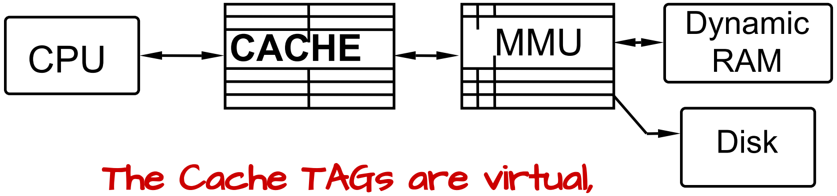
ALMOST OVER

1) Last Problem Set is due Tonight
2) Final Exam on Saturday at 8am
   50 questions - Open book, open notes, open internet
      ~25 on pipelining, pipelining CPUs, caches, virtual memory
      ~25 on earlier course material

USING CACHES WITH VIRTUAL MEMORY

Virtual Cache

Tags match virtual addresses



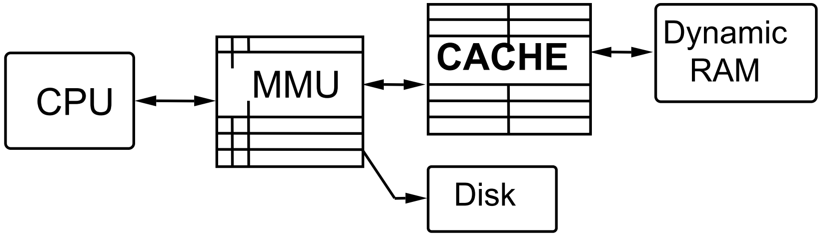
The Cache TAGs are virtual, they represent addresses before translation.

- Problem: cache becomes invalid after context switch
- FAST: No MMU time on HIT

Physical Cache

Tags match physical addresses

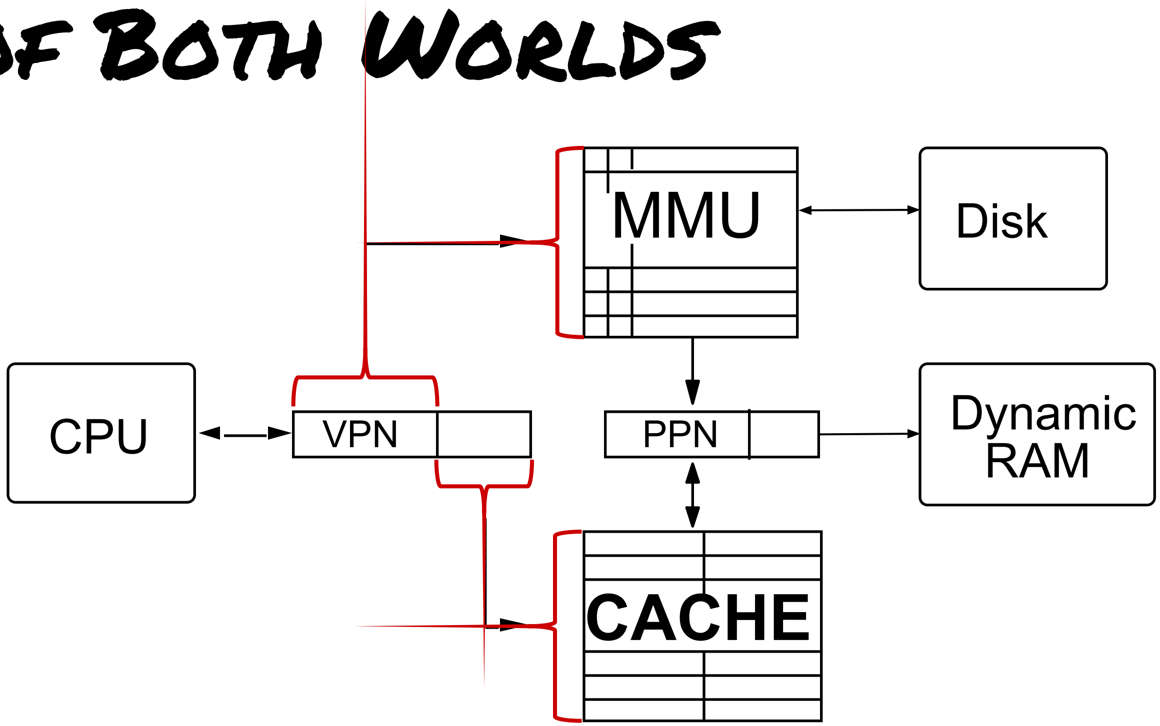
These TAGs are physical, they hold addresses after translation.



