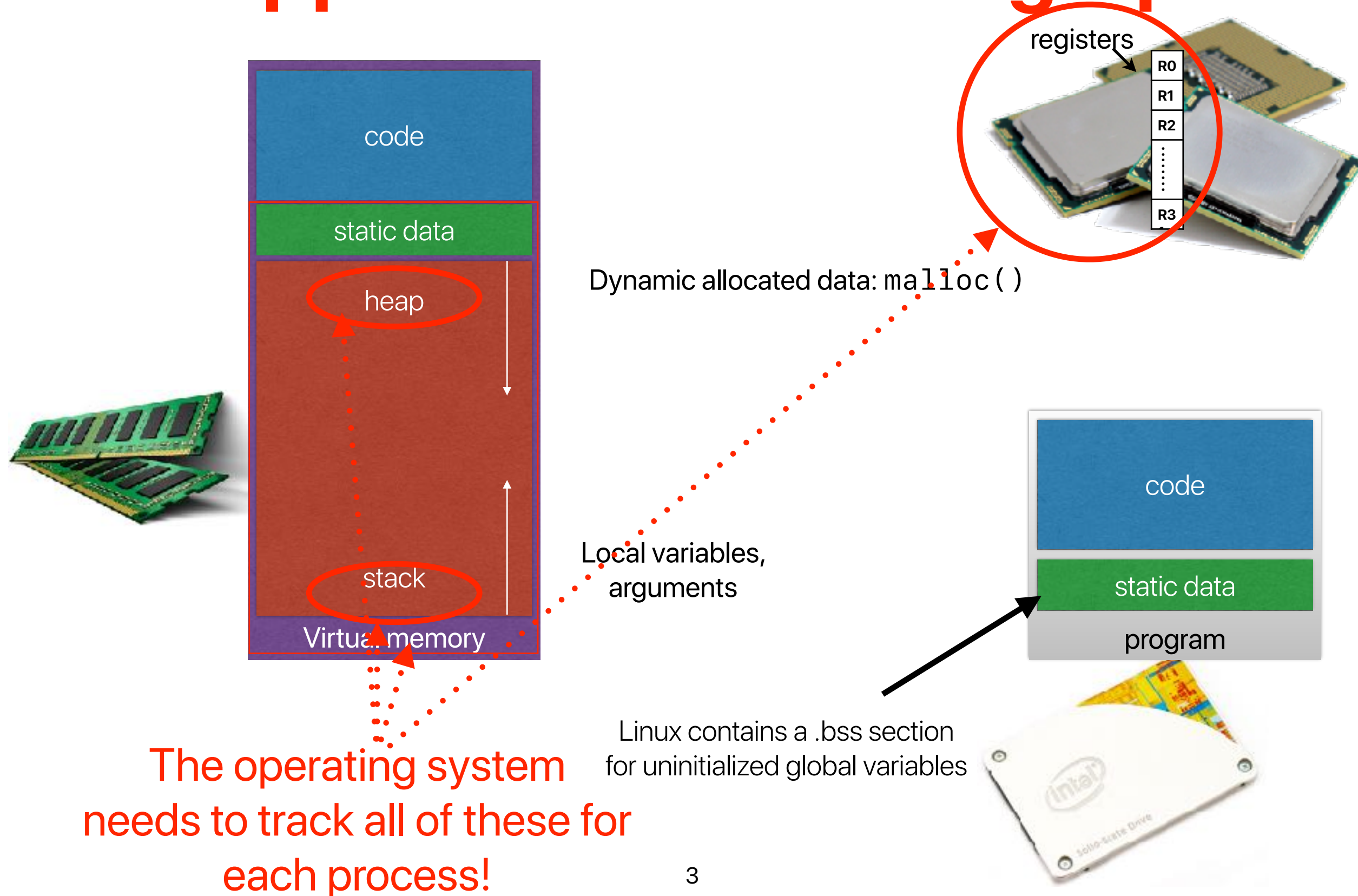
Recap: Restricted operations

- Most operations can directly execute on the processor without OS's intervention

- The OS only takes care of protected resources, change running processes or anything that the user program cannot handle properly

- Divide operations into two modes

  - User mode

    - Restricted operations

    - User processes

  - Kernel mode

    - Can perform privileged operations

    - The operating system kernel
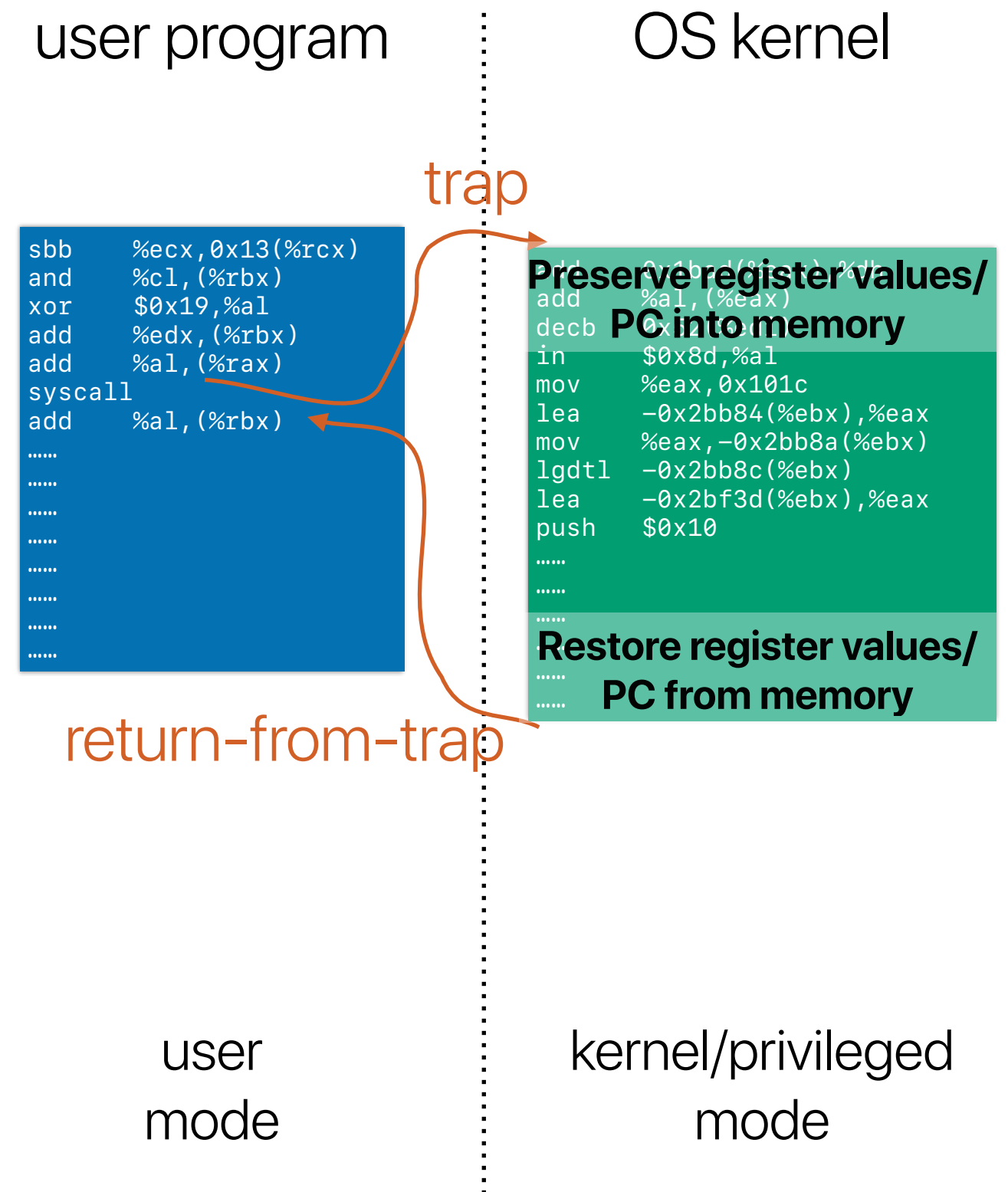
- Requires architectural/hardware supports

Recap: Architectural support: privileged instructions

- The processor provides **normal** instructions and **privileged** instructions
  - Normal instructions: ADD, SUB, MUL, and etc ...
  - Privileged instructions: HLT, CLTS, LIDT, LMSW, SIDT, ARPL, and etc...
- The processor provides different modes
  - User processes can use normal instructions
  - Privileged instruction can only be used if the processor is in proper mode — otherwise, it incurs an **exception** and the OS handler needs to deal with it

Ring 3
Ring 2
Ring 1
Ring 0

Kernel

Device Drivers

Device Drivers

Applications

Least privileged

Most privileged

Recap: How applications can use privileged operations?

