Design philosophy of operating systems (I)

Hung-Wei Tseng

Outline

- How to read research papers
- The Structure of the 'THE'-Multiprogramming System
- HYDRA: The Kernel of a Multiprocessor Operating System

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

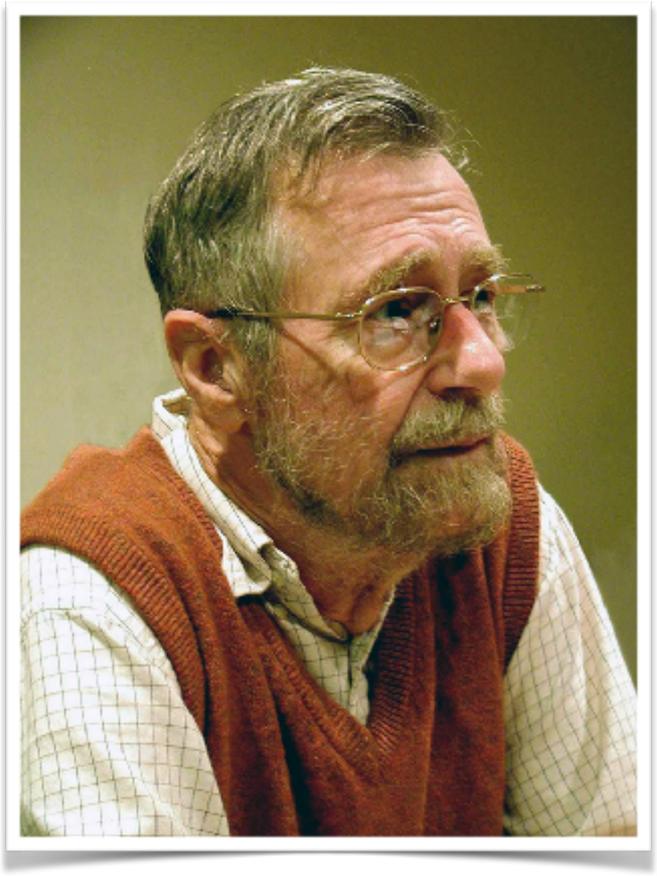
- As a researcher
  - You want to identify important problems
  - You want to know what has been accomplished
- As an engineer
  - You want to know if there is a solution of the design problems of your systems, applications
  - You want to know if you can apply the proposed mechanism
  - You want to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

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1–4, 1967.