- Avoids stale cache data after context switch
- SLOW: MMU time on HIT

Physically addressed Caches are the trend, because they better support parallel processing

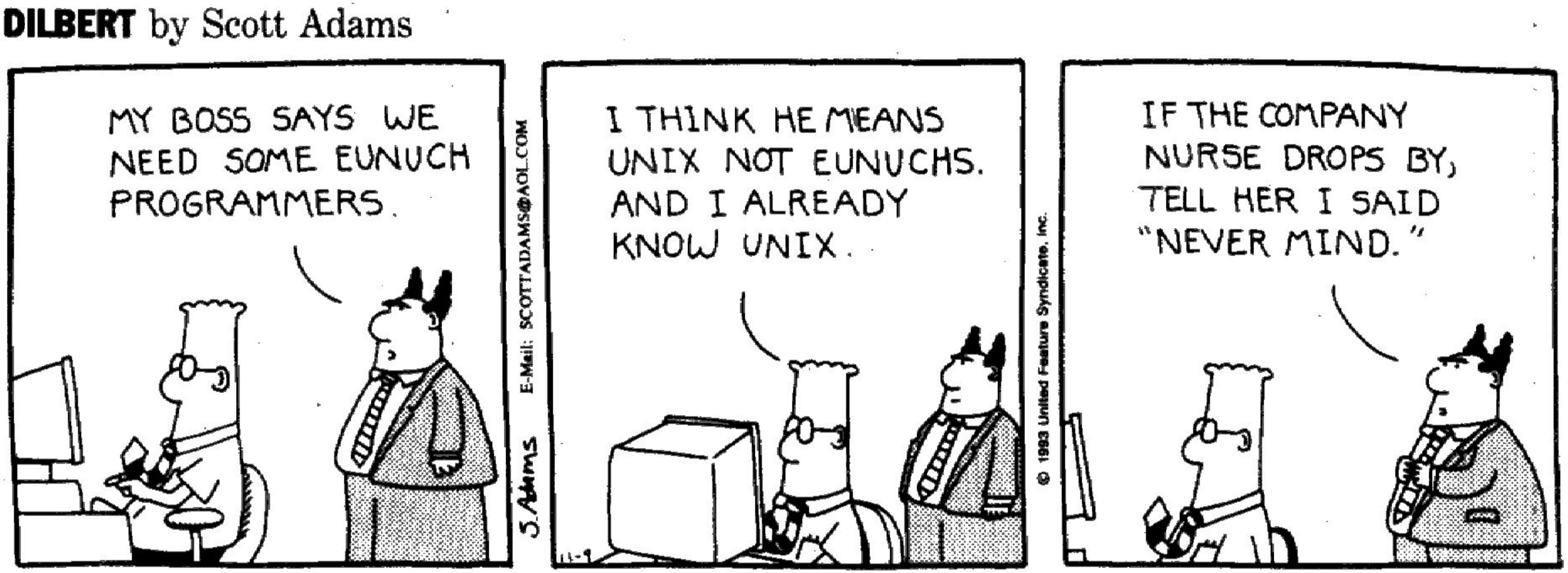
Best of Both Worlds



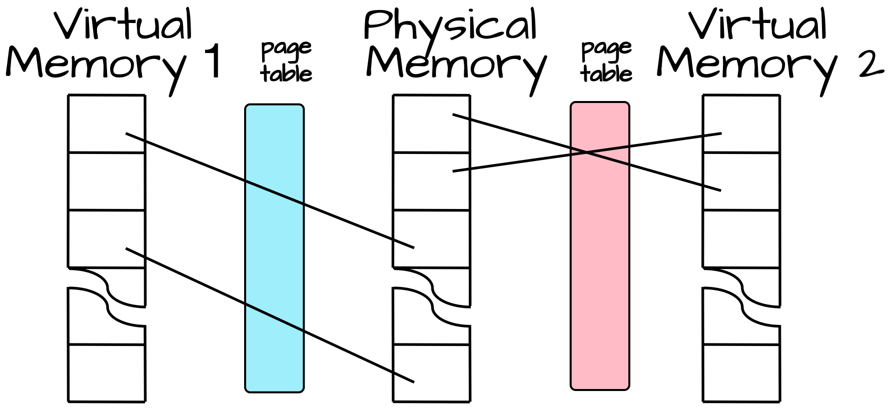
OBSERVATION: If cache line selection is based on *unmapped* page offset bits, RAM access in a physical cache can overlap page map access. Tag from cache is compared with physical page number from MMU.