- Through the API: **System calls**

- Implemented in "trap" instructions
  - Raise an exception in the processor
  - The processor saves the exception PC and jumps to the corresponding exception handler in the OS kernel

user program   OS kernel

trap

```
sbb     %ecx,0x13(%rcx)
and     %cl,(%rbx)
xor     $0x19,%al
add     %edx,(%rbx)
add     %al,(%rax)
syscall
add     %al,(%rbx)
……
……
……
……
……
……
```

Preserve register values/ PC into memory
```
add     %al,(%eax)
decb    
in      $0x8d,%al
mov     %eax,0x101c
lea     -0x2bb84(%ebx),%eax
mov     %eax,-0x2bb8a(%ebx)
lgdtl   -0x2bb8c(%ebx)
lea     -0x2bf3d(%ebx),%eax
push    $0x10
……
```

Restore register values/ PC from memory

return-from-trap

user mode   kernel/privileged mode

6

Recap: Kernel

- The OS kernel only get involved when necessary
  - System calls
  - Hardware interrupts
  - Exceptions
- The OS kernel works on behave of the requesting process — not a process
  - Somehow like a function call to a dynamic linking library
  - The the process' PC set to the invoked kernel code and enters **kernel mode**
  - Kernel code preserve the current architectural states and update the PCB
  - As a result — overhead of copying registers, allocating local variables for kernel code and etc…

Outline

- Kernel (cont.)
- How to read research papers
- The Structure of the 'THE'-Multiprogramming System
- The Nucleus of a Multiprogramming System

The overhead of kernel

Latency Numbers Every Programmer Should Know (2020 Version)

Operations	Latency (ns)	Latency (us)	Latency (ms)	
L1 cache reference	0.5 ns			~ 1 CPU cycle
Branch mispredict	3 ns			
L2 cache reference	4 ns			14x L1 cache
Mutex lock/unlock	17 ns			
Send 2K bytes over network	44 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	2,000 ns	2 us		
Read 1 MB sequentially from memory	3,000 ns	3 us		
Read 4K randomly from SSD*	16,000 ns	16 us		
Read 1 MB sequentially from SSD*	49,000 ns	49 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from disk	825,000 ns	825 us		
Disk seek	2,000,000 ns	2,000 us	2 ms	4x datacenter roundtrip
Send packet CA-Netherlands-CA	150,000,000 ns	150,000 us	150 ms	

https://colin-scott.github.io/personal_website/research/interactive_latency.html

Demo: Kernel Switch Overhead

- Measure kernel switch overhead using lmbench http://www.bitmover.com/lmbench/

How does the processor knows where to jump to?

power on/boot

install trap tables using privileged instructions

kernel mode

system call handlers

system call

system call

user process

user process

user mode

15
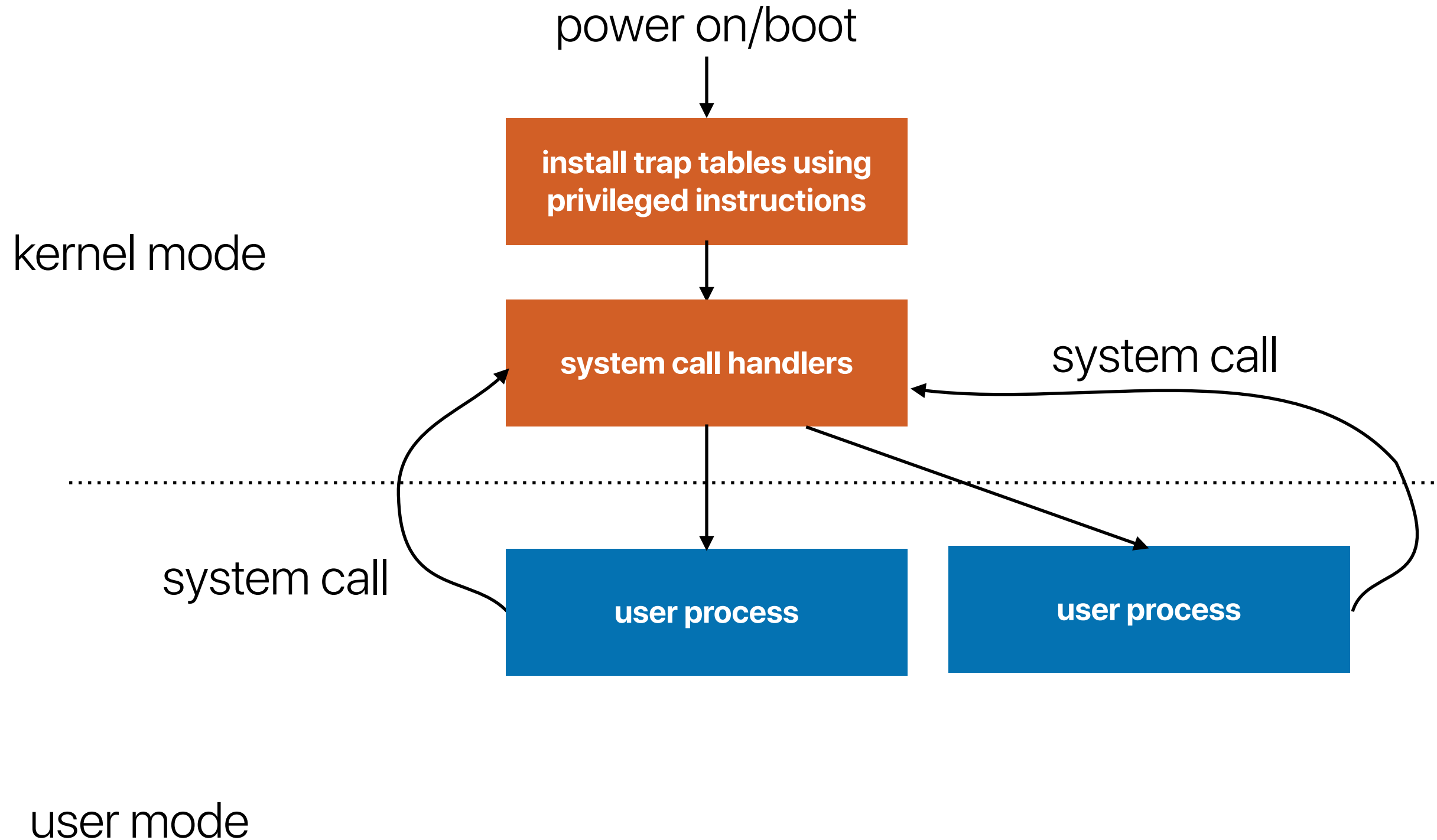
How to read research papers

How to read research papers

- For each paper, you should identify the followings:

  - Why? **The most important thing when you're reading/writing a paper**
    - Why should we care about this paper?
    - What's the problem that this paper is trying to address?

