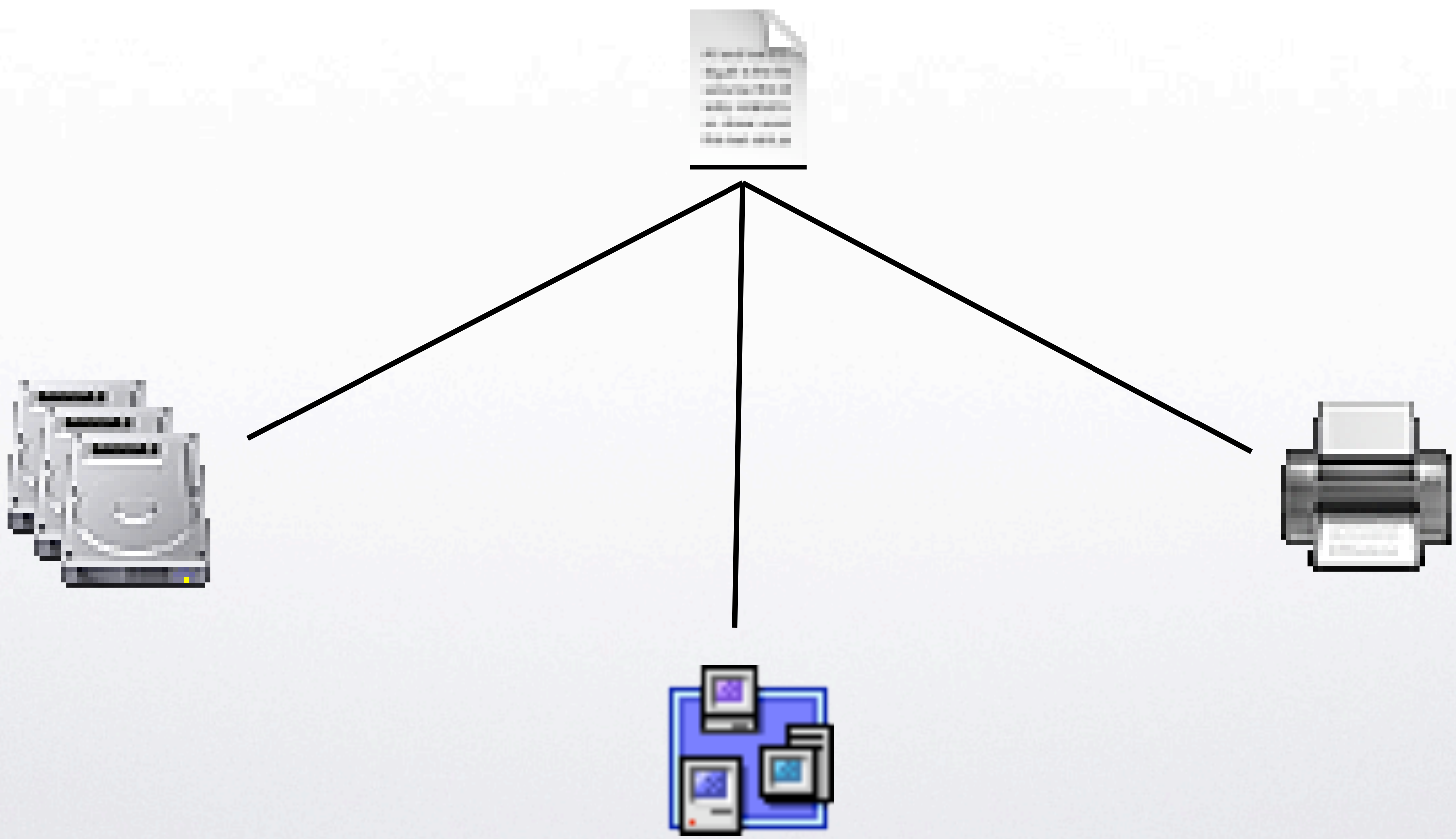
CS 4410 Operating Systems

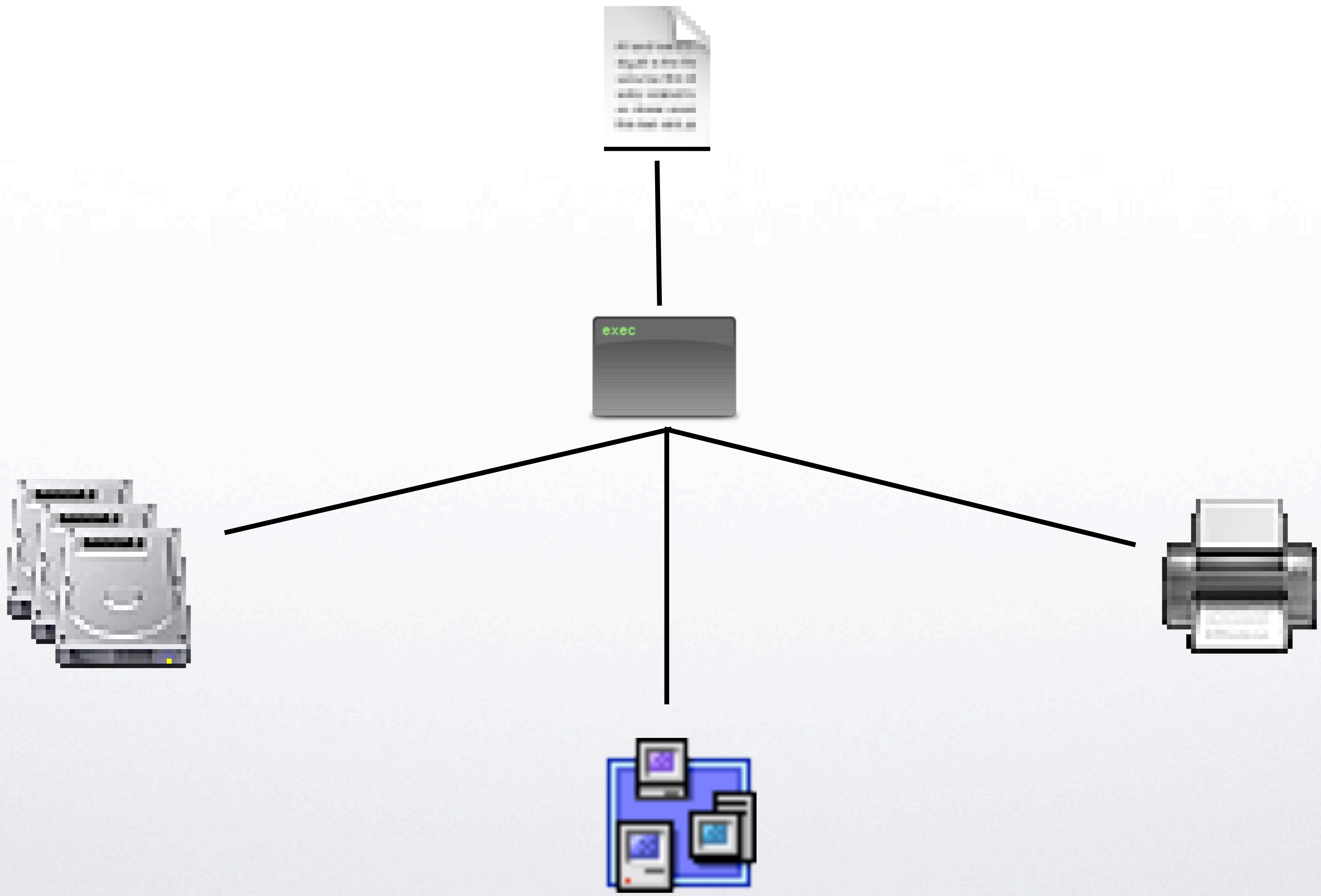