Want "small" cache index / small page size → go with more associativity

VIRTUAL MACHINES & THE OS KERNEL



POWER OF CONTEXTS: SHARING A CPU



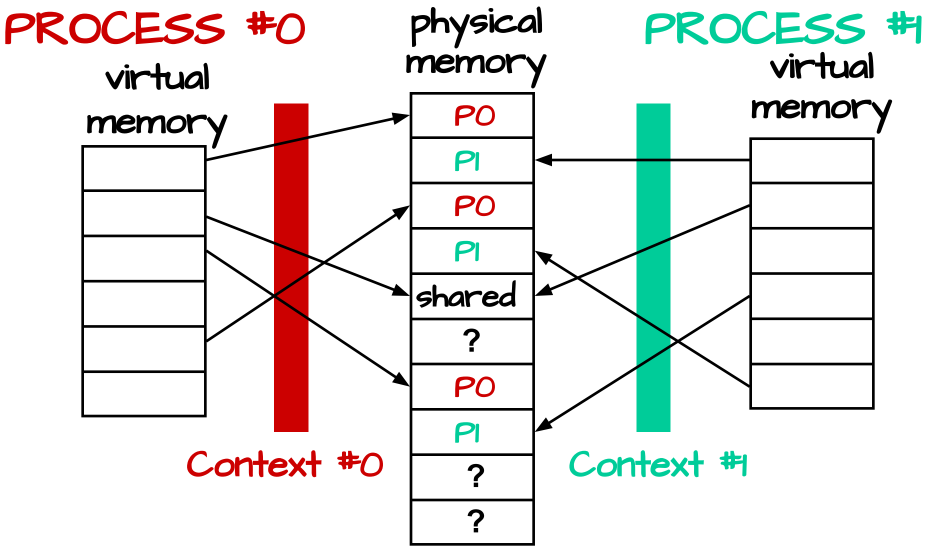
Every application can be written as if it has access to all of memory, without considering where other applications reside.

More than Virtual Memory
A VIRTUAL MACHINE

1. TIMESHARING among several programs --
   - Programs alternate running in time slices called "Quanta"
   - Separate context for each program
   - OS loads appropriate context into pagemap when switching among pgms
2. Separate context for OS "Kernel" (eg, interrupt handlers)...
   - "Kernel" vs "User" contexts
   - Switch to Kernel context on interrupt;
   - Switch back on interrupt return.

What is this OS KERNEL thingy?