  - What? **The second most important thing when you're reading/writing a paper**
    - What has been proposed?
    - Contributions of the paper

  - How? **They are important only if you want to implement the proposed idea**
    - How does the paper accomplish the proposed idea?
    - How does the result perform?

Recap & Brainstorm

- What are those related papers that you read before?

- Compare with those related papers and re-exam their **whys**, **whats** and hows

- **What will you propose** if you're solving the same "why"?

Why is reading papers important

- As a researcher
  - You want to identify important problems
  - You want to know what has been accomplished
- As an engineer
  - You want to know if there is a solution of the design problems of your systems, applications
  - You want to know if you can apply the proposed mechanism
  - You want to know how to do it

The Structure of the 'THE'-Multiprogramming System

Edsger W. Dijkstra
Technological University, Eindhoven, The Netherlands

Edsger W. Dijkstra

- 11 May 1930 – 6 August 2002
- Dijkstra's algorithm (single-source shortest path problem)
- Synchronization primitive, Mutual exclusion, Critical sections — appendix of this paper
- Dining philosophers problem
- Program verification
- Multithreaded programming
- Concurrent programming
- Dijkstra–Scholten algorithm
- ......
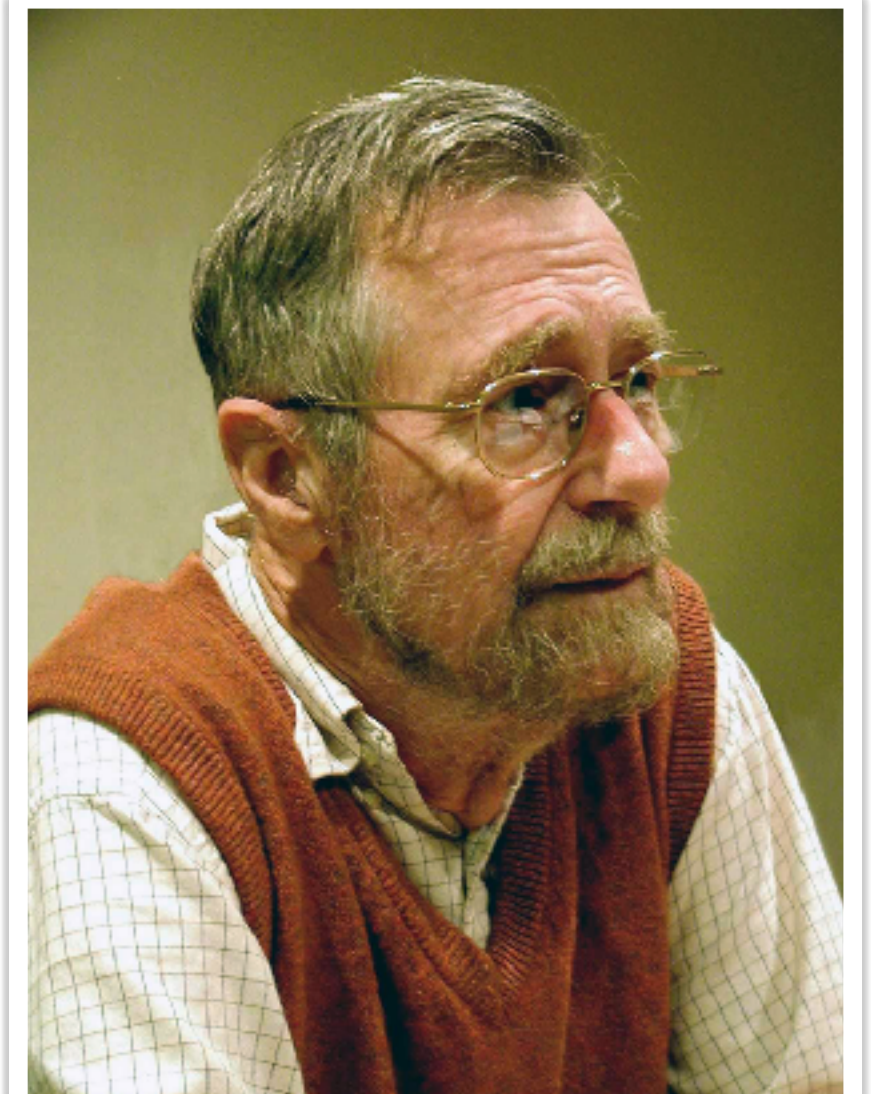
Where is why?

(3) Be aware of the fact that experience does by no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your previous experiences.

Usually, you should be able to identify the **why** in the very beginning part of a paper

The primary goal of the system is to process smoothly a continuous flow of user programs as a service to the University. A multiprogramming system has been chosen with the following objectives in mind: (1) a reduction of turn-around time for programs of short duration, (2) economic use of peripheral devices, (3) automatic control

I shall not deny that the construction of these testing programs has been a major intellectual effort: to convince oneself that one has not overlooked "a relevant state" and to convince oneself that the testing programs generate them all is no simple matter. The encouraging thing is that (as far as we know!) it could be done.

of backing store to be combined with economic use of the central processor, and (4) the economic feasibility to use the machine for those applications for which only the flexibility of a general purpose computer is needed, but (as a rule) not the capacity nor the processing power.

The system is not intended as a multiaccess system. There is no common data base via which independent users can communicate with each other: they only share the configuration and a procedure library (that includes a translator for ALGOL 60 extended with complex numbers). The system does not cater for user programs written in machine language.

