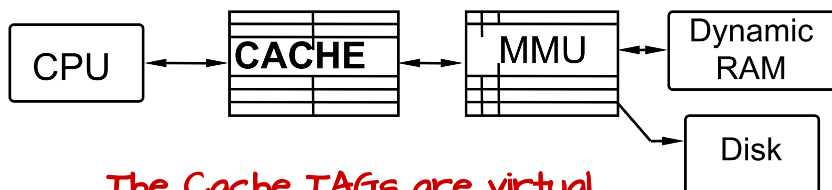
Almost Over

1) Last Problem Set is due Tonight
2) Final Exam on Thursday 12/8 from noon-3pm
   40 questions - Open book, open notes, open internet
   ~20 on pipelining, pipelining CPUs, caches, virtual memory
   ~20 on earlier course material

Using Caches with Virtual Memory

Virtual Cache
Tags match virtual addresses

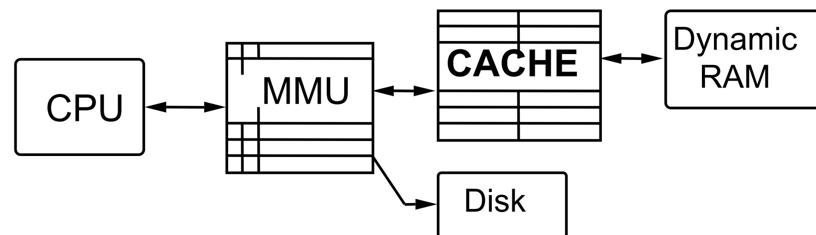
Physical Cache
Tags match physical addresses



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

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

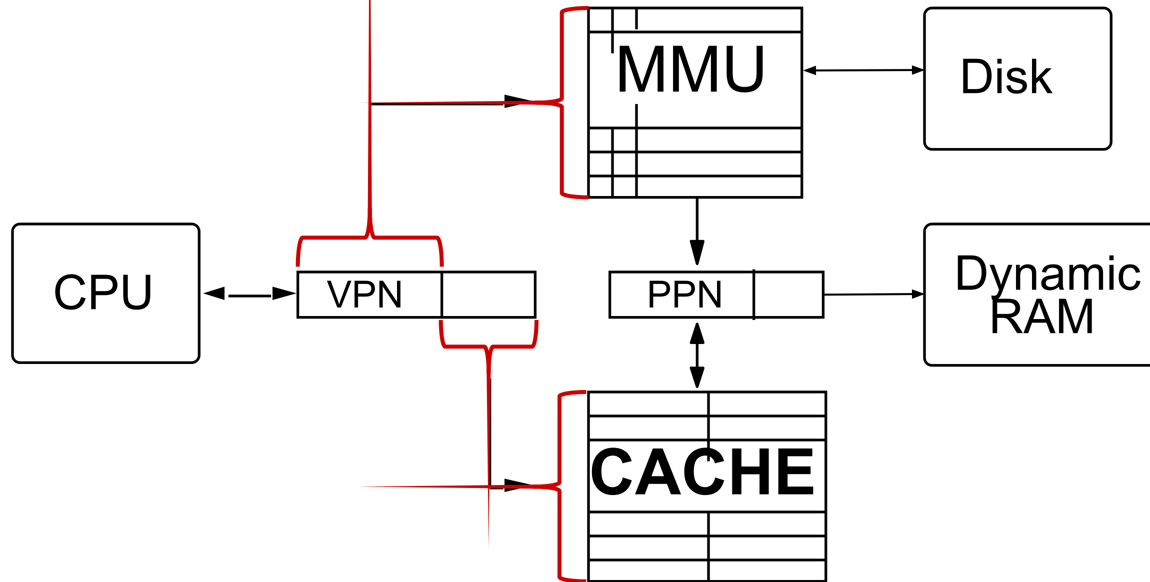
These TAGs are physical, they hold addresses after translation.