BUILDING A VIRTUAL MACHINE

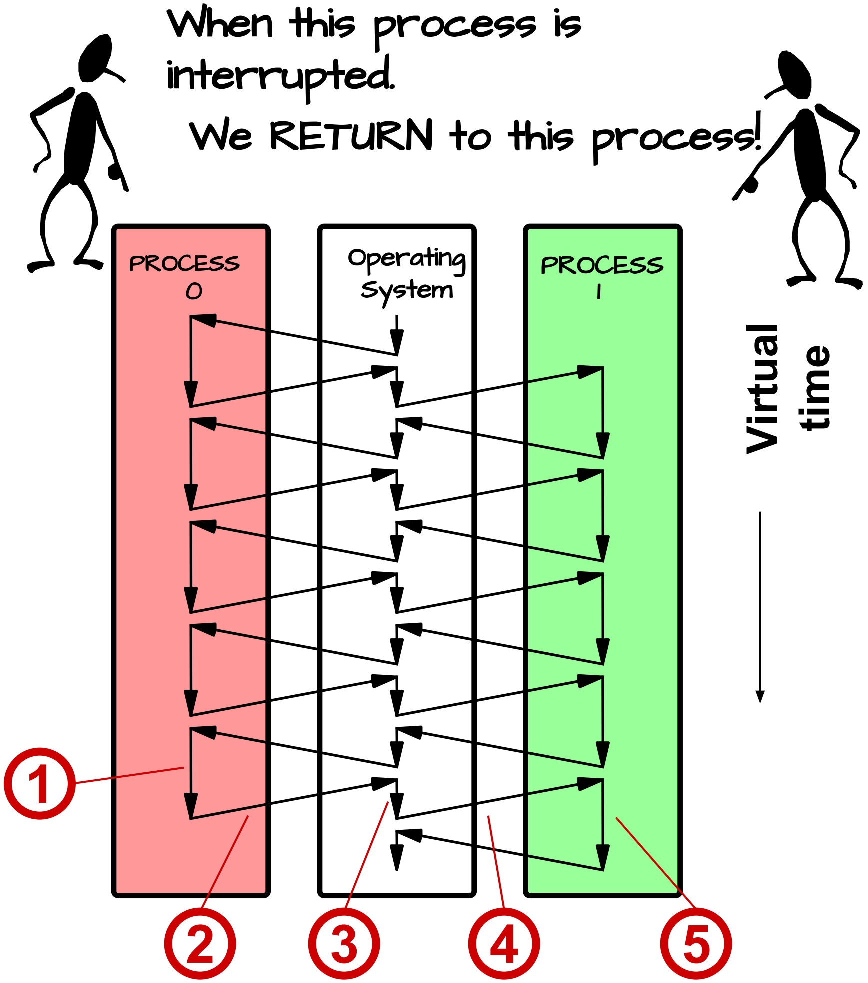


Goal: give each program its own "VIRTUAL MACHINE";
programs don't "know" about each other...

Abstraction: create a PROCESS, with its own

- machine state: r0, ..., r16, psr
- context (pagemap)
- stack
- program (w/ possibly shared code)
- virtual I/O devices (console...)

MULTIPLEXING THE CPU



When this process is interrupted.

We RETURN to this process!

1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC in \$27 (\$k1)
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like a return from other trap handlers (ex. jr \$27) but we're returning from a different trap than happened in step 2!
5. Running in process #1

And, vice versa.

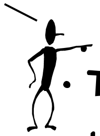
Result: Both processes get executed, and no one is the wiser

STACK-BASED INTERRUPT HANDLING

The interrupt forces these 2 instructions to be executed (similar to reset)

BASIC SEQUENCE:

- Program A is running when some EVENT happens.
- PROCESSOR STATE saved on stack (like a procedure CALL)

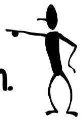


```
srmfd sp!,{r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,sp,lp,pc};  b handler
```

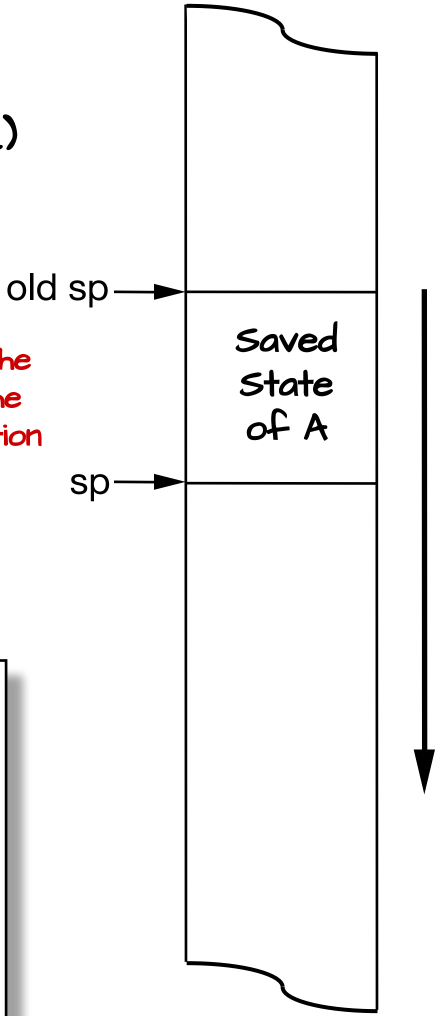
- The HANDLER program to be run is selected.
- HANDLER runs to completion
- State of interrupted program A is re-installed

```
lrmfd sp!,{r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,sp,lp,pc}
```

- Program A continues, unaware of interruption.



R13 will have the address of the "next" instruction before the interrupt



CHARACTERISTICS:

- *TRANSPARENT* to interrupted program!
- Handler runs to completion before returning
- Obeys stack discipline: handler can "borrow" stack from interrupted program (and return it unchanged) or use a special handler stack.

EXTERNAL (ASYNCHRONOUS) INTERRUPTS

Example:

System maintains current time of day (TOD) count at a well-known memory location that can be accessed by programs.
This value must be updated periodically in response to
A clock "interupt" triggered perhaps 100 times per second.