showed up during testing were trivial coding errors (occurring with a density of one error per 500 instructions), each of them located within 10 minutes (classical) inspection by the machine and each of them correspondingly easy to remedy. At the time this was written the testing had not yet been completed, but the resulting system is guaranteed to be flawless. When the system is delivered we shall not live in the perpetual fear that a system derailment may still occur in an unlikely situation, such as might result from an unhappy "coincidence" of two or more critical occurrences, for we shall have proved the correctness of the system with a rigor and explicitness
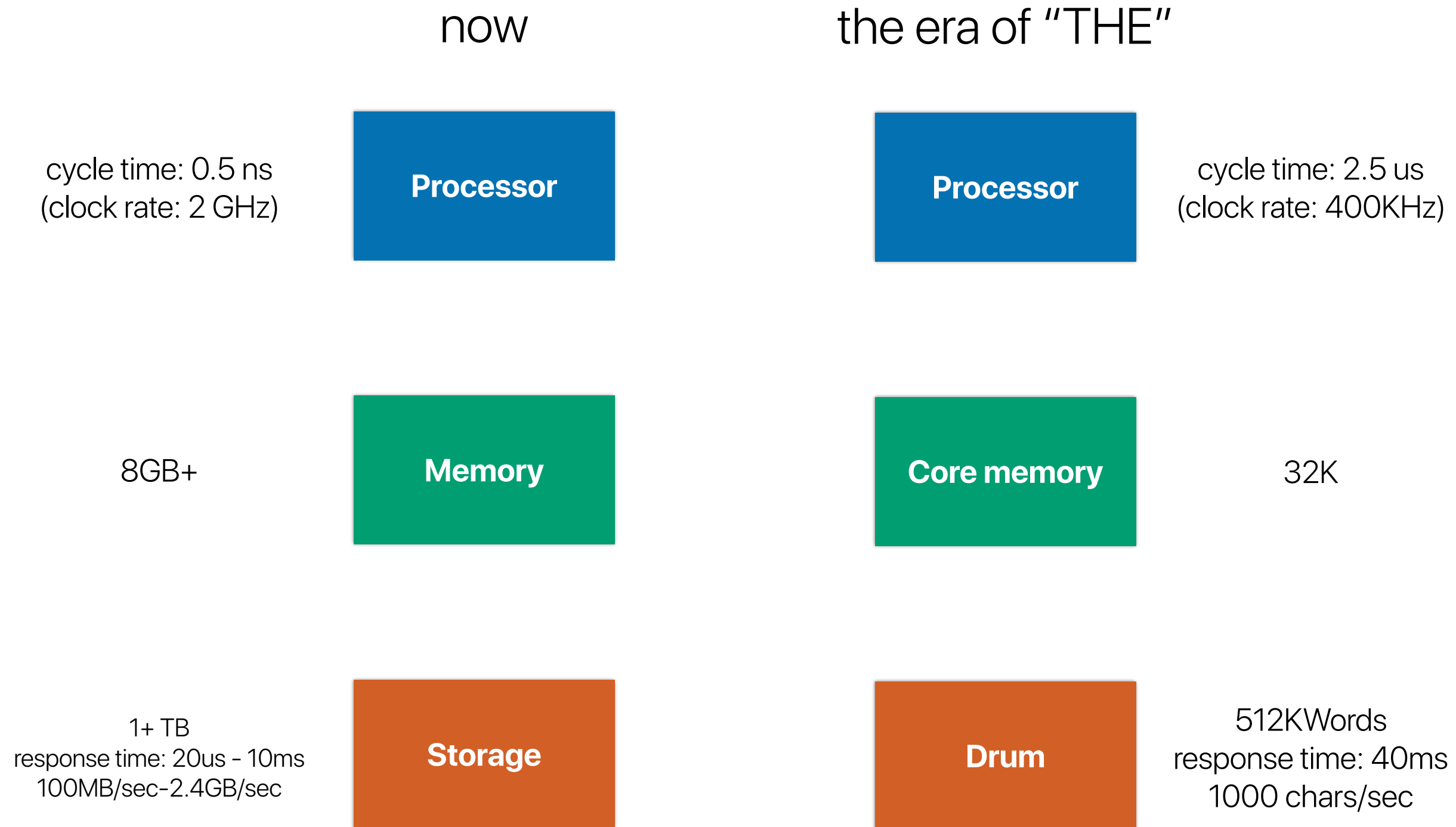
Performance Metrics

- CPU utilization — how busy we keep the CPU to be — CPU is busy does not mean it's making progress for the user program

- Latency — the time between **start execution** and **completion**

- Throughput — the **amount** of "tasks/processes/threads" that we can finish **within a given amount of time**

- Turnaround time — the time between **submission/arrival** and **completion**

- Response time — the time between **submission** and **the first time when the job is scheduled**

- Wait time — the time between **the job is ready** (not including the overhead of queuing, command processing) and **the first time when the job is scheduled**

- Fairness — every process should get a fair chance to make progress

THE

- Why should people care about this paper in 1968?
  - Turn-around time of **short** programs
  - Economic use of peripherals
  - Automatic control of backing storage
  - Economic use of the machine
  - Designing a system is difficult in 1968
    - Difficult to verify soundness
    - Difficult to prove correctness
    - Difficult to deal with the complexities

The computer in the era of "THE"

now   the era of "THE"

cycle time: 0.5 ns   cycle time: 2.5 us
(clock rate: 2 GHz)   **Processor**   **Processor**   (clock rate: 400KHz)

8GB+   **Memory**   **Core memory**   32K

1+ TB   512KWords
response time: 20us – 10ms   **Storage**   **Drum**   response time: 40ms
100MB/sec–2.4GB/sec   1000 chars/sec

processes

Where is what?

strict layered design

A Survey of the System Structure

Storage Allocation. In the classical von Neumann machine, information is identified by the address of the memory location containing the information. When we started to think about the automatic control of secondary storage we were familiar with a system (viz. GIER ALGOL) in which all information was identified by its drum address (as in the classical von Neumann machine) and in which the function of the core memory was nothing more than to make the information "page-wise" accessible.

We have followed another approach and, as it turned out, to great advantage. In our terminology we made a strict distinction between memory units (we called them "pages" and had "core pages" and "drum pages") and corresponding information units (for lack of a better word we called them "segments"), a segment just fitting in a page. For segments we created a completely independent identification mechanism in which the number of possible segment identifiers is much larger than the total number of pages in primary and secondary store. The segment identifier gives fast access to a so-called "segment variable" in core whose value denotes whether the segment is still empty or not, and if not empty, in which page (or pages) it can be found.

As a consequence of this approach, if a segment of information, residing in a core page, has to be dumped onto the drum in order to make the core page available for other use, there is no need to return the segment to the same drum page from which it originally came. In fact, this freedom is exploited: among the free drum pages the one with minimum latency time is selected.

A next consequence is the total absence of a drum allocation problem: there is not the slightest reason why, say, a program should occupy consecutive drum pages. In a multiprogramming environment this is very convenient.

Processor Allocation. We have given full recognition to the fact that in a single sequential process (such as can be performed by a sequential automaton) only the time succession of the various states has a logical meaning, but not the actual speed with which the sequential process is performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with undefined speed ratios. To each user program accepted by the system corresponds a sequential process, to each input peripheral corresponds a sequential process (buffering input streams in synchronism with the execution of the input commands), to each output peripheral corresponds a sequential process (unbuffering output streams in synchronism with the execution of the output commands); furthermore, we have the "segment controller" associated with the drum and the "message interpreter" associated with the console keyboard.

This enabled us to design the whole system in terms of these abstract "sequential processes." Their harmonious cooperation is regulated by means of explicit mutual synchronization statements. On the one hand, this explicit mutual synchronization is necessary, as we do not make any assumption about speed ratios; on the other hand, this mutual synchronization is possible because "delaying the progress of a process temporarily" can never be harmful to the interior logic of the process delayed. The fundamental consequence of this approach—viz. the explicit mutual synchronization—is that the harmonious cooperation of a set of such sequential processes can be established by discrete reasoning; as a further consequence the whole harmonious society of cooperating sequential processes is independent of the actual number of processors available to carry out these processes, provided the processors available can switch from process to process.

System Hierarchy. The total system admits a strict hierarchical structure.

At level 0 we find the responsibility for processor allocation to one of the processes whose dynamic progress is logically permissible (i.e. in view of the explicit mutual synchronization). At this level the interrupt of the real-time clock is processed and introduced to prevent any process to monopolize processing power. At this level a priority rule is incorporated to achieve quick response of the system where this is needed. Our first abstraction has been achieved; above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor that had lost its identity having disappeared from the picture.

At level 1 we have the so-called "segment controller," a sequential process synchronized with respect to the drum interrupt and the sequential processes on higher levels. At level 1 we find the responsibility to cater to the bookkeeping resulting from the automatic backing store. At this level our next abstraction has been achieved; at all higher levels identification of information takes place in terms of segments, the actual storage pages that had lost their identity having disappeared from the picture.