- 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

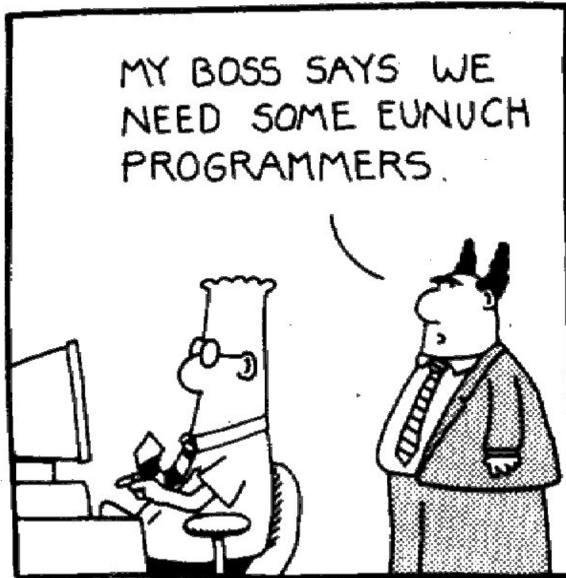


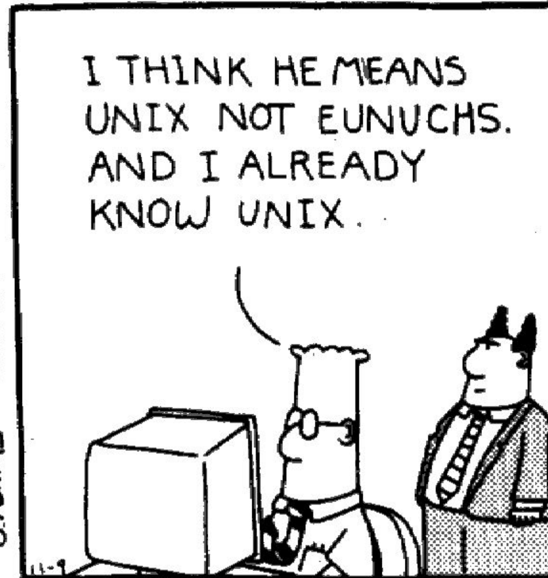
OBSERVATION: If cache line selection is based on *unmapped* page offset bits, RAM access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page number from MMU.