Volume 11 / Number 5 / May, 1968

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

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

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

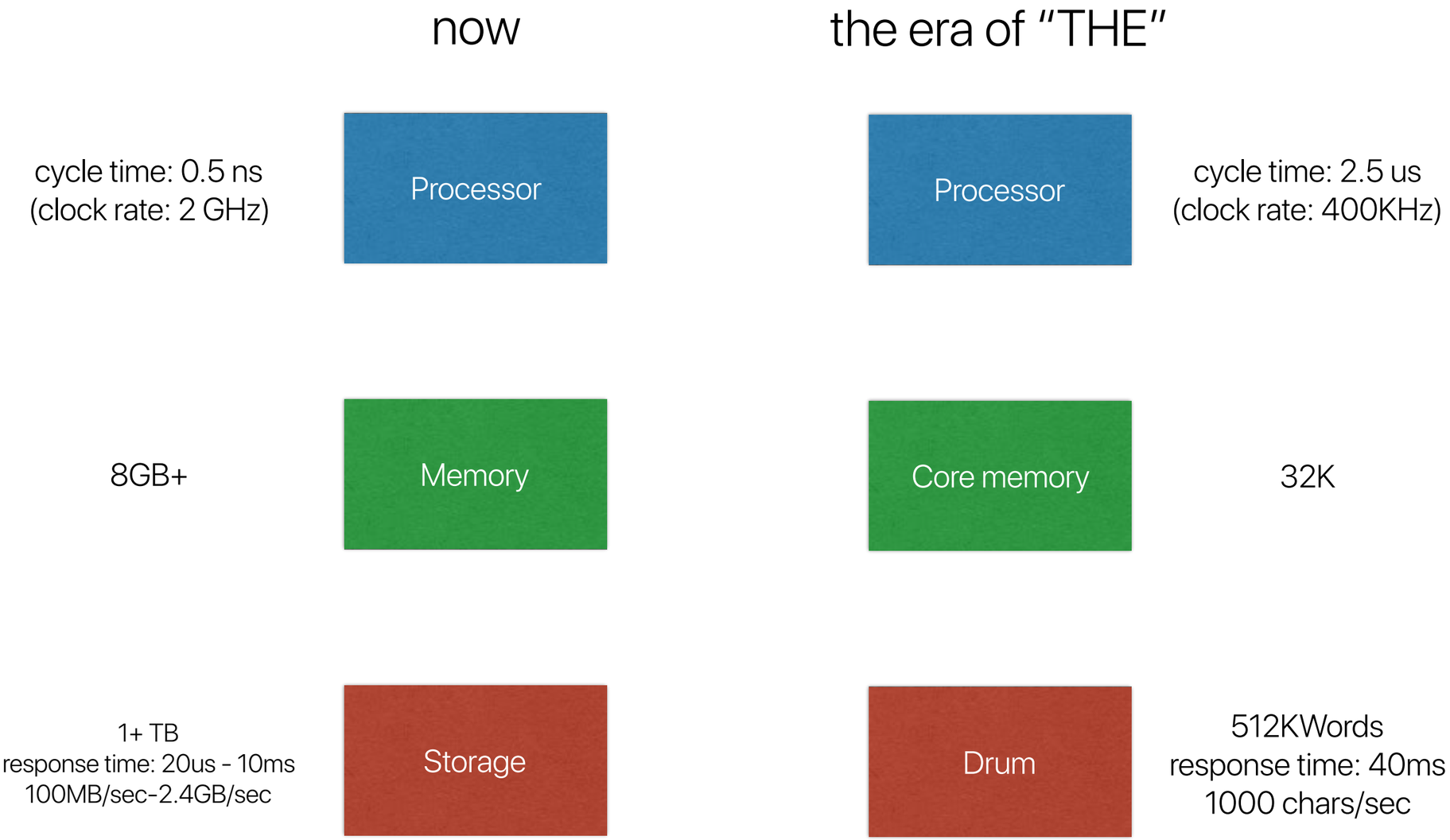
Scheduling Metrics

- CPU utilization — how busy we keep the CPU to be
- 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”



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 synchronisation 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, 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."

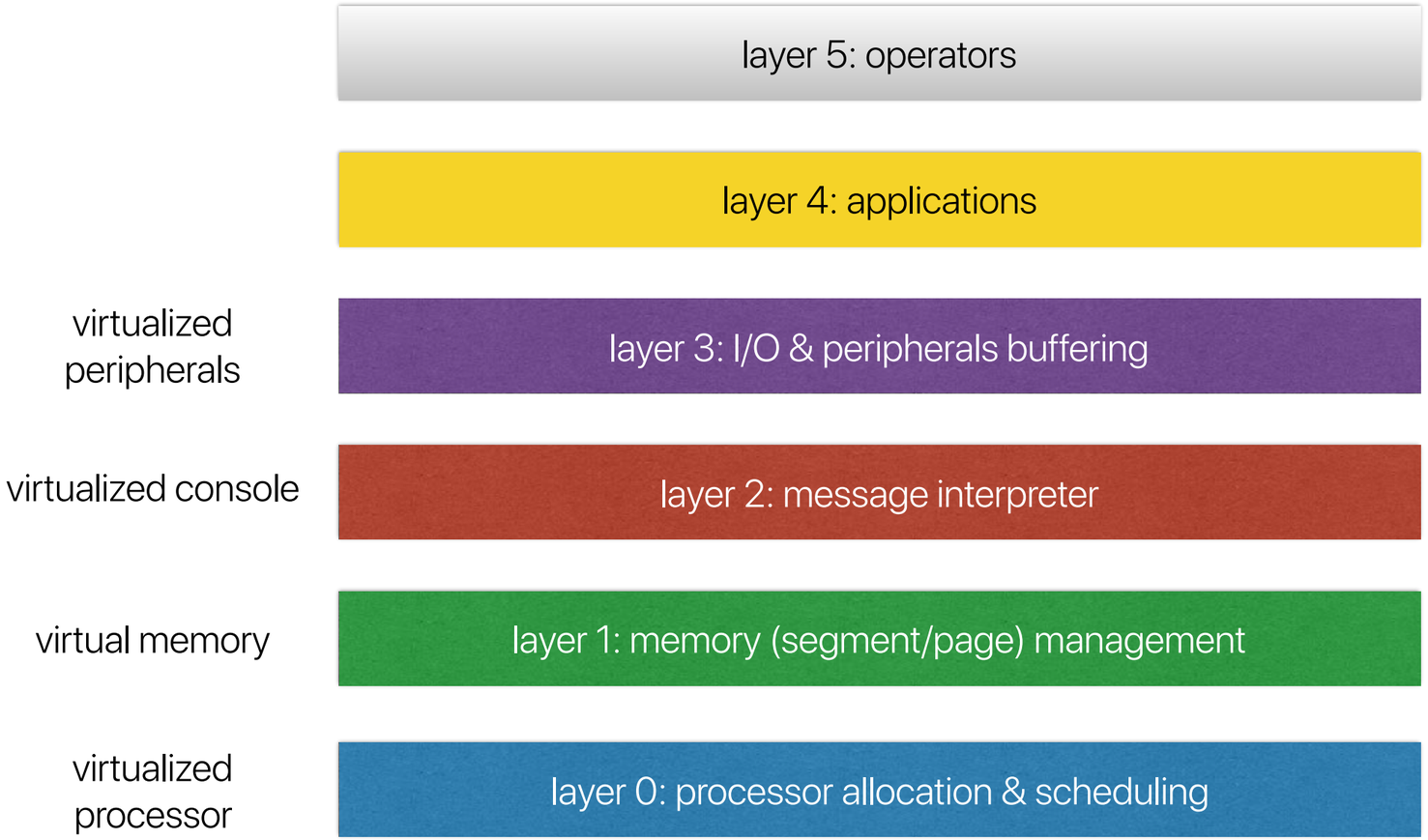
Above level 2 it is as if each process had its private conversational console. The fact that they share the same physical console is translated into a resource restriction of the form "only one conversation at a time," a restriction that is satisfied via mutual synchronization. At this level the next 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).