Program A (Application)

- Executes instructions of the user program.
- Doesn't want to know about clock interrupts
- Checks TOD by examining the memory location.

Clock Handler

- GUTS: Sequence of instructions that increments TOD. Written in C.
- Entry/Exit sequences save & restore interrupted state, call the C handler. Written as assembler "stubs".

INTERRUPT HANDLER CODING

"Interrupt stub" (written in assembly)

```
Clock_h: mov   r0,#User
         mov   r1,16
save:    ldr   r2,[sp,r1,lsl #2]
         str   r2,[r0,r1,lsl #2]
         subs  r1,r1,#1
         bne   save
         bl    Clock_Handler
         mov   r0,#User
         mov   r1,16
restore: ldr   r2,[r0,r1,lsl #2]
         str   r2,[sp,r1,lsl #2]
         subs  r1,r1,#1
         bne   restore
         mov   r0,#UMODE
         msr   r0,PSR
         lrmfd sp!,{r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,sp,lp,pc}
```

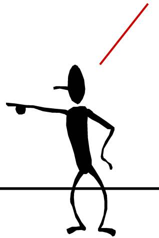
Handler (written in C)

```
long TimeOfDay;
struct Mstate { int R1,R2,…,SP,LP,PC } User;

/* Executed 100 times/sec */
Clock_Handler() {
   TimeOfDay = TimeOfDay + 10; // in milliseconds
}
```

TIME-SHARING THE CPU

We can make a small modification to our clock handler implement time sharing.

```
long TimeOfDay;
struct Mstate { int R1,R2,…,SP,LP,PC } User;

/* Executed 100 times/sec */
Clock_Handler(){
   TimeOfDay = TimeOfDay + 10;
   if (TimeOfDay % QUANTUM == 0) Scheduler();
}
```



Our clock handler calls another function

A Quantum is that smallest time-interval that we allocate to a process, typically this might be 50 to 100 ms. (Actually, most OS Kernels vary this number based on the processes priority).

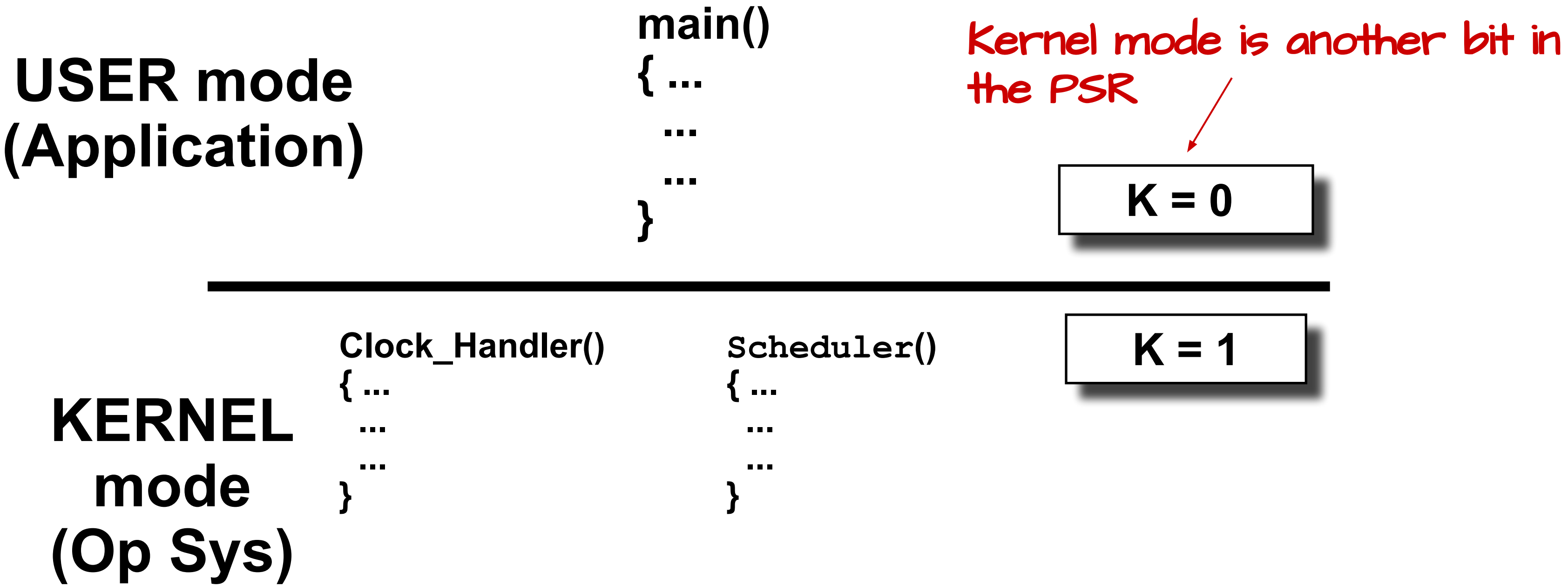
SIMPLE TIMESHARING SCHEDULER

```
long TimeOfDay;
struct Mstate { int R1,R2,…,SP,LP,PC } User;
.
.
.
struct PCB {
   struct MState State;          /* Processor state   */
   Context PageMap;              /* VM Map for proc   */
   int DPYNum;                   /* Console number    */
} ProcTbl[N];                    /* one per process   */

int Cur = 0;                     /* "Active" process  */

Scheduler() {
   ProcTbl[Cur].State = User;    /* Save Cur state */
   Cur = (Cur+1) % N;            /* Incr mod N     */
   User = ProcTbl[Cur].State;    /* Install for next User */
}
```