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

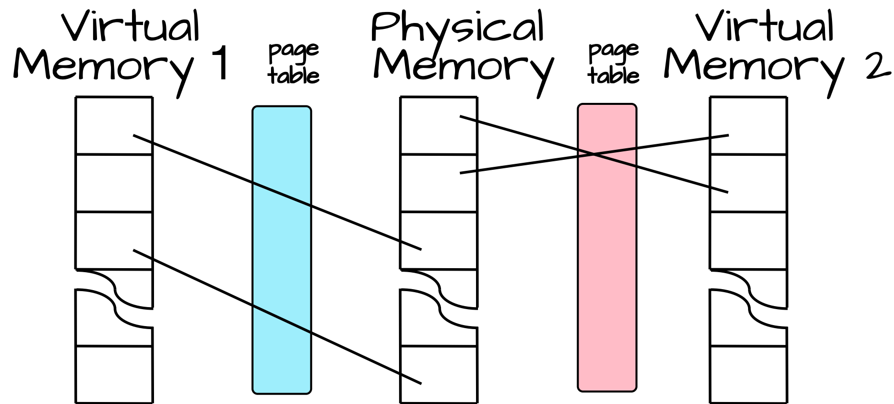
Virtual Machines & the OS Kernel

DILBERT by Scott Adams





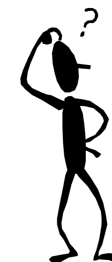
Power of Contexts: Sharing a CPU



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

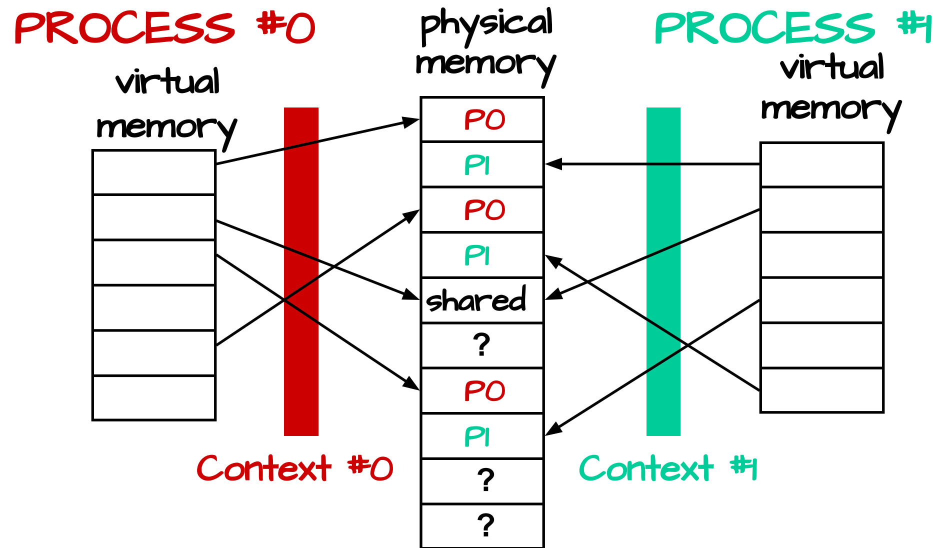
More than Virtual Memory
A VIRTUAL MACHINE

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

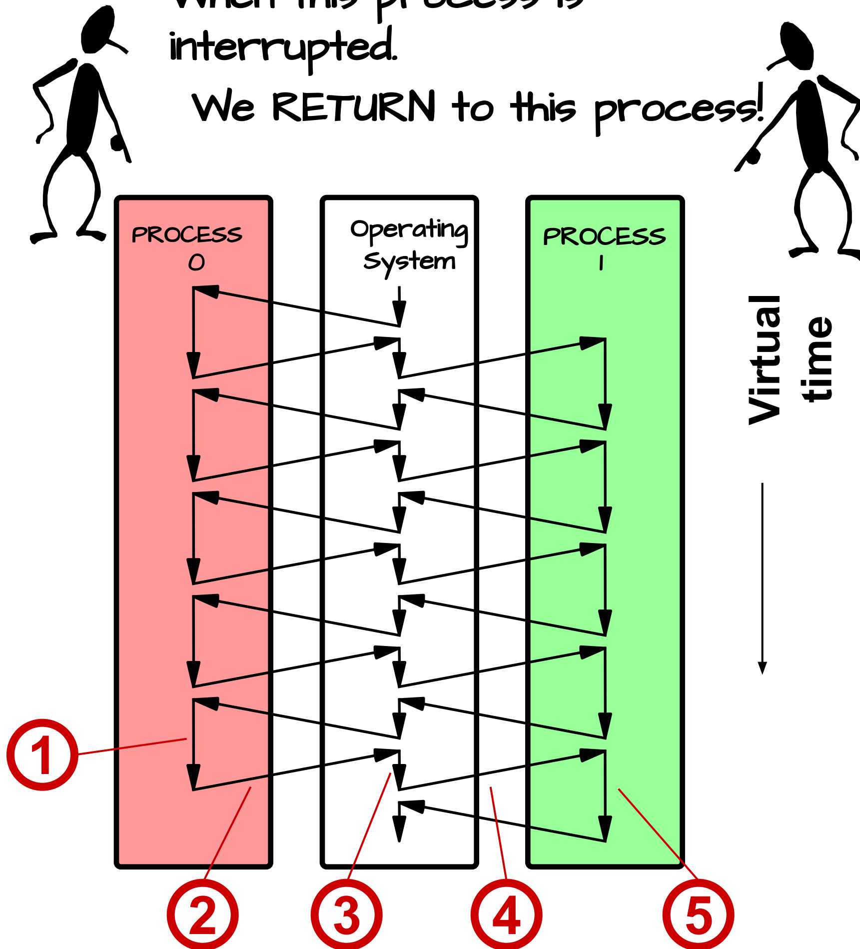


Goal: give each program its own "VIRTUAL MACHINE";
programs don't "know" about each other...