Computer Architecture Review

Oliver Kennedy

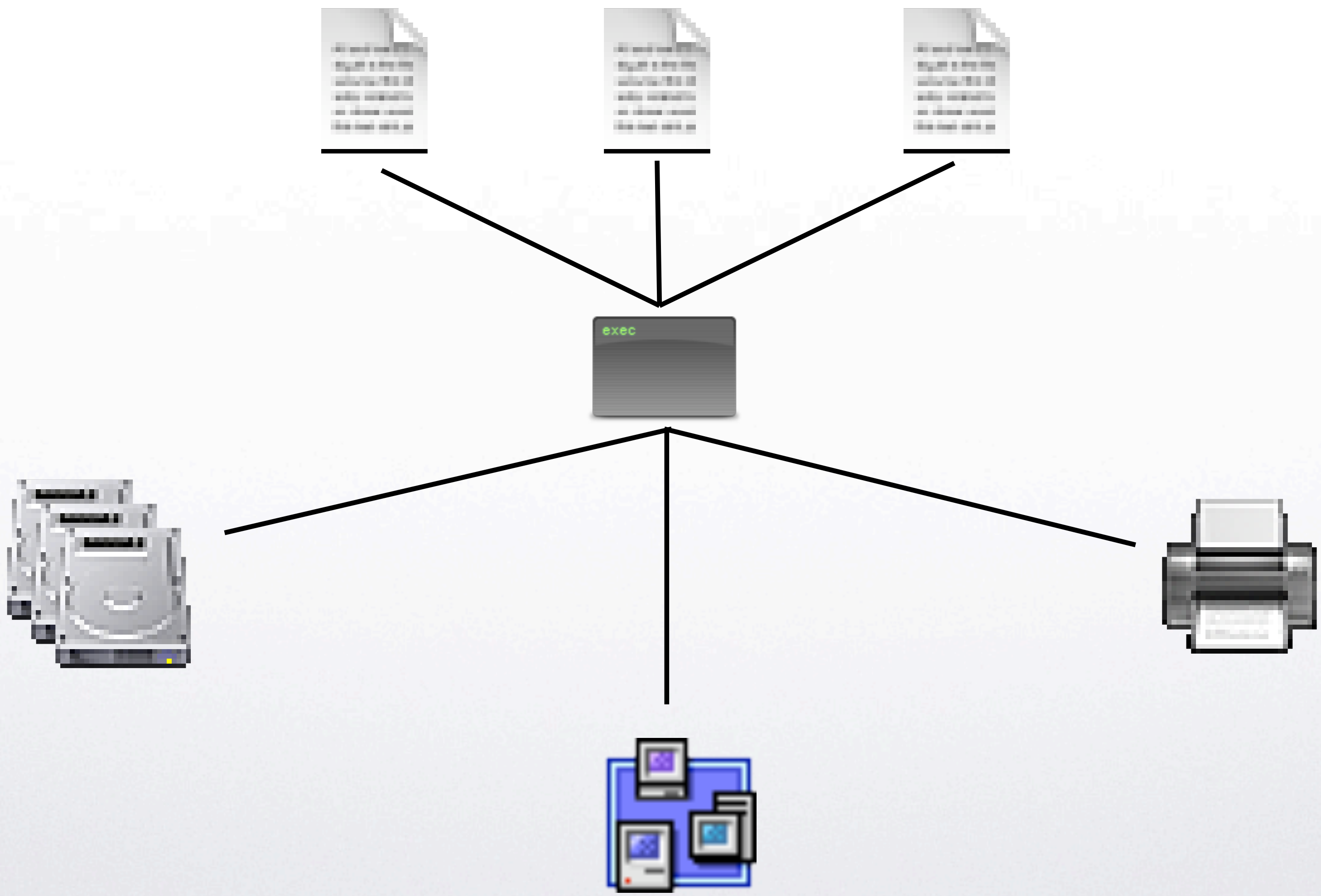
