Jan 18

1. What is this course about:
   (a) naming things,
   (b) clarifying distinctions.
   (c) making distinctions, so that technical decisions will be informed.
2. What is an operating system:
   (a) presentation of a virtual machine.
   (b) allocation and management of resources.
3. Where does the operating system live:
   (a) In a layer of system software interposed between the hardward layer and the user layer.
   (b) In particular, it is the bottom half of the system layer.
4. Layering and Objects as design principles for operating systems.
5. Historic review of operating systems:
   (a) Batch and Spooling.
   (b) Time Share.
   (c) Multiprocessing and Networking.
6. What is computation:
   (a) process.
   (b) memory.
   (c) communication.
7. Processes - What is a process:
   (a) context.
   (b) a causal flow of events.
8. What is the context of a process? For example,
   (a) the Program Counter (PC).
   (b) the Stack Pointer (SP).
   (c) the Status of a processor, such as:
      i. condition flags,
      ii. interrupt mask,

iii. process permissions.

(d) CPU registers.

9. The context can be saved to suspend a process, and restored to restart it.

10. Interaction of processes:

(a) sequentially performed.

(b) simultaneously performed, perhaps they communicate and interact.

(c) interruption of one process to perform another.

11. Reasons for interruption:

(a) efficiency.

(b) urgency.

(c) preemption.

12. Types of interrupts:

(a) Hardware events

(b) Exceptions, alarms (firmware)

(c) Traps and systems calls (software)

13. How interrupts work:

(a) Interrupt causes a call to an interrupt handler.

(b) The call disables further interrupts and places the context on the stack in a *frame*.

(c) The handler either handles the interrupt directly or enables another task to handle the interrupt.

(d) Restoring the saved context from the stack frame restarts the interrupted process.

JAN 20

14. Time-slicing and pseudo-parallelism.

(a) An interrupt *preempts* the currently running process.

(b) A new stack is choosen. Restoring the context from this stack restarts a previously suspended process.

(c) This is called a *context switch*.

(d) The *Process Table* and the state of a process:

i. Running,

ii. Suspended,

iii. Blocked.

The transition from Running to Suspended is a *preempt*, from Running to Blocked is a *sleep*, from Blocked to Suspended is a *wakeup*.

15. More on interrupts:
    (a) Interrupt requests, disabling interrupts, interrupt acknowledgement and enabling interrupts.
    (b) Multiple interrupts on a single level:
        i. polling
        ii. daisy chaining the acknowledgement
    (c) Multiple level interrupts, hierarchy and interrupt vectors.
16. The *synchronization kernel* is in abstract machine implementing the process model. That is, it has the native machine instruction set plus primitives to map the processes and interrupts onto the hardware, and for their synchronization, also often their intercommunication.
17. Cooperation and competition of processes for resources. The example of sleep and wakeup on the bounded buffer problem.
    (a) Critical Sections.
    (b) Mutual Exclusion.
    (c) Serialization of requests
18. Solutions to mutual exclusion,
    (a) Question of atomicity of actions, governed by hardware
    (b) Turning off interrupts
    (c) Test and Set, Spin Locks, a.k.a. Busy Wait
    (d) Peterson's Algorithm

JAN 25

19. I have come to the realization that there are two models for considering interrupt tasks. Up until now I have embraced the *interrupt model.* This model maps directly to what physically happens. The interrupt software runs nested inside the interrupt process. I do not mean that the interrupt runs on the user's stack, or that context is not completely switched. I mean that the mind considers the interrupt to be during the suspension of some other task. An illustration of the weakness of this model is the inability for it to give definitive answers as to the correct priorities of interrupts, or even to elucidate the issues concerned in assigning priorities.

    The *process model* lets the interrupt process be its own every present thread of computation. The interrupt is separately considered as a kernel trigger to schedule and communicate to this thread. As a scheduling mechanism, it is often used to place the interrupt thread onto the hardware almost immediately. The interrupted task is still running, it has however been deprived of hardware. The

second aspect clarified is the interrupts role as a communication mechanism. It cannot be forgotten that it generates two messages, a stop and a go message, which occur in forced synchronization. To the processes sharing the hardware, the messages are absolutely simultaneous.

We can now unify various aspects of traditional interrupt theory: that the interrupt process runs immediately is a matter of efficiency, so that that signal to start become effective as soon as possible. Also, in the case of exceptions, it is a signal to the excepting process to stop, as least until the handler has a chance to decide what to do with the exception.

The role of what we shall now call the "preemeptor" becomes clearer in the process model. It occasionally marks preemptable processes as suspend or running. A non-preemptable process is exempt, the preempt falls through to the first preemptable process. Hence we are no longer confused whether the clock is a more "important" interrupt than let's say the disk handler. It is because the pseudo-parallel threads of computation will most likely seem to be simultaneous if the signal to preempt reaches it's target as soon as possible, even though its effect in mapping process to cycles will not be felt until later.

This is what happens under Unix. Preemption is considered to occur at the moment of return out of the kernel. This allows the kernel to check the status of a preempt-waiting flag as well as the preemptablity of the return-to process before deciding what process to next give cycles to. In concrete terms, if the clock cannot interrupt the disk handler, it might become necessary to save clock ticks. But if it can, how can the disk handler be saved from preemption? The solution is to mark the disk handler as non-preemptable and pass the preemption signal on to the currently "running" preemptable task, although at the moment the disk handler has put that task in the background. When the disk handler returns to the process it interrupted, it does so by passing through the kernel. The kernel inspects the state of affairs just prior to the return back to the user process, and at that moment effectuates the preemption.

Viewed in the interrupt model, the Unix preemption scheme is unexpected, subtle and clever. The process model "explains" this scheme, and is therefore a more useful model to gauge the scheme's correctness and effectiveness.

There is a third model: the machine model, in which the entire computation is a single process, purely a sequence of machine instructions, without any sort of interpretive layer placed upon it by software designers.

20. Kernel provides mapping of process to hardware:
    (a) preemption
    (b) interruption
    (c) multiple processors
21. Reentrant processes, sharing of code. Typical of interrupt handlers.
    (a) offset addressing
    (b) stack relative addressing
22. Scheduling

(a) round-robin scheduling.
(b) priority scheduling.
(c) preemptable and non-preemptable processes.
(d) Issues
    i. latency (priority)
    ii. fairness (aging)
    iii. resource efficiency

23. Semaphores.
    (a) Dijstra's PV
    (b) As a mutex
    (c) As a counting semaphore
    (d) Implementation in the synchronization kernel
24. Virtual Resouces: the monitor
    (a) Virtual resource
    (b) Allocation
    (c) Serialization
    (d) Blocked enter.
    (e) Signal and wait on conditions
25. Message Passing.
    (a) Passing to a process, message queue.
    (b) Passing to a mailbox.
    (c) Size of mailbox, rendez-vous