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



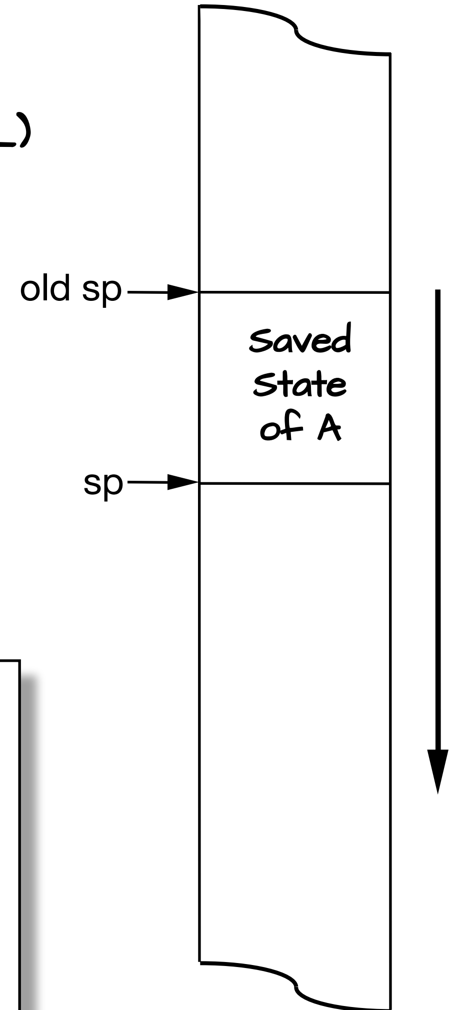
1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC in $27 ($k1)
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just 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

BASIC SEQUENCE:

- Program A is running when some EVENT happens.
- PROCESSOR STATE saved on stack (like a procedure CALL)
- The HANDLER program to be run is selected.
- HANDLER runs to completion
- State of interrupted program A is re-installed
  - Program A continues, unaware of interruption.



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: sw    tp,savetp
         lui   tp,(User>>12)        # make tp point to
         addi  tp,tp,User           # "User" struct
         sw    x1,0(tp)             # Save registers of
         sw    x2,4(tp)             # interrupted
         ...                        # application pgm…
         sw    x31,124(tp)          # program
         addi  sp,x0,KStack         # Use KERNEL stack
         jal   Clock_Handler        # call handler
         lw    x1,0(tp)             # Restore saved
         lw    x2,4(tp)             # registers
         …
         lw    x31,124(tp)
         lw    tp,savetp
         jalr  x1                    # Return to app.
```

Handler (written in C)

```
long TimeOfDay;
struct Mstate { int x1,x2,…,x31 } User;