The Dawn of Computing



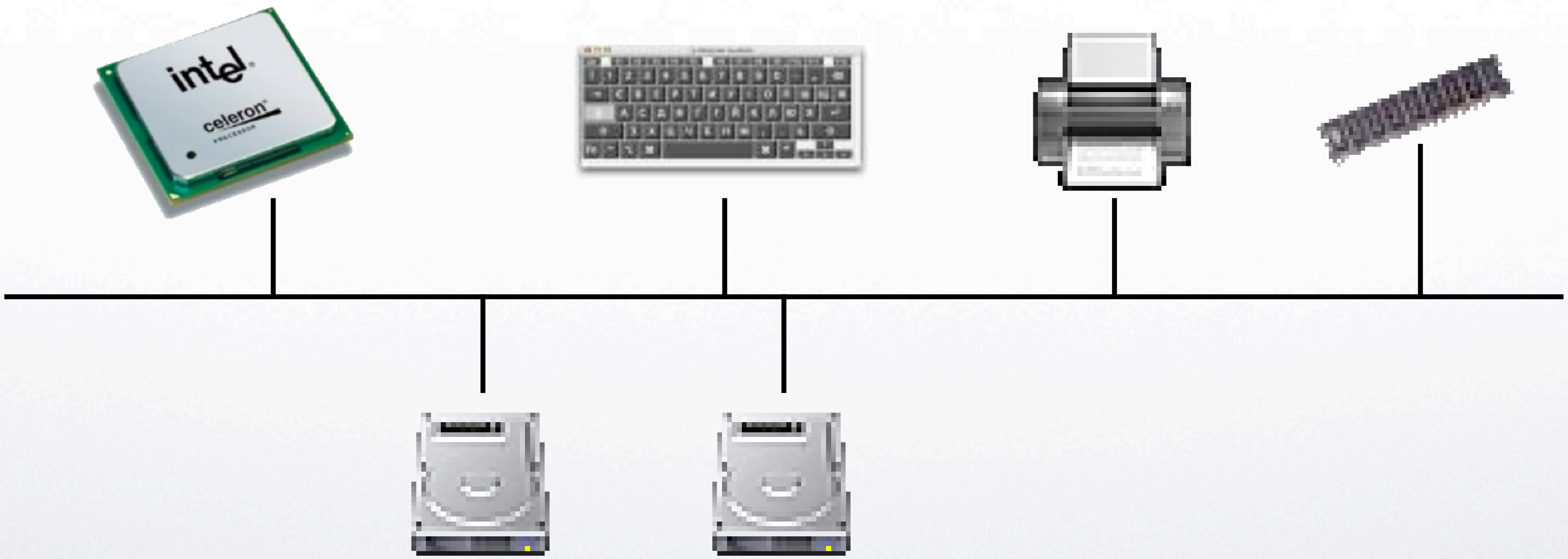
The OS!