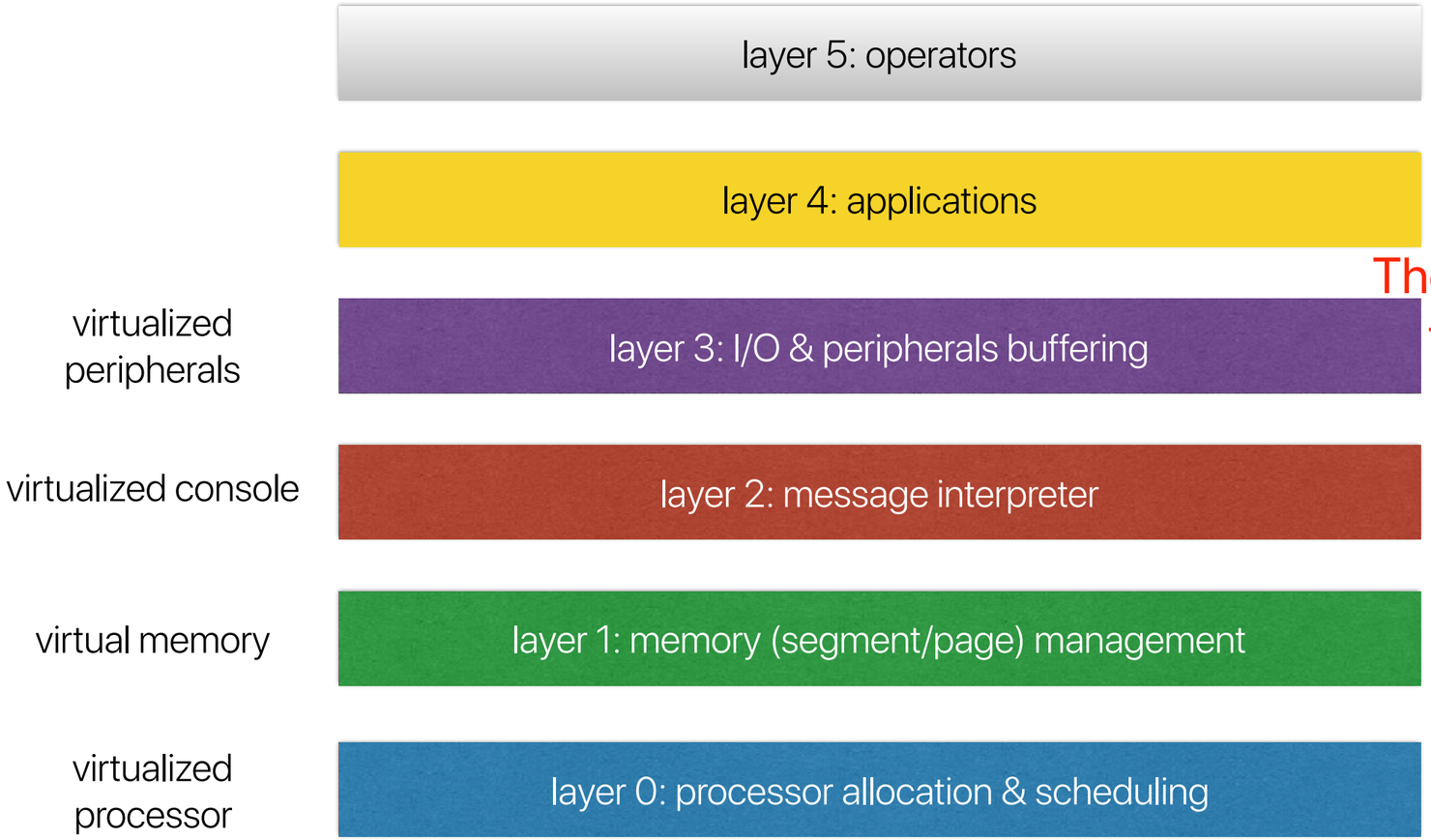
The system structure has been described at length in order to make the next section intelligible.

What has been proposed?



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

Potential problems?



Careful layout of levels:
The peripherals always need to go through message interpreter.
Why?

What if the program of processor allocation/scheduling needs more memory?

Where is how?

performed. Therefore we have arranged the whole system as a society of sequential processes, progressing with unversations between the operator and any of the higher level processes can be carried out. The message interpreter

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 eachhardware is converted 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.

Design Experience

The conception stage took a long time. During that period of time the concepts have been born in terms of which we sketched the system in the previous section. Furthermore, we created the sort of reasoning by which we could decide from our own invention the way in which the processes would influence each other by their mutual synchronization so that those requirements would be met. (The requirements being that no information can be used before it has been produced, that no peripheral can be sent to two tasks simultaneously, etc.). Finally we learned the art of reasoning by which we could prove that the society composed of processes thus mutually synchronized by each other would indeed in its time behavior satisfy all requirements.