/* Executed 60 times/sec */
Clock_Handler(){
   TimeOfDay = TimeOfDay + 1;
}
```

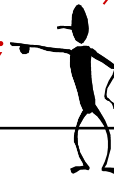
Time-Sharing the CPU

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

KERNEL mode (Op Sys)

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

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

K = 1

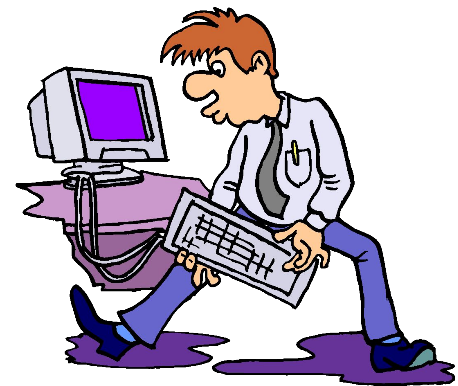
Other Interrupt Sources

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

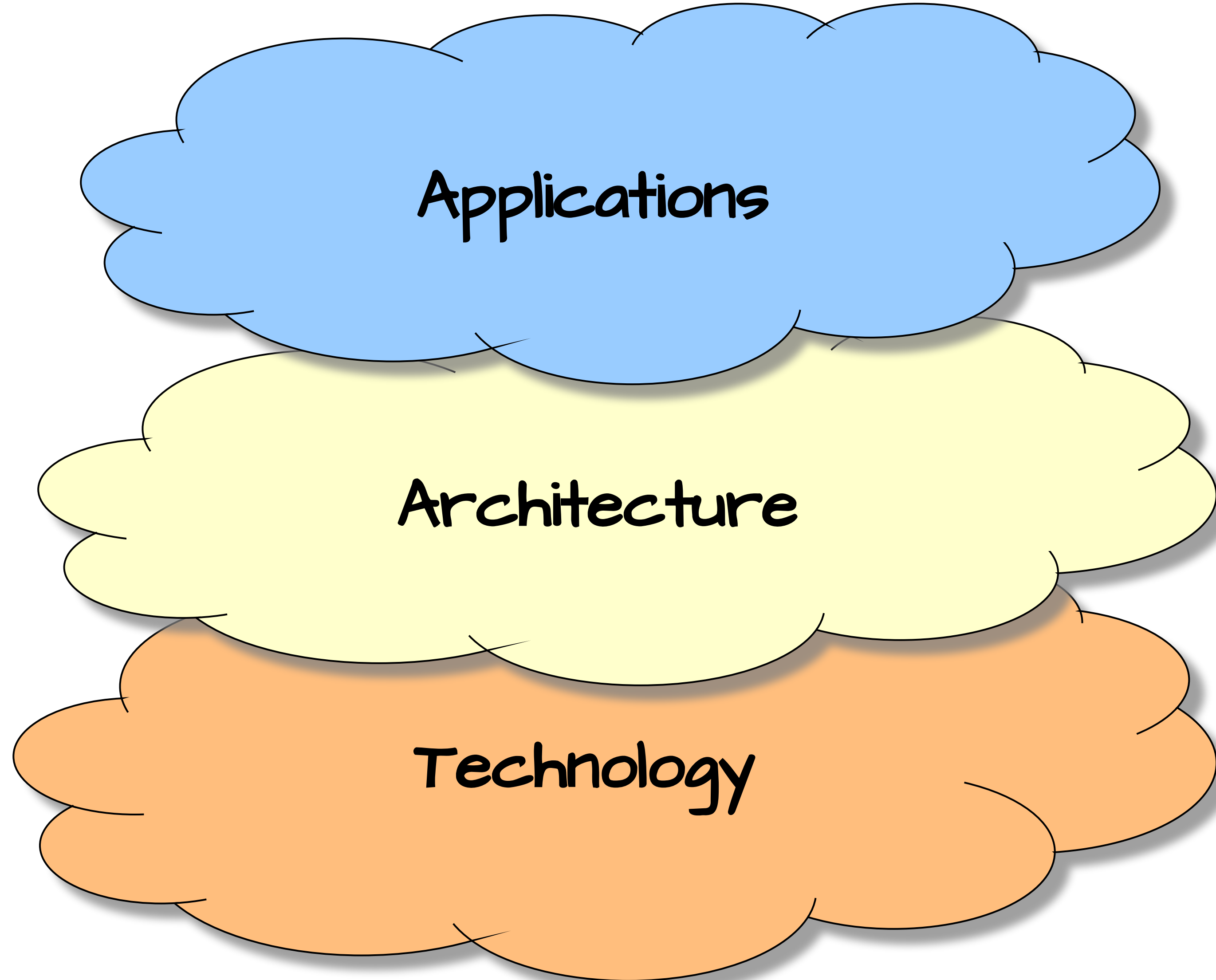
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- 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?

To predict his stuff, follow the news and think creatively

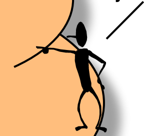
Natural language/speech interfaces, Virtual Assistants, Computer vision, systems that "learn" rather than require programming, field-programmable microbes, direct brain interfaces, human augmentation ...

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

This is the hard part.

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

This stuff is relatively easy to predict.



CMOS:
450 billion transistors
10 GHz clock

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