THE, Nucleus

Outline

- What os needs to do
  - How to do that
    - VM, scheduling, multi-programming
    - Cpu, memory, disk, i/o devices
  - How to organize the code to do that
    - THE's hierarchical structure
    - Nucleus' micro-kernel structure
- Multiprogramming and synchronization
  - These are the beginning of synchronized multiprogramming
- Reliability, security, performance, programmability

Background

- Multiprogramming
  - Multiple programs running concurrent
  - Comparison with uni-programming
    - Better utilization of resource
    - Shorter turn-around time for short job
  - Process
    - Execution unit
    - Resource unti
- Virtual memory
  - Illusion of memory
  - Comparison w/ non-VM
    - Faster? No
    - Productivity
    - Protection
    - Portability
- Q: multi-programming → VM ?
  - How to protect if no
- Q: VM → multi-programming?
  - Why do we have VM?
- Paging (THE calls page-segment)
  - Divide program space to fixed-size units (unit) for the illusion
  - Q: VM→paging?

Software
OS
hardware
CPU
devices
memory

(Management
Protection
Communication)
(write these later)

The key thing

- THE
  - Layered
  - Upper level calls/invokes lower level
  - User programs are at the highest level
  - \+ Clear design (easy to reason)
  - \+ Easy to test

  Q: how to design the layer?

  *THE's layering is just a development logic; not really `layer'
- Nucleus (microkernel)
  - Put the most essential thing in Nucleus; other part of OS same as user
  - \+ Small and clean Nucleus
  - \+ Easy to extend
  - \+ reliable (isolation)
  - – Performance

  Q: What are the essential things to include in kernel?

The only two elegant and practical designs [1970's]

Other important things

- THE
  - Semaphore synchronization
  - Support for reliability (verification/testing)
- Nucleus
  - Message passing synchronization
  - Support for reliability

Interesting details

- THE
  - Multi-programmed batch system w/ fixed-static jobs
  - How is memory managed
    - S/w VM
  - How is CPU scheduled
  - `Future' impact
- Nucleus
  - Multi-programmed; dynamic-created processes
  - How is memory managed
    - ?? (extensible)
    - Special hardware support
  - How is CPU scheduled
  - `Future' impact

Background of THE

- Machine
  - Memory, disk
  - Interrupt handler; support indirect addressing
  - Peripheral devices
- The design purpose of the machine
  - smooth processing of a continuous flow of user programs
  - A correct system (nightmare of interrupt …)

Overview of THE

- From user's perspective
  - Programs written in ALGOL (only in ALGOL)
  - Operators `schedule' the jobs
    - 1 long job plus 3 short jobs together
  - The operator submits jobs through paper tapes
  - Not multi-access, little shared, not even file (no file storage, sort of! Nothing during the execution will be saved on disk)
- From a software system perspective
  - 15 processes
    - 5 user-level processes (1 for each job; usually the 5th is reserved for …
    - 1 process for each device: drum, console, printer, plotter, 3 readers, 3 punches (each of them synchronizes its activity with respect to interrupts from corresponding device)
  - OS routines

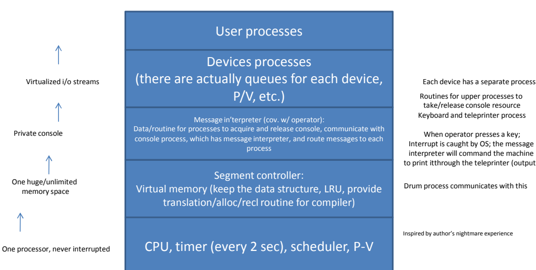
Is THE a good system at that time?

- Yeah
- Performance
  - 20% slower than single machine
  - Short turn-around time (latency) for short jobs
  - Benefit from the multi-programming design choice
- Programmability
  - Big memory
  - Benefit from the VM design choice
- Reliability

THE design choices

- Multi-programming vs. uni-programming
  - Quick turn-around time for short-task
  - Economic use of CPU and devices
- Virtual memory
  - Paging (page -- segment)
  - Advantage
    - Large space
    - No need to be consecutive on disk (vs. `segment')
    - No need to return a memory page to the same slot on disk!