The coding stage has been rather dull, though perhaps even old fashioned, that is, plain routine work. Reprogramming on account of a change of specifications has been rare, a circumstance that must have contributed greatly to the feasibility of the "clean method." That the first few stages took more time than planned was somewhat compensated by a delay in the delivery of the machine.

In the verification stage we had the machine, during short shots, completely at our disposal. Once were shots during which we worked with a virgin machine without any software aids for debugging. Starting at level 0 the system was tested, each time adding (a portion of) the next level only after the previous level had been thoroughly tested. Each test also itself contained, on top of the (partial) system to be tested, a number of testing processes with a double function. First, they had to force the system into all different relevant states, second, they had to verify that the system continued to meet according to specification.

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.

This fact was one of the happy consequences of the hierarchical structure.

Testing level 0 (the real-time clock and processor allocation) implied a number of testing sequential processes on top of it, inspecting together that under all circumstances processor time was divided among them according to the rules. This being established, sequential processes as such were implemented.

Testing the segment controller at level 1 meant, that all "relevant states" could be formulated in terms of sequential processes making (in various combinations) demands on core pages, situations that could be provoked by explicit synchronization among the testing programs. At this stage the existence of the real-time clock—although interrupting all the time—was so immaterial that one of the testers indeed forgot its existence.

By that time we had implemented the mutual synchronization upon the (mutually unsynchronized) interrupts from the real-time clock and the drum. If we had not introduced the separate levels 0 and 1, and if we had not created a terminology (viz. that of the rather abstract sequential processes) in which the existence of the clock interrupt could be discarded, but had instead tried in a traditional construction, to make the central processor react directly upon any weird time succession of these two interrupts, the number of "relevant states" would have exploded to such a height that exhaustive testing would have been an illusion. (Apart from that it is doubtful whether we would have had the means to generate them all, drum and clock speed being outside our control.)

For the sake of completeness I must mention a further happy consequence. As stated before, these level 1, core and drum pages have lost their identity, and buffering of input and output streams (at level 3) therefore occurs in terms of segments. While testing at level 2 or 3 the drum channel hardware broke down for some time, but testing proceeded by restricting the number of segments to the number that could be held in core. If building up the teleprinter output streams had been implemented as "dumping onto the drum" and the actual printing as "printing from the drum," this advantage would have been denied to us.

Conclusion

As far as program verification is concerned I present nothing essentially new. In testing a general purpose object (be it a piece of hardware, a program, a machine, or a system), one cannot subject it to all possible cases; for a computer this would imply that one feeds it with all possible programs! Therefore one must test it with a set of relevant test cases. What is, or is not, relevant cannot be decided as long as one regards the mechanism as a black box; in other words, the decision has to be based upon the internal structure of the mechanism to be tested. It seems to be the designer's responsibility to construct his mechanism in such a way—i.e. so effectively structured—that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so comprehensive that he will not have overlooked any situation. I have presented a survey of our system because I think it can be an example of the form that such a structure might take.

In my experience, I am sorry to say, industrial software makers tend to react to the system with mixed feelings. On the one hand, they are inclined to think that we have done a kind of model job, on the other hand, they express doubts whether the techniques used are applicable outside the sheltered atmosphere of a University and express the opinion that we were successful only because of the modest scope of the whole project. It is not my intention to underestimate the organizing ability needed to handle a much bigger job, with a lot more people, but I should like to con-

APPENDIX

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 minimum 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, i.e. 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 fluttor.)