At level 2 we find the "message interpreter" taking care of the allocation of the console keyboard via which conversations between the operator and any of the higher level processes can be carried out. The message interpreter works in close synchronism with the operator. When the operator presses a key, a character is sent to the machine together with an interrupt signal to announce the next keyboard character, whereas the actual printing is done through an output command generated by the machine under control of the message interpreter. (As far as the hardware is concerned the console teleprinter is regarded as two independent peripherals: an input keyboard and an output printer.) If one of the processes opens a conversation, it identifies itself in the opening sentence of the conversation for the benefit of the operator. If, however, the operator opens a conversation, he must identify the process he is addressing, in the opening sentence of the conversation, i.e. this opening sentence must be interpreted before it is known to which of the processes the conversation is addressed! Here lies the logical reason for the introduction of a separate sequential process for the console teleprinter, a reason that is reflected in its name, "message interpreter."
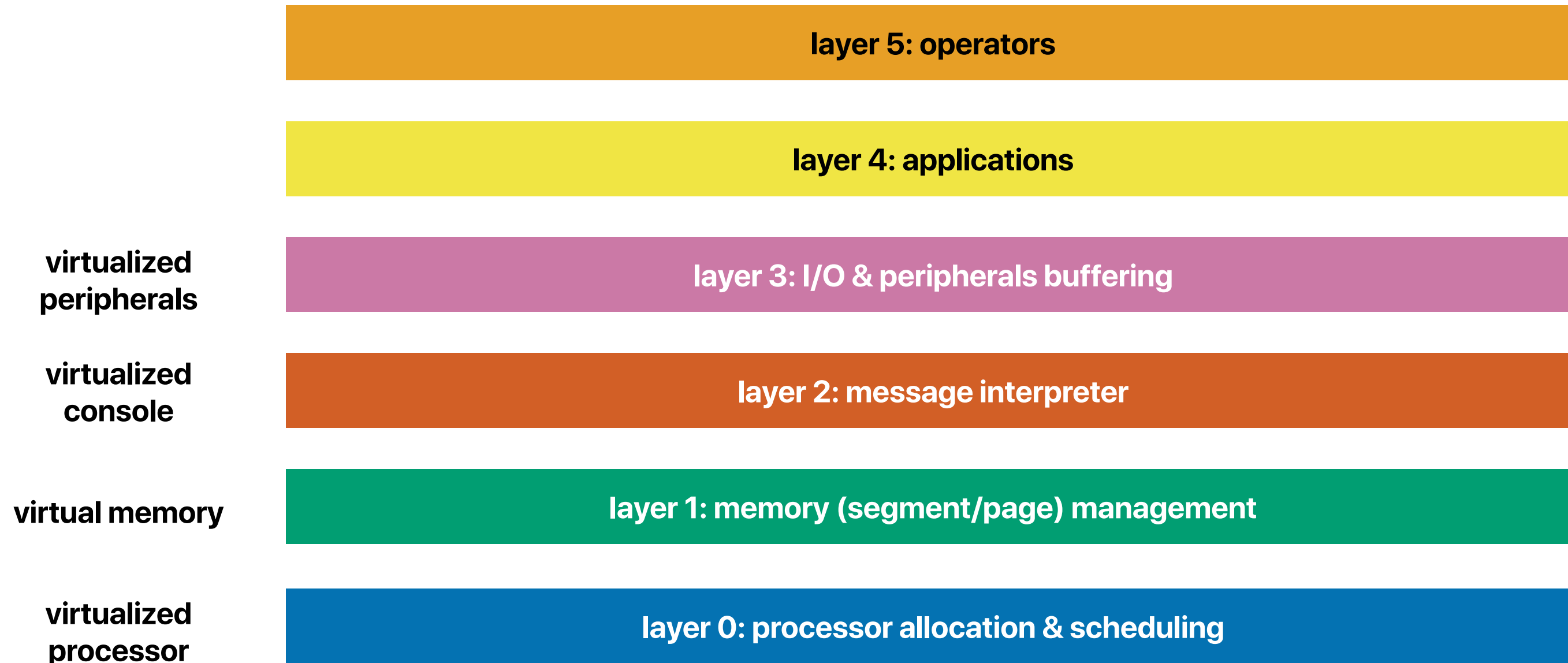
Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form "only one conversation at a time," a restriction that is satisfied via mutual synchronization. At this level the next abstraction has been implemented; at higher levels the actual console teleprinter loses its identity. (If the message interpreter had not been on a higher level than the segment controller, then the only way to implement it would have been to make a permanent reservation in core for it; as the conversational vocabulary might become large (as soon as our operators wish to be addressed in fancy messages), this would result in too heavy a permanent demand upon core storage. Therefore, the vocabulary in which the messages are expressed is stored on segments, i.e. as information units that can reside on the drum as well. For this reason the message interpreter is one level higher than the segment controller.)

At level 3 we find the sequential processes associated with buffering of input streams and unbuffering of output streams. At this level the next abstraction is effected, viz. the abstraction of the actual peripherals used that are allocated at this level to the "logical communication units" in terms of which are worked in the still higher levels. The sequential processes associated with the peripherals are of a level above the message interpreter, because they must be able to converse with the operator (e.g. in the case of detected malfunctioning). The limited number of peripherals again acts as a resource restriction for the processes at higher levels to be satisfied by mutual synchronization between them.

At level 4 we find the independent-user programs and at level 5 the operator (not implemented by us).

The system structure has been described at length in order to make the next section intelligible.

What has been proposed?

	layer 5: operators
	layer 4: applications
virtualized peripherals	**layer 3: I/O & peripherals buffering**
virtualized console	**layer 2: message interpreter**
virtual memory	**layer 1: memory (segment/page) management**
virtualized processor	**layer 0: processor allocation & scheduling**

Each layer has a different privilege mode — your processor needs to provide 5 levels of execution modes

31

Where is how?

performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with unavoidable with the execution of the output commands; furthermore, we have the "segment controller" associated with the drum and the "message interpreter" associated with the console keyboard.

This enabled us to design the whole system in terms of these abstract "sequential processes." Their harmonious cooperation is regulated by means of explicit mutual synchronization statements. On the one hand, this explicit mutual synchronization is necessary, as we do not make any assumption about speed ratios; on the other hand, this mutual synchronization is possible because "delaying the progress of a process temporarily" can never be harmful to the interior logic of the process delayed. The fundamental consequence of this approach—viz. the explicit mutual synchronization—is that the harmonious cooperation of a set of such sequential processes can be established by discrete reasoning; as a further consequence the whole harmonious society of cooperating sequential processes is independent of the actual number of processors available to carry out these processes, provided the processors available can switch from process to process.

System Hierarchy. The total system admits a strict hierarchical structure.

At level 0 we find the responsibility for processor allocation to one of the processes whose dynamic progress is logically permissible (i.e. in view of the explicit mutual synchronization). At this level the interrupt of the real-time clock is processed and introduced to prevent any process to monopolize processing power. At this level a priority rule is incorporated to achieve quick response of the system where this is needed. Our first abstraction has been achieved; above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor that had lost its identity having disappeared from the picture.

At level 1 we have the so-called "segment controller," a sequential process synchronized with respect to the drum interrupt and the sequential processes on higher levels. At level 1 we find the responsibility to cater to the bookkeeping resulting from the automatic backing store. At this level our next abstraction has been achieved; at all higher levels identification of information takes place in terms of segments, the actual storage pages that had lost their identity having disappeared from the picture.

At level 2 we find the "message interpreter" taking care of the allocation of the console keyboard via which conversations between the operator and any of the higher level processes can be carried out. The message interpreter (as far as the hardware is concerned the console teleprinter is regarded as two independent peripherals: an input keyboard and an output printer.) If one of the processes opens a conversation, it identifies itself in the opening sentence of the conversation for the benefit of the operator. If, however, the operator opens a conversation, he must identify the process he is addressing, in the opening sentence of the conversation, i.e. this opening sentence must be interpreted before it is known to which of the processes the conversation is addressed! Here lies the logical reason for the introduction of a separate sequential process for the console teleprinter, a reason that is reflected in its name, "message interpreter."

Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form "only one conversation at a time," a restriction that is satisfied via mutual synchronization. At this level the next abstraction has been implemented; at higher levels the actual console teleprinter loses its identity. (If the message interpreter had not been on a higher level than the segment controller, then the only way to implement it would have been to make a permanent reservation in core for it; as the conversational vocabulary might become large (as soon as our operators wish to be addressed in fancy messages), this would result in too heavy a permanent demand upon core storage. Therefore, the vocabulary in which the messages are expressed is stored on segments, i.e. as information units that can reside on the drum as well. For this reason the message interpreter is one level higher than the segment controller.)

At level 3 we find the sequential processes associated with buffering of input streams and unbuffering of output streams. At this level the next abstraction is effected, viz. the abstraction of the actual peripherals used that are allocated at this level to the "logical communication units" in terms of which are worked in the still higher levels. The sequential processes associated with the peripherals are of a level above the message interpreter, because they must be able to converse with the operator (e.g. in the case of detected malfunctioning). The limited number of peripherals again acts as a resource restriction for the processes at higher levels to be satisfied by mutual synchronization between them.

At level 4 we find the independent-user programs and at level 5 the operator (not implemented by us).

The system structure has been described at length in order to make the next section intelligible.

Synchronizing Primitives

Explicit mutual synchronization of parallel sequential processes is implemented via so-called "semaphores." They are special purpose integer variables allocated in the universe in which the processes are embedded; they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the P-operation and the V-operation.

A process, "Q" say, that performs the operation "P(sem)" decreases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is nonnegative, process Q can continue with the execution of its next statement; if, however, the resulting value is negative, process Q is stopped and booked on a waiting list associated with the semaphore concerned. Until further notice (i.e. a V-operation on this very same semaphore), dynamic progress of process Q is not logically permissible and no processor will be allocated to it (see above "System Hierarchy," at level 0).

A process, "R" say, that performs the operation "V(sem)" increases the value of the semaphore called "sem" by 1. If the resulting value of the semaphore concerned is positive, the V-operation in question has no further effect; if, however, the resulting value of the semaphore concerned is nonpositive, one of the processes booked on its waiting list is removed from this waiting list, i.e. its dynamic progress is again logically permissible and in due time a processor will be allocated to it (again, see above "System Hierarchy," at level 0).

COROLLARY 1. *If a semaphore value is nonpositive its absolute value equals the number of processes booked on its waiting list.*

COROLLARY 2. *The P-operation represents the potential delay, the complementary V-operation represents the removal of a barrier.*

Note 1. P- and V-operations are "indivisible actions";

i.e. if they occur "simultaneously" in parallel processes they are noninterfering in the sense that they can be regarded as being performed one after the other.

Note 2. If the semaphore value resulting from a V-operation is negative, its waiting list originally contained more than one process. It is undefined—i.e. logically immaterial—which of the waiting processes is then removed from the waiting list.

Note 3. A consequence of the mechanisms described above is that a process whose dynamic progress is permissible can only lose this status by actually progressing, i.e. by performance of a P-operation on a semaphore with a value that is initially nonpositive.

During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.

Mutual Exclusion

In the following program we indicate two parallel, cyclic processes (between the brackets "parbegin" and "parend") that come into action after the surrounding universe has been introduced and initialized.

```
begin semaphore mutex; mutex := 1;
  parbegin
    begin L1: P(mutex); critical section 1; V(mutex);
      remainder of cycle 1; go to L1
    end;
    begin L2: P(mutex); critical section 2; V(mutex);
      remainder of cycle 2; go to L2
    end
  parend
end
```

As a result of the P- and V-operations on "mutex" the actions, marked as "critical sections" exclude each other mutually in time; the scheme given allows straightforward extension to more than two parallel processes,

the maximum value of mutex equals 1, the minimum value equals $-(n-1)$ if we have n parallel processes.

Critical sections are used always, and only for the purpose of unambiguous inspection and modification of the state variables (allocated in the surrounding universe) that describe the current state of the system (as far as needed for the regulation of the harmonious cooperation between the various processes).

Private Semaphores

Each sequential process has associated with it a number of private semaphores and no other process will ever perform a P-operation on them. The universe initializes them with the value equal to 0, their maximum value equals 1, and their minimum value equals -1.

Whenever a process reaches a stage where the permission for dynamic progress depends on current values of state variables, it follows the pattern:

P(mutex);
"inspection and modification of state variables including a conditional V(private semaphore)";
V(mutex);
P(private semaphore)

If the inspection learns that the process in question should continue, it performs the operation "V(private semaphore)"—the semaphore value then changes from 0 to 1—otherwise, this V-operation is skipped, leaving to the other processes the obligation to perform this V-operation at a suitable moment. The absence or presence of this obligation is reflected in the final values of the state variables upon leaving the critical section.

Whenever a process reaches a stage where as a result of its progress possibly one (or more) blocked processes should now get permission to continue, it follows the pattern:

P(mutex);
"modification and inspection of state variables including zero or more V-operations on private semaphores of other processes";
V(mutex).

By the introduction of suitable state variables and appropriate programming of the critical sections any strategy assigning peripherals, buffer areas, etc. can be implemented.

The amount of coding and reasoning can be greatly reduced by the observation that in the two complementary critical sections sketched above the same inspection can be performed by the introduction of the notion of "an

unstable situation," such as a free reader and a process needing a reader. Whenever an unstable situation emerges it is removed (including one or more V-operations on private semaphores) in the very same critical section in which it has been created.

Proving the Harmonious Cooperation

The sequential processes in the system can all be regarded as cyclic processes in which a certain neutral point can be marked, the so-called "homing position," in which all processes are when the system is at rest.

When a cyclic process leaves its homing position it accepts a "task"; when the task has been performed and not earlier, the process returns to its homing position. Each cyclic process has a specific task processing power (e.g. the execution of a user program or unbuffering a portion of printer output, etc.).

The harmonious cooperation is mainly proved in roughly three stages.