(PCB = Process Control Block)

AVOIDING RE-ENTRANCE

Handlers which are interruptable are called *RE-ENTRANT*, and pose special problems... miniARM, like many systems, disallows reentrant interrupts! Mechanism: Interrupts are disabled in "Kernel Mode":



USER mode (Application)

```
main()
{ ...
  ...
  ...
}
```

Kernel mode is another bit in the PSR

K = 0

K = 1

KERNEL mode (Op Sys)

```
Clock_Handler()
{ ...
  ...
  ...
}
```

```
Scheduler()
{ ...
  ...
  ...
}
```

OTHER INTERRUPT SOURCES

Asynchronous Inputs:

Keyboard, mouse events, disk access, etc.

Ex: On a keystrike a special type of handler called a "device driver" saves the key-code at a known location (much like the TimeOfDay variable), and clears a "buffer empty" flag.

User code reads this value when needed from the known location. But, if no key has been struck, what then?

WAITING IS WASTEFUL

The user code could sit in a loop waiting for the buffer-empty location to be cleared. This is called a "spin-lock".

This procedure is possibly user code.

```
keycodeType ReadKey()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* Nothing to do but wait */
    }
    return ReadInputBuffer(kbdnum);
}
```

Wastes CPU cycles until quantum is over.

READKEY SYNCHRONOUS SYSCALL

This procedure is performed as a kernel service…

```
keycodeType ReadKey_Handler()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        User.pc = User.pc - 4;
        Scheduler( );
    }
    return ReadInputBuffer(kbdnum);
}
```

BETTER: On I/O wait, YIELD remainder of time slot (quantum):

RESULT: Better CPU utilization!! Samples event every quantum.