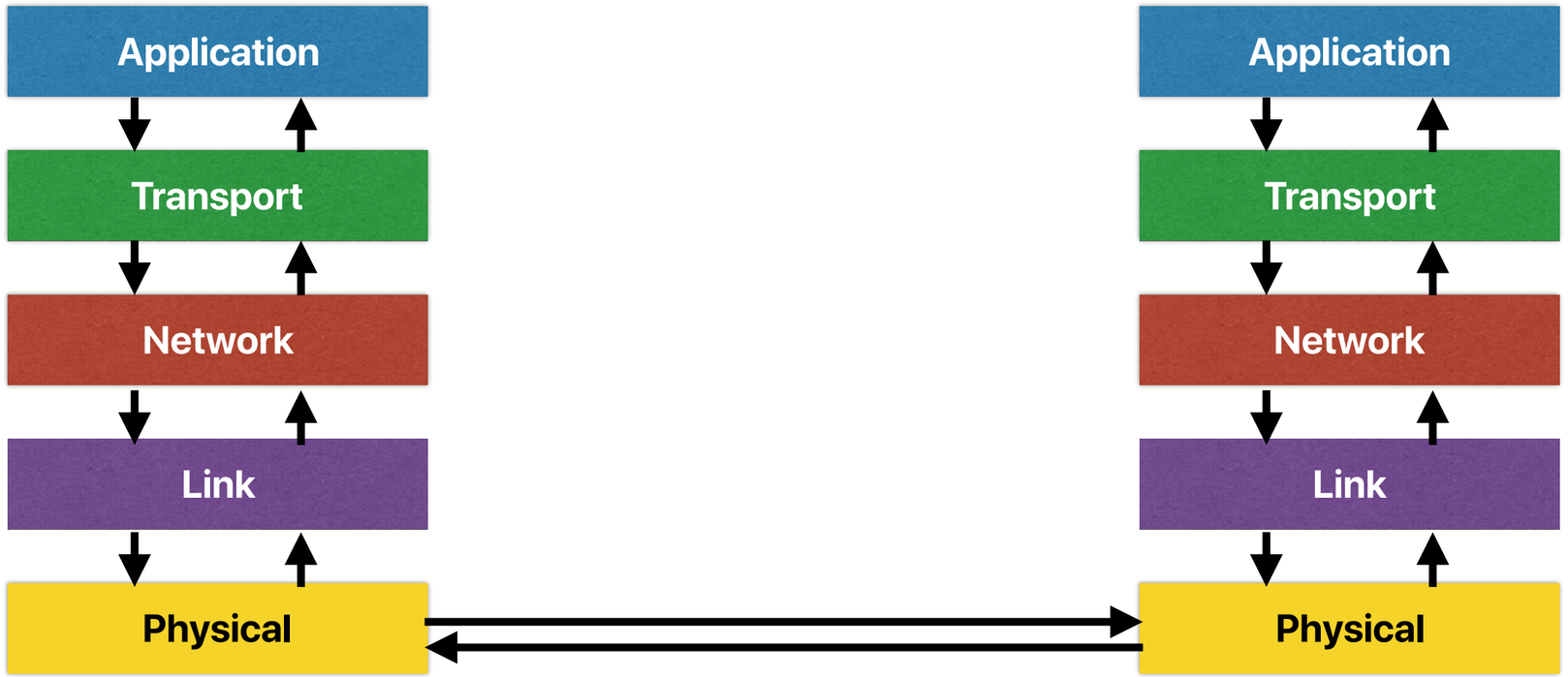
(2) It is proved that it is impossible that all processes have returned to their homing position while somewhere in the system there is still pending a generated but unaccepted task. (This is proved via instability of the situation just described.)

(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
- Implemented 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?



Impacts of THE

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

HYDRA: The Kernel of a Multiprocessor Operating System

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack.
Carnegie-Mellon University

Let’s talk about HYDRA’s whats first

Where is the “what”?

Design Philosophy

The design philosophy of HYDRA evolved from both the environment in which the system was to function and a set of principles held by its designers. The central goals of the system together with the attitudes expressed below suggest that, at the heart of the system, one should build a collection of facilities of "universal applicability" and "absolute reliability" a set of mechanisms from which an arbitrary set of operating system facilities and policies can be conveniently, flexibly, efficiently, and reliably constructed. Moreover, lest the flexibility be constrained at any instant, it should be possible for an arbitrary number of systems created from these facilities to co-exist simultaneously. The collection of such basic facilities has been called the *kernel* or *nucleus* [1] of an operating system. The more specific considerations are listed below.

1. *Multiprocessor environment.* Although multiprocessors have been discussed for well over a decade and a

[1] Manufactured by Digital Equipment Corporation.

few have been built, both the potentials and problems of these systems are dimly perceived. The design of HYDRA was constrained to be sufficiently conservative to insure its construction and utility in a reasonable time frame, yet flexible enough to permit experimental exploration within the design space bounded by its hardware capabilities.

2. *Separation of mechanism and policy.* Among the major causes of our inability to experiment with, and adapt, existing operating systems is their failure to properly separate mechanisms from policies. (Hansen [1] has presented cogent arguments for this separation.) Such separation contributes to the flexibility of the system, for it leaves the complex decisions in the hands of the person who should make them—the higher-level system designer.

3. *Integration of the design with implementation methodology.* It has been observed that the structure of extant operating systems bears a remarkable resemblance to that of the organization which created them. This observation is one of a set which asserts the (practical) impossibility of separating the design from the methodology to be used in implementing the design. The authors' predisposition for implementation methodology is a hybrid of structured programming as advocated by Dijkstra and others [2] and the modularization philosophy of Parnas [8].

4. *Rejection of strict hierarchical layering.* The notion of a strict hierarchically layered system has become popular since first described by Dijkstra for the THE system [3]. While we believe that the system as viewed by any single user should be hierarchically structured, we reject the notion as a global design criterion. We believe that if the entire system is so structured, the design will severely limit the flexibility available to the high-level user and will strangle experimentation. In particular, there is no reason to believe that the same hierarchical relation should exist for control as for resource allocation, or as for protection, etc.

5. *Protection.* Flexibility and protection are closely related, but not inversely proportional. We believe that protection is not merely a restrictive device imposed by "the system" to insure the integrity of user operations, but is a key tool in the proper design of operating systems. It is essential for protection to exist in a uniform manner through the system, and not to be applied only to specific entities (e.g. files). The idea *of* capabilities (in the sense of Dennis [5]) is most important in the HYDRA design; the kernel provides a protection facility for all entities in the system. This protection includes not only the traditional read, write, execute capabilities, but arbitrary protection conditions whose meaning is determined by higher-level software.