(1) It is proved that although a process performing a task may in so doing generate a finite number of tasks for other processes, a single initial task cannot give rise to an infinite number of task generations. The proof is simple as processes can only generate tasks for processes at lower levels of the hierarchy so that circularity is excluded. (If a process needing a segment from the drum has generated a task for the segment controller, special precautions have been taken to ensure that the segment asked for remains in core at least until the requesting process has effectively accessed the segment concerned. Without this precaution finite tasks could be forced to generate an infinite number of tasks for the segment controller, and the system could get stuck in an unproductive page flutter.)
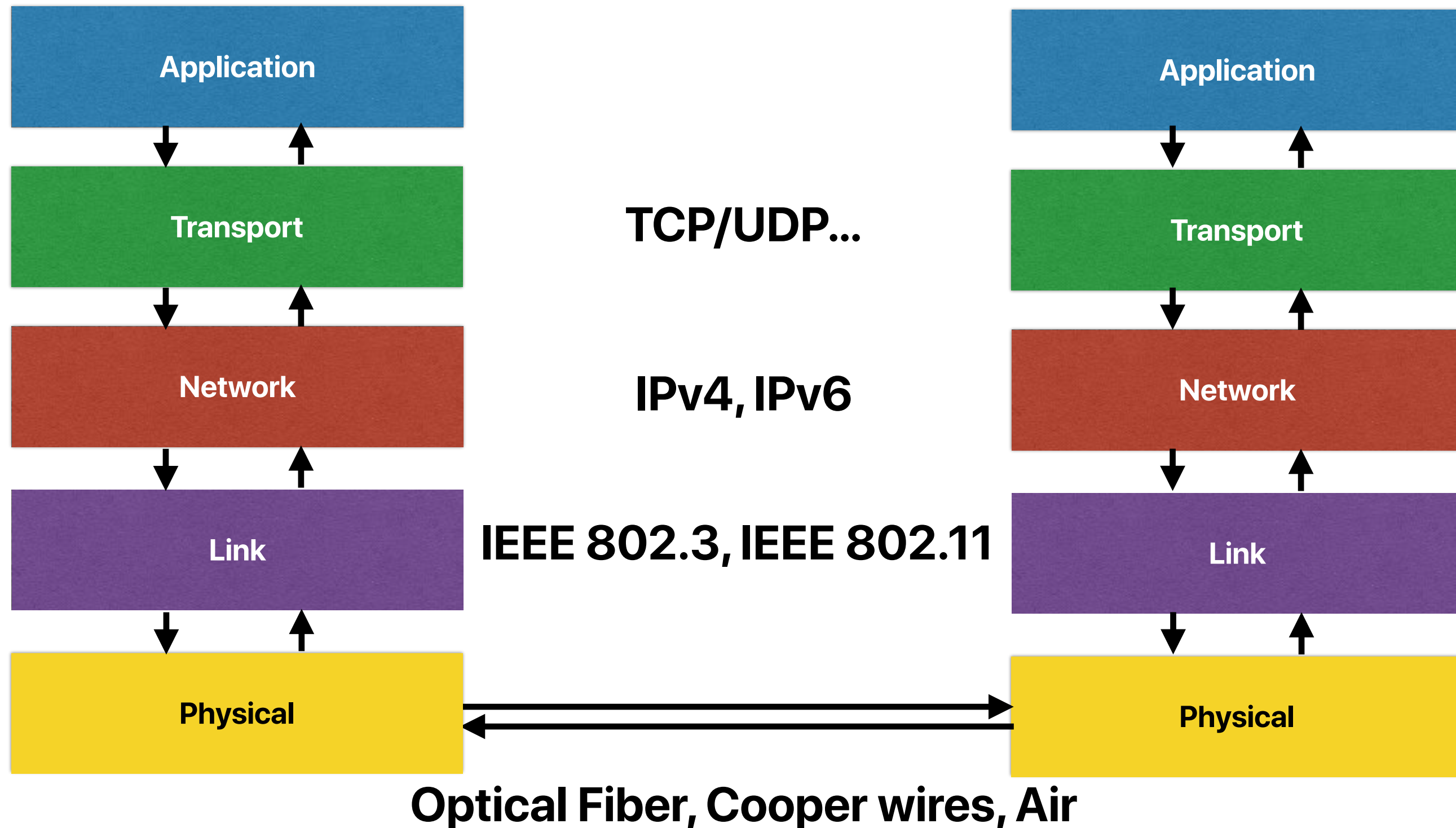
(2) It is proved that it is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task. (This is proved via instability of the situation just described.)

(3) It is proved that after the acceptance of an initial task all processes eventually will be (again) in their homing position. Each process blocked in the course of task execution relies on the other processes for removal of the barrier. Essentially, the proof in question is a demonstration of the absence of "circular waits": process P waiting for process Q waiting for process R waiting for process P. (Our usual term for the circular wait is "the Deadly Embrace.") In a more general society than our system this proof turned out to be a proof by induction (on the level of hierarchy, starting at the lowest level), as A. N. Habermann has shown in his doctoral thesis.

How they achieved these goals?

- Built the layered system to facilitate debugging
- Priority scheduling to improve turn-around time
- Mutual synchronization for sharing resource among processes
  - Processor allocation for processes
  - Access of the physical console among virtual consoles
  - Access peripherals among user programs
  - Keep this in mind, we will discuss mutual exclusion in detail later

Where else do you see hierarchical designs?

Application		Application
Transport	TCP/UDP...	Transport
Network	IPv4, IPv6	Network
Link	IEEE 802.3, IEEE 802.11	Link
Physical		Physical

Optical Fiber, Cooper wires, Air

Impacts of THE

- Process abstraction
- Hierarchical system design
- Virtual memory
- Mutual Synchronization

The Nucleus of a Multiprogramming System

Per Brinch Hansen
A/S Regnecentralen, Copenhagen, Denmark

System Nucleus …

2. System Nucleus

Our basic attitude during the designing was to make no assumptions about the particular strategy needed to optimize a given type of installation, but to concentrate on the fundamental aspects of the control of an environment consisting of parallel, cooperating processes.

Our first task was to assign a precise meaning to the process concept, i.e. to introduce an unambiguous terminology defining what a process is and how it is implemented on the actual computer.

The next step was to select primitives for the synchronization and transfer of information among parallel processes.

Our final decisions concerned the rules for the dynamic creation, control, and removal of processes.

The purpose of the system nucleus is to implement these fundamental concepts: simulation of processes; communication among processes; creation, control, and removal of processes.

rupt system. We do not regard the system nucleus as an independent process, but rather as a software extension of the hardware structure, which makes the computer more attractive for multiprogramming. Its function is to implement our process concept and primitives that processes can invoke to create and control other processes and communicate with them.
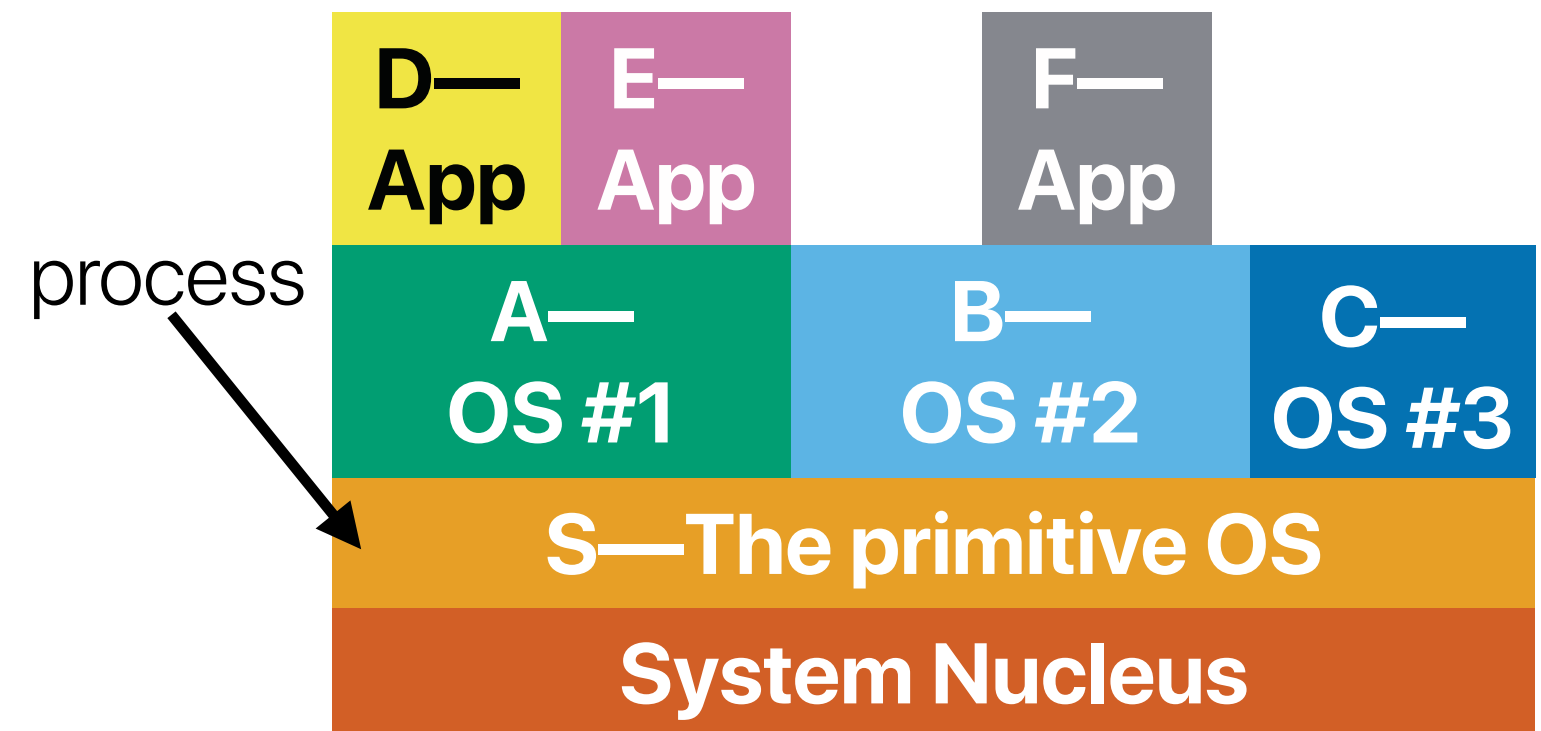
process