THE's novel design: layering



What does this design mean?

- Implication in terms of some design choice
  - P/V, scheduler cannot use virtual memory
    - They reside in physical memory
  - Message interpreter can use disk (it has large vocabulary)
  - Device drivers could sit in virtual memory
- Performance implication

PL and s/w VM

- ALGOL
  - Was kind of invented by Dijkstra
  - It is an imperative language, solved some problem of FORTRAN, first to use begin/end pair.
  - Inspired/influenced many following languages like C.
- Static VM
  - Paging,
  - 1 table for memory; 1 table for disk; 1 table for each process
  - Compiler instrumented checking/paging routine
  - LRU swapping

Testing and verification in THE

- Testing
  - Modular, layered, clear interface
    - Specifically, …
    - E.g. for scheduler, whether we are switching from user-to-user; or user-to-device does not matter
  - VERY Nice. This makes testing much easier (solve the state explosion issue :P)

  - Something more about testing
- Verification
  - Nobody will generate infinite number of requests (this is NOT trivial!!)
  - There will be no task hanging there without people to pick up
  - There will not be process who accepts a task but never finish

*essentially, they are proving no-hang. Correctness/crash? (not included)!
  - Deadlock issue (banker's algorithm)

Semaphore

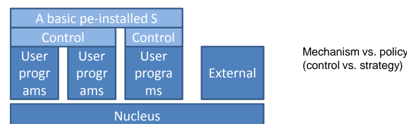
- Details
  - Value associated with the semaphore
  - P and V are each atomic
    - Hardware support in implementation
- Usage
  - Shared semaphore (mutual exclusion)
  - Private semaphore (order)
    - Private means that only one process will consume it
- System call
- Not much used
  - Not solve deadlock
  - Hard to verify
  - Hard to use

Where is this used?
e.g. reserve/release devices
Inside that there is P/V
e.g. console process also keeps
Checking a shared variable: any request?

Nucleus

- Goal
  - Minimize the nucleus
  - Extensible OS

  - More reliable system



What should be included in `kernel'?

- What is included
  - Interrupt handler
    - Initiation of data transfers to or from peripherals
    - It can disable interrupts
  - Provide utility to support process* creation, stop, termination, communication
  - Process schedule*
    - Timer interrupt at 25 milli second; round-robin time slice queue

  - Support in assigning memory and disk
- What is NOT included
  - HOW to schedule among processes
  - VM control and swapping
  - Some device operations

How to finish the job with the `kernel'?

- CPU scheduling
  - Kernel support: round-robin queue
  - User OS: parent-process (de)activate children
- Memory management
  - User OS: parent gives part of its physical memory to children (@ creation time)
  - Kernel: make sure parents do not distribute memory not belongs to them
  - Hardware: protection key
- I/O
  - Kernel: support external process, support the message passing

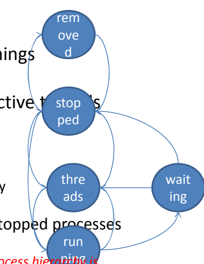
Internal process creation

- Parent creates children (how process hierarchy is built)
  - **Memory region**
  - File access privilege
  - Protection key
- Parent loads kid's image (input, from disk, I believe)
- Parent loads kid's register
- Nucleus support
  - A descriptor for each process (memory range, name, state)
    - THE does not have this because of its `static-process'!!
  - Keep queue for each process
  - Check memory region

User can specify starting process via console

Internal process life time

- Parent can stop child
  - Implement scheduling policy
- Parent can remove child
  - Implement swapping, VM
- Process can use message to wait for things
- Kernel only respond for round-robin active t[illegible]s
- Kernel provides utility routine
  - Check at stop, etc.
    - Maximum is 23 processes;
    - Hardware can protect 8 processes' security
    - `storage' means memory
  - Generate dummy answer on behalf of stopped processes



This is how important process hierarchy is

Memory Management

- Can be extended for a VM system
  - Just stop a process, use output to write its memory to backing storage
  - Bring in its sibling to use the same physical memory region
  - That is it.