6. *Reliability.* The existence of multiple copies of most critical hardware resources in C.mmp suggests the possibility of highly reliable operation. Our desire is to provide commensurate reliability in the software. Reliability not only requires that the system be correct,

but that it be able to detect and recover from errors that do exist—as the result of hardware malfunction, for example.

Defining a kernel with all the attributes given above is difficult, and perhaps impractical at the current state of the art. It is, nevertheless, the approach taken in the HYDRA system. Although we make no claim either that the set of facilities provided by the HYDRA kernel is minimal (the most primitive "adequate" set) or that it is maximally desirable, we do believe the set provides primitives which are both necessary and adequate for the construction of a large and interesting class of operating environments. It is our view that the set of functions provided by HYDRA will enable the user of C.mmp to create his own operating environment without being confined to predetermined command and file systems, execution scenarios, resource allocation policies, etc.

Given the general decision to adopt the "kernel system" approach, the question remains as to what belongs in a kernel, and, perhaps more important, what does not. Nonspecific answers to this question are implicit in the attitudes enumerated earlier; e.g. a set of mechanisms may be appropriate in a kernel, but policy decisions certainly are not. For other, more specific, answers we must step outside these attitudes alone and consider the nature of the entity to be built using the facilities of a kernel.

If a kernel is to provide facilities for building an operating system and we wish to know what these facilities should be, then it is relevant to ask what an operating system *is* or *does*. Two views are commonly held: (1) an operating system defines an "abstract machine" by providing facilities, or resources, which are more convenient than those provided by the "bare" hardware; and (2) an operating system allocates (hardware) resources in such a way as to most effectively utilize them. Of course these views are, respectively, the bird's-eye and worm's eye views of what is a single entity with multiple goals. Nevertheless, the important observation for our purposes is the emphasis placed, in both views, on the central role of *resources*—both physical and abstract.

The mechanisms provided by the HYDRA kernel are all intended to support the abstracted notion of a resource (incarnations of a resource are called *objects*). These mechanisms provide for the creation and representation of new *types* of resources, as well as operations defined on them, protected access to instances of one or more resources within controlled execution domains, and controlled passing of both control and resources between execution domains. The key aspects of these facilities are the generalized notion of resource, the definition of an execution domain, and the protection mechanism which allows or prevents access to resources within a domain. The remainder of this paper focuses on these issues, thus deemphasizing several of the other issues raised earlier.

What HYDRA proposed

- Supporting multiple processors
- Separation of mechanism and policy
- Integration of the design with implementation methodology
- **Rejection of strict hierarchical layering**
- Protection
- Reliability

HYDRA

- Why should we care about HYDRA?
  - Hardware efficiency/utilization
  - Facilitate construction of an environment for **flexible** & **secure** operating systems

Operating Systems
C. Weissman Editor

HYDRA: The Kernel of a Multiprocessor Operating System

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack
Carnegie-Mellon University

This paper describes the design philosophy of HYDRA —the kernel of an operating system for C.mmp, the Carnegie-Mellon Multi-Mini-Processor. This philosophy is realized through the introduction of a generalized notion of "resource," both physical and virtual, called an "object." Mechanisms are presented for dealing with objects, including the creation of new types, specification of new operations applicable to a given type, sharing, and protection of any reference to a given object against improper application of any of the operations defined

Introduction

The HYDRA system is the "kernel" base for a collection of operating systems designed to exploit and explore the potential inherent in a multiprocessor computer system. Since the field of parallel processing in general, and multiprocessing in particular, is not current art, the design of HYDRA has a dual goal imposed upon it: (1) to provide, as any operating system must, an environment for effective utilization of the hardware resources, and (2) to facilitate the construction of such environments. In the latter case the goal is to provide a meta-environment which can serve as the host for exploration of the space of user-visible operating environments.

The particular hardware on which HYDRA has been implemented is C.mmp, a multiprocessor constructed at Carnegie-Mellon University. Although the details of the design of C.mmp are not essential to an understanding of the material which follows, the following brief description has been included to help set the context (a more detailed description may be found in [9]). C.mmp permits the connection of 16 processors to 32 million bytes of shared primary memory through a cross-bar switch. The processors are any of the various models of PDP-11 minicomputers. Each processor is actually an independent computer system with a small amount of private memory, secondary memories, I/O devices, etc. Processors may interrupt each other at any of four priority levels; a central clock serves for

"Kernel"

Defining a kernel with all the attributes given above is difficult, and perhaps impractical at the current state of the art. It is, nevertheless, the approach taken in the HYDRA system. Although we make no claim either that the set of facilities provided by the HYDRA kernel is minimal (the most primitive "adequate" set) or that it is maximally desirable, we do believe the set provides primitives which are both necessary and adequate for the construction of a large and interesting class of operating environments. It is our view that the set of functions provided by HYDRA will enable the user of C.mmp to create his own operating environment without being confined to predetermined command and file systems, execution scenarios, resource allocation policies, etc.