S

System Nucleus
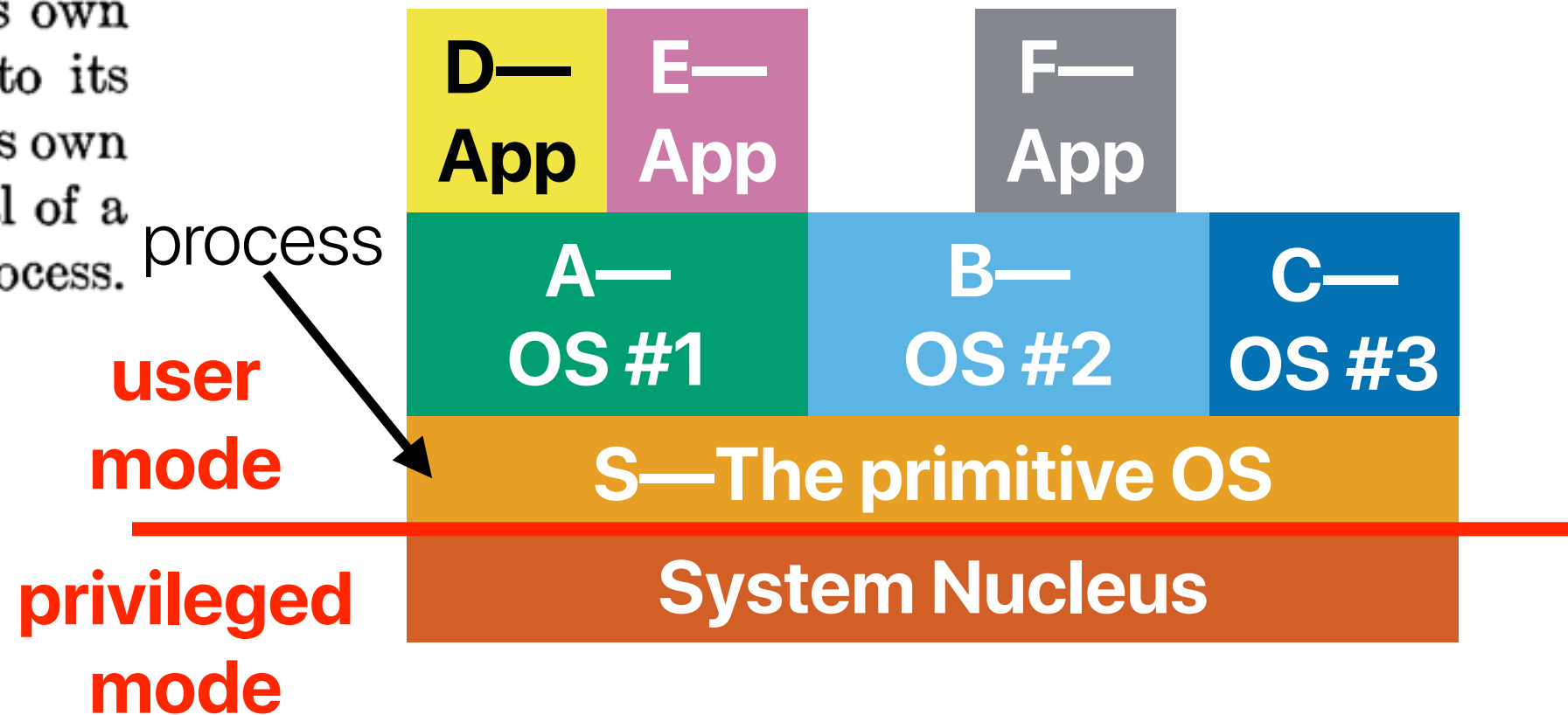
47
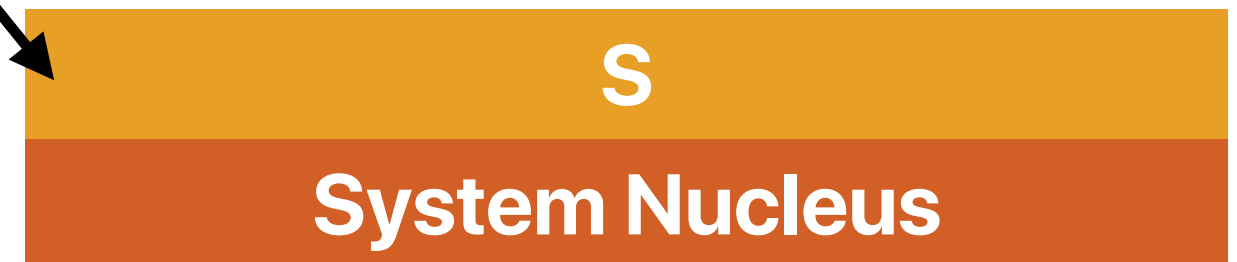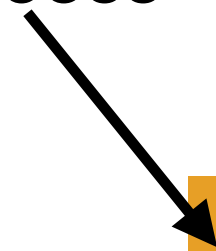
Can multiple OSs running concurrently?

For a given installation we still need, as part of the system, programs that control strategies of operator communication, program scheduling, and resource allocation; but it is essential for the orderly growth of the system that these *operating systems* be implemented as other programs.

After initial loading, the internal store contains the system nucleus and a basic operating system, S, which can create parallel processes, A, B, C, etc., on request from consoles. The processes can in turn create other processes, D, E, F, etc. Thus while S acts as a primitive operating system for A, B, and C, these in turn act as operating systems for their children, D, E, and F. This is illustrated by

process →

D—App	E—App		F—App		
A— OS #1		B— OS #2		C— OS #3	
S—The primitive OS					
System Nucleus					

49

How many layers?

In this multiprogramming system, all privileged functions are implemented in the system nucleus, which has no built-in strategy. Strategies can be introduced at the various higher levels, where each process has the power to control the scheduling and resource allocation of its children. The only rules enforced by the nucleus are the following: a process can only allocate a subset of its own resources (including storage and message buffers) to its children; a process can only start, stop, and remove its own children (including their descendants). After removal of a process, its resources are returned to the parent process.

process

user mode

privileged mode

D— App

E— App

F— App

A— OS #1

B— OS #2

C— OS #3

S—The primitive OS

System Nucleus

50

Hierarchical design v.s. flat structure

- Hierarchical
  - Ease of debugging/verification/testing
  - Lack of flexibility — you can only interact with neighbor layers
  - Overhead in each layer — not so great for performance
- Flat
  - Flexibility/Freedom

Announcement

- Reading quizzes due this Thursday
- We will make both midterm and final exams online this quarter
  - Avoid the uncertainty of COVID-19
  - Avoid high-density in the classroom (only sits 60 and we have 59 for now) during examines