FALLACY: Timesharing causes a CPUs to be less efficient

SOPHISTICATED SCHEDULING

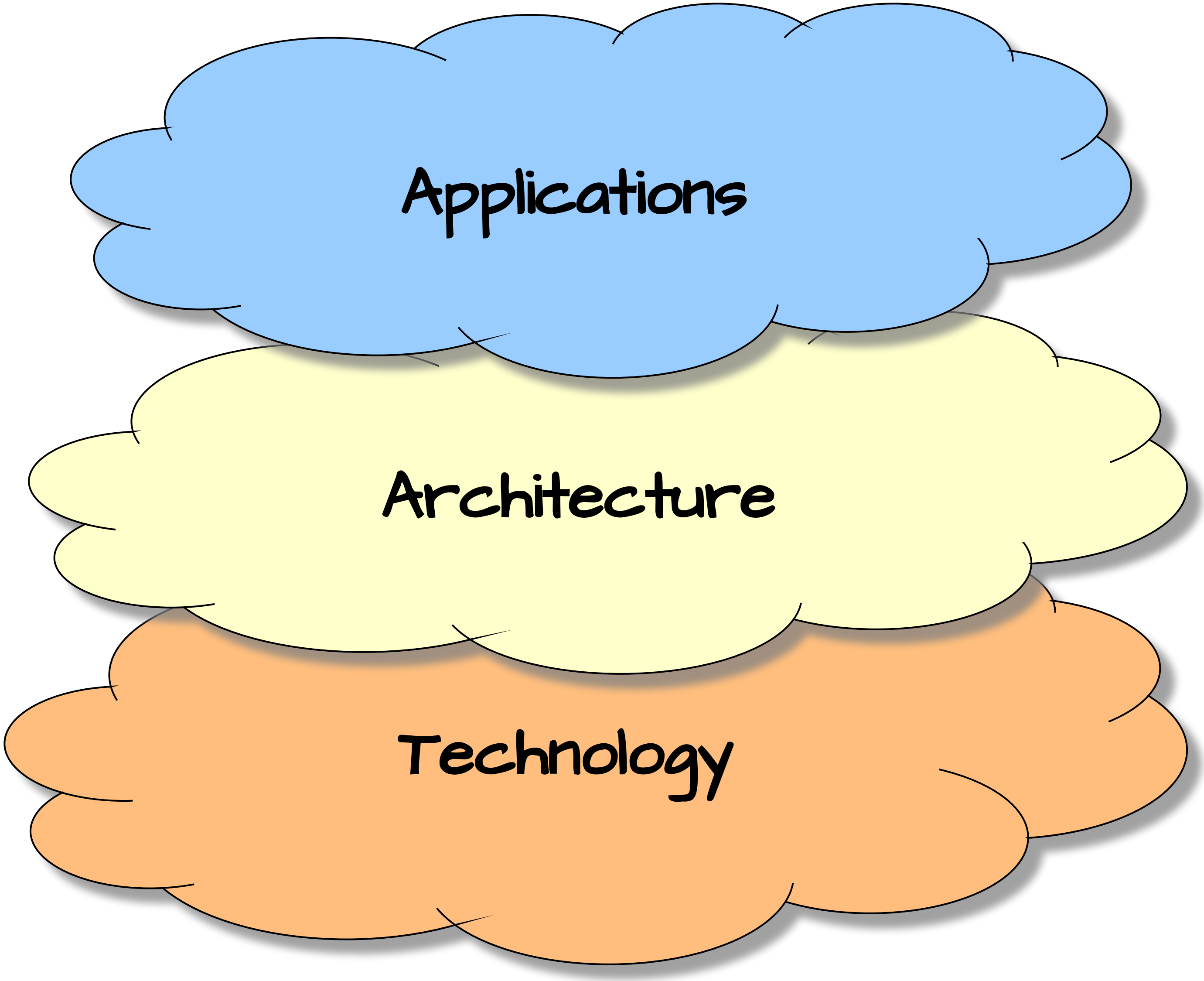
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- Processes can be in ACTIVE or WAITING ("sleeping") states;
- Scheduler cycles among ACTIVE PROCESSES only;
- Active process moves to WAITING status when it tries to read a character and buffer is empty;
- Waiting processes each contain a code (eg. in PCB) designating what they are waiting for (eg. keyboard N);
- Device interrupts (eg. on keyboard N) move any processes waiting on that device to ACTIVE state.

UNIX kernel utilities:

- sleep(reason) - Puts CurProc to sleep. "Reason" is an arbitrary binary value giving a condition for reactivation.
- wakeup(reason) - Makes active any process in sleep(reason).

411 WAS AN INTRODUCTION TO COMPUTER SCIENCE "SYSTEMS"



SYSTEMS: 2018

Tablet computing, Client computing (Chrome, HTML 5), Cloud computing, E-commerce, Android, Arduino, IoT, Wireless, Streaming Media, ...

Von Neumann Architectures, Multi-Core
Procedures, Objects, Processes
(hidden: pipelining, superscalar, SIMD, ...)

CMOS: 4.3 billion transistors/chip
(2018 6-core/12 thread Kaby Lake)
10x transistors every 5 years
1% performance/week!

SYSTEMS 2025?

Computer Science is the fastest changing field in the history of mankind!

Natural language/speech interfaces, Virtual Assistants, Computer vision, systems that "learn" rather than require programming, field-programmable microbes, direct brain interfaces, human augmentation ...

Von Neumann Architecture???
1024-way multicore?
Neural Nets?
How will we program them?