If a kernel is to provide facilities for building an operating system and we wish to know what these facilities should be, then it is relevant to ask what an operating system *is* or *does*. Two views are commonly held: (1) an operating system defines an "abstract machine" by providing facilities, or resources, which are more convenient than those provided by the "bare" hardware; and (2) an operating system allocates (hardware) resources in such a way as to most effectively utilize them. Of course these views are, respectively, the bird's-eye and worm's eye views of what is a single entity with multiple goals. Nevertheless, the important observation for our purposes is the emphasis placed, in both views, on the central role of *resources*—both physical and abstract.

THE v.s. Hydra

THE

privilege boundary

layer 3: I/O & peripherals buffering

privilege boundary

layer 2: message interpreter

privilege boundary

layer 1: memory (segment/page) management

privilege boundary

layer 0: processor allocation & scheduling

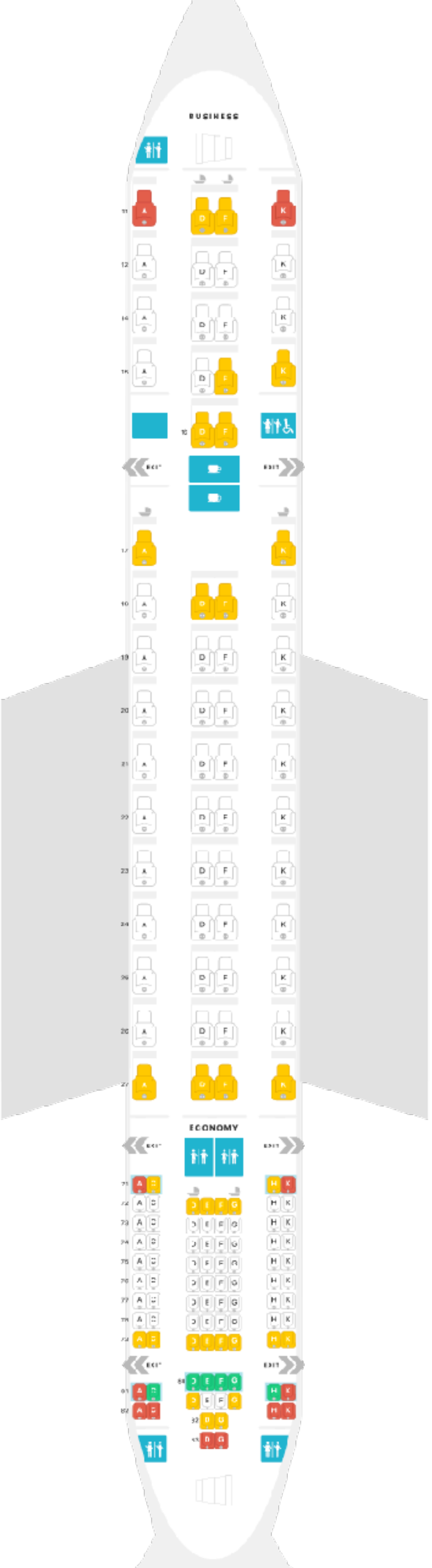
Hydra

privilege boundary

Kernel

What HYDRA proposed

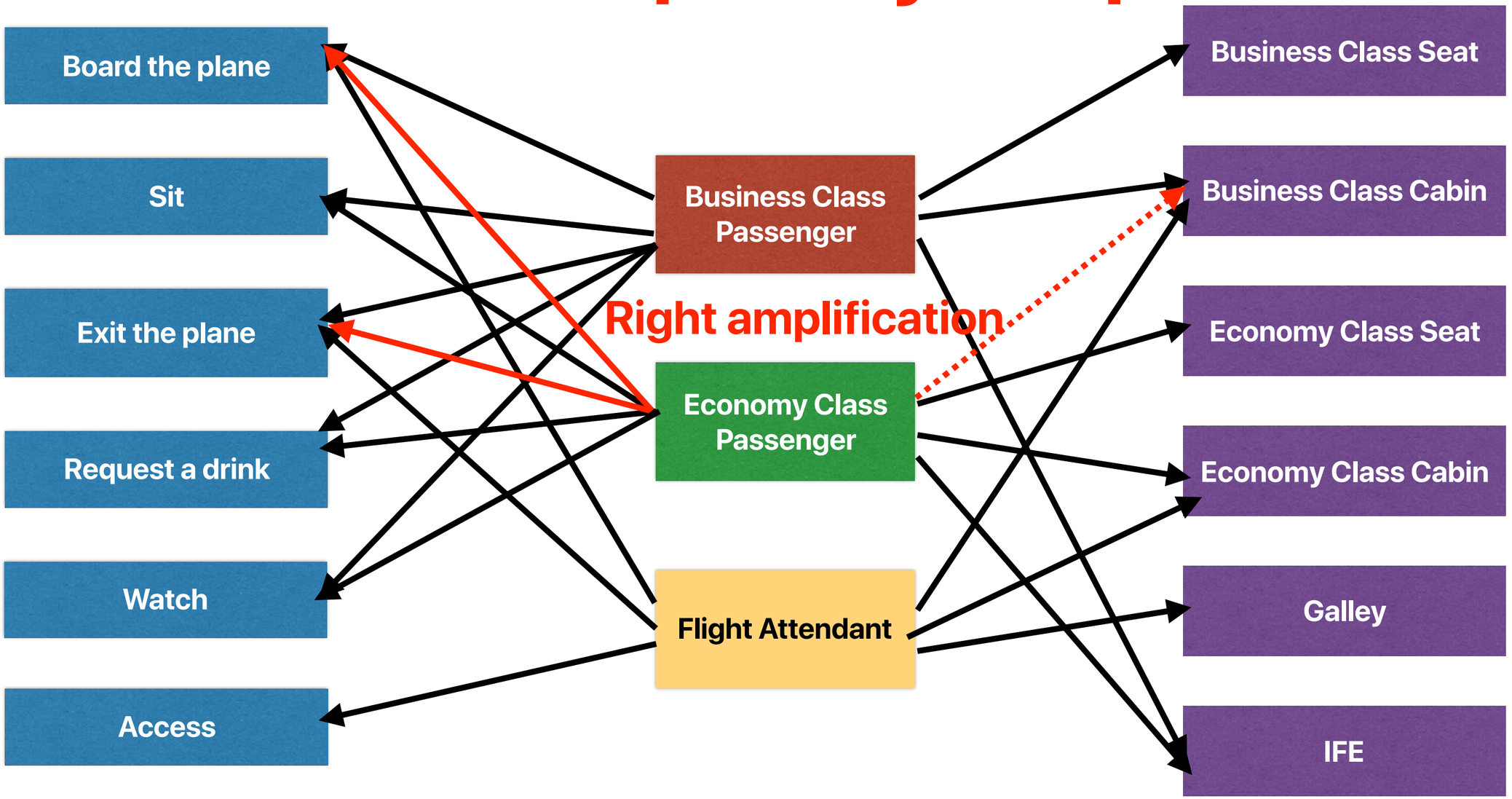
- Supporting multiple processors
- **Separation of mechanism and policy**