Message Passing

- Basic

Basic mechanism:
send (receiver IN, message IN, buffer OUT)

wait (sender OUT, message OUT, buffer OUT)
send_a(result ?, answer IN, buffer IN)

wait_a (result OUT, answer OUT, buffer OUT)

- Advanced

  Wait_event(…)

- Efficiency: 'cause of queue, sender can continue after get the buffer-id, it will be blocked only at `wait'
- Security concern
  - ID
  - Buffer quota
    - A limit on how many messages a process can send simultaneously

Only two iterations, no more!
After wait-a, the buffer is returned to pool;

Because there is queue, the sender does not need to wait.
It can go back to do the `wait' later.

Implementation

- Nucleus keeps buffer pool
- Nucleus keeps the queue for each process
- Process calls utility function
- Nucleus finds free buffer; copy message to buffer, link the buffer to the destination queue

Compare message passing w/ shared memory

- Special design in this paper
  - Wait does not need to know whom it is waiting for
  - Buffer id is used for privacy protection
    - Nucleus sign buffer with sender/recv names
    - Id also used for pairing message with answer
  - There are two chances of information exchange
    - This style was inspired by external process – internal process communication
  - Processes can go away dynamically (nucleus will respond with dummy or …)
  - Problem: buffer is a limited resource
- Comparing using buffer with no-buffer (end-to-end). The advantage of efficiency of asynchronous style communication (enabled by queue)
- Note: they can NOT wait for a particular process
- Compare with semaphore
  - Some comments are semaphore is too low level; not easy to use for high-level implementation; message passing is more `natural' here

How I/O is conducted through message passing

- Internal process, external process
- Example message:
    - operation, 1st mem-address, 2nd mem-addr
    - Status bits, # of bytes, # of characters
- (let's treat internal process and external process as black box at this point)

- Another example:
  - A converter process keeps waiting for
    - Wait
    - Contact printer to print
    - Send answer
    - Back to wait
  - Normal process: get data; send to converter; wait answer

External process and I/O

- External process creation
  - Start by internal process
    - Create_peripheral_process (name, device_number, result)
  - Process handler
- The role of Nucleus at communication between external and internal processes
  - I sends message
  - N puts message onto the queue of E
  - N finds out the handler function of E (based on the type descriptor of the process E, that was specified when I creates E)
  - N executes the handler
  - At interrupt, N replies to I and clean E's queue, or execute another task in E's queue
- External process could be totally implemented as user-process

External vs. Internal processes

- The difference between external process and internal process
  - External process' code is mostly executed by nucleus (with interrupt disabled) in the format of an interrupt handler
  - It definitely CAN be implemented as an internal process, which means after getting message, it will be activated (it was conversational wait). It will be put back to the scheduling queue and wait for its own turn to get executed.
  - Either way is O.K. actually. The external process has less latency, but it might hurt other device/process' response time. It could be replaced as an internal process and it will has large latency but more friendly to others.

- Nucleus actually implements part of its functionality as anonymous internal process when interrupt is allowed

Protection

- Nucleus checking at
  - Process creation, I/O message
  - Note: sibling's memory region CAN overlap

  (this is how virtual memory can be implemented)
- Nucleus makes sure that process cannot manipulate non-children processes
- Hardware support
  - Tagged memory
  - During access, memory tag will be compared with process' protection register

  →at process creation, its whole memory region needs to be re-tagged

  External device actually has a bit vector saying which processes can use me. Initial operating system is included. IntialOS can give this privilege to its children.

  Disk (backing store): there is a catelog, specifying who can access and who cannot, etc.

Extensibility

- How to make it a batching system

- How to make it time-sharing
  - One process for each user
  - Keep swap in and out
  - Need to have buffer for the communication with users when their process is swapped out

impact

- Very reliable
  - NASA
- Message passing
- OS future ??
- Used as Virtual Machine monitor
- Performance issue