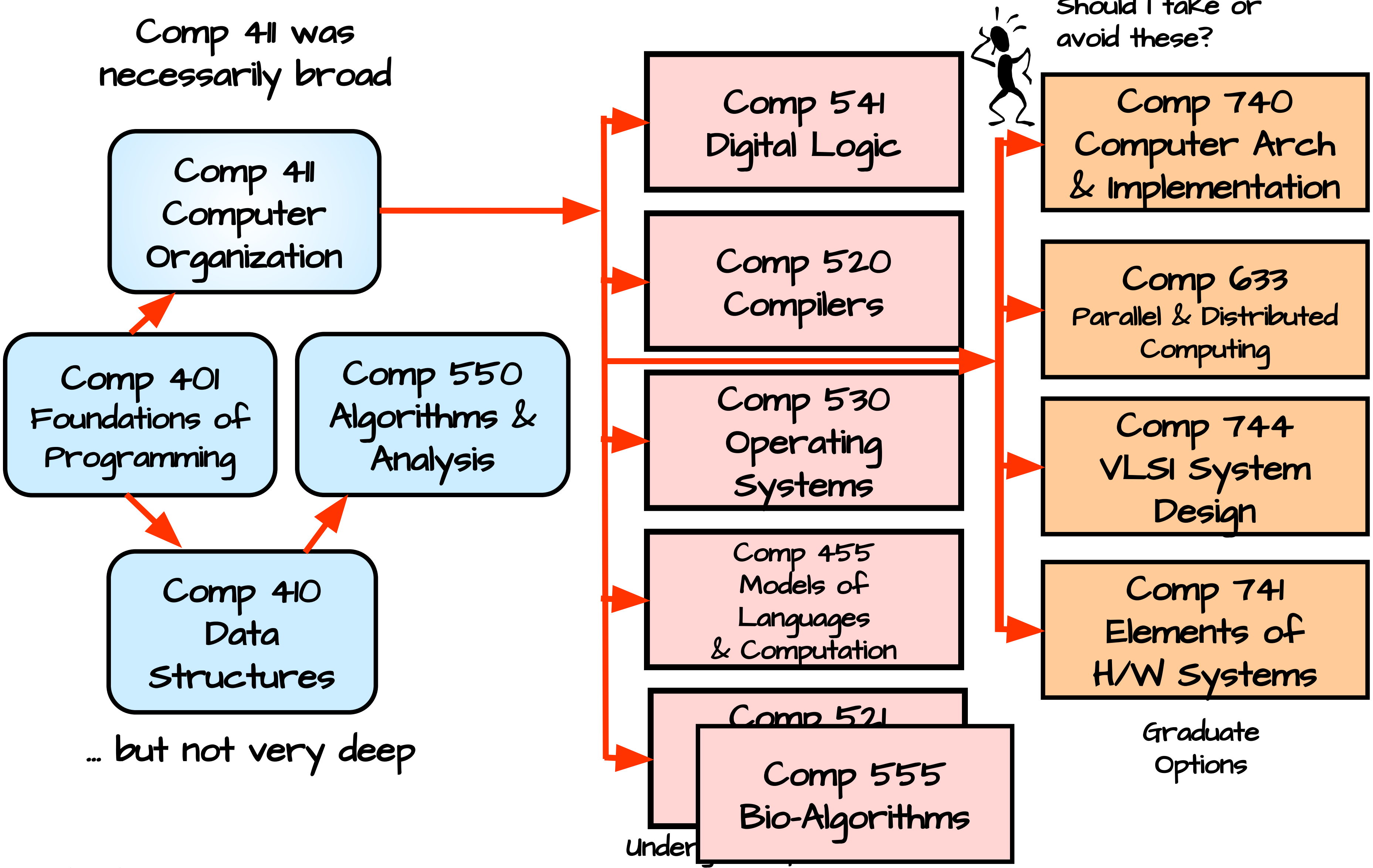
CMOS:
450 billion transistors
10 GHz clock

To predict his stuff, follow the news and think creatively

This is the hard part.

This stuff is relatively easy to predict.

WHAT NEXT? SOME OPTIONS...



Comp 411 was necessarily broad

Comp 411
Computer
Organization

Comp 401
Foundations of
Programming

Comp 550
Algorithms &
Analysis

Comp 410
Data
Structures

... but not very deep

Comp 541
Digital Logic

Comp 520
Compilers

Comp 530
Operating
Systems

Comp 455
Models of
Languages
& Computation

Comp 521

Comp 555
Bio-Algorithms

Under

Should I take or avoid these?

Comp 740
Computer Arch
& Implementation

Comp 633
Parallel & Distributed
Computing

Comp 744
VLSI System
Design

Comp 741
Elements of
H/W Systems

Graduate
Options