- Integration of the design with implementation methodology
- Rejection of strict hierarchical layering
- Protection
- Reliability

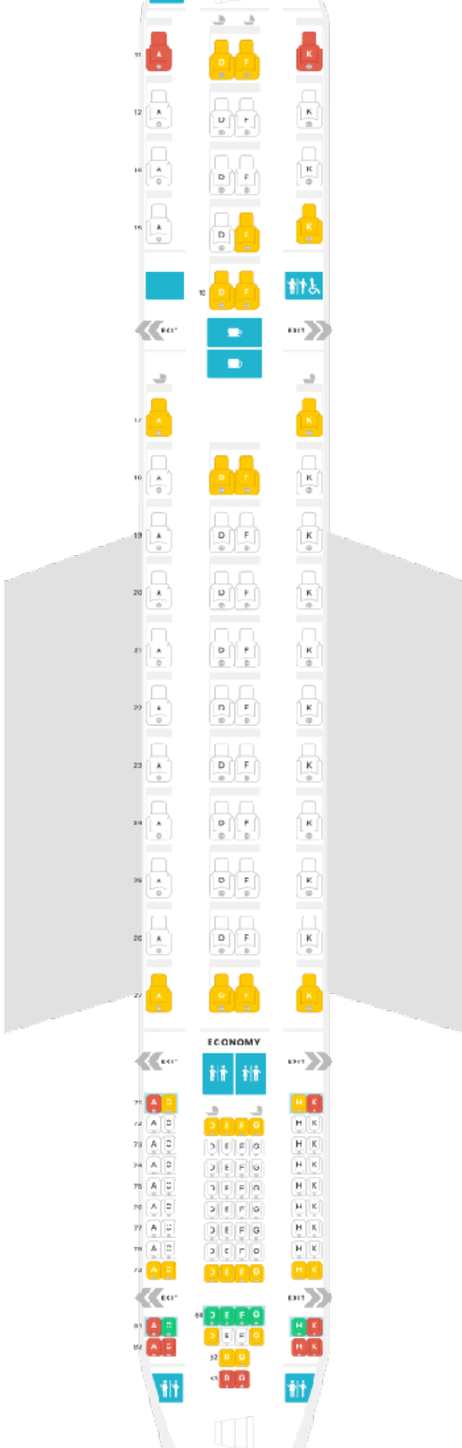

BUSINESS
ECONOMY

Capability v.s. boarding pass

- You can only enjoy the ground services (objects) that your booking class provides
- You can only access the facilities (objects) on the airplane according to the booking class

Capability in a plane





What is capability?

- An access control list associated with an object
  - Thinking about the “protect”, “public”, “private” in Java classes
- Contains the following:
  - A reference to an object
  - A list of access rights
- Whenever an operation is attempted:
  - The requester supplies a capability of referencing the requesting object
  - The OS kernel examines the access rights
    - Type-independant rights
    - Type-dependent rights

Impacts of HYDRA

- Object oriented programming
- A unified abstraction of system resources (objects)
- Protection mechanism — exists in many modern OSes with different implementations
- Flat system design to provide flexibility

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
  - Lower overhead — great for performance
  - Debugging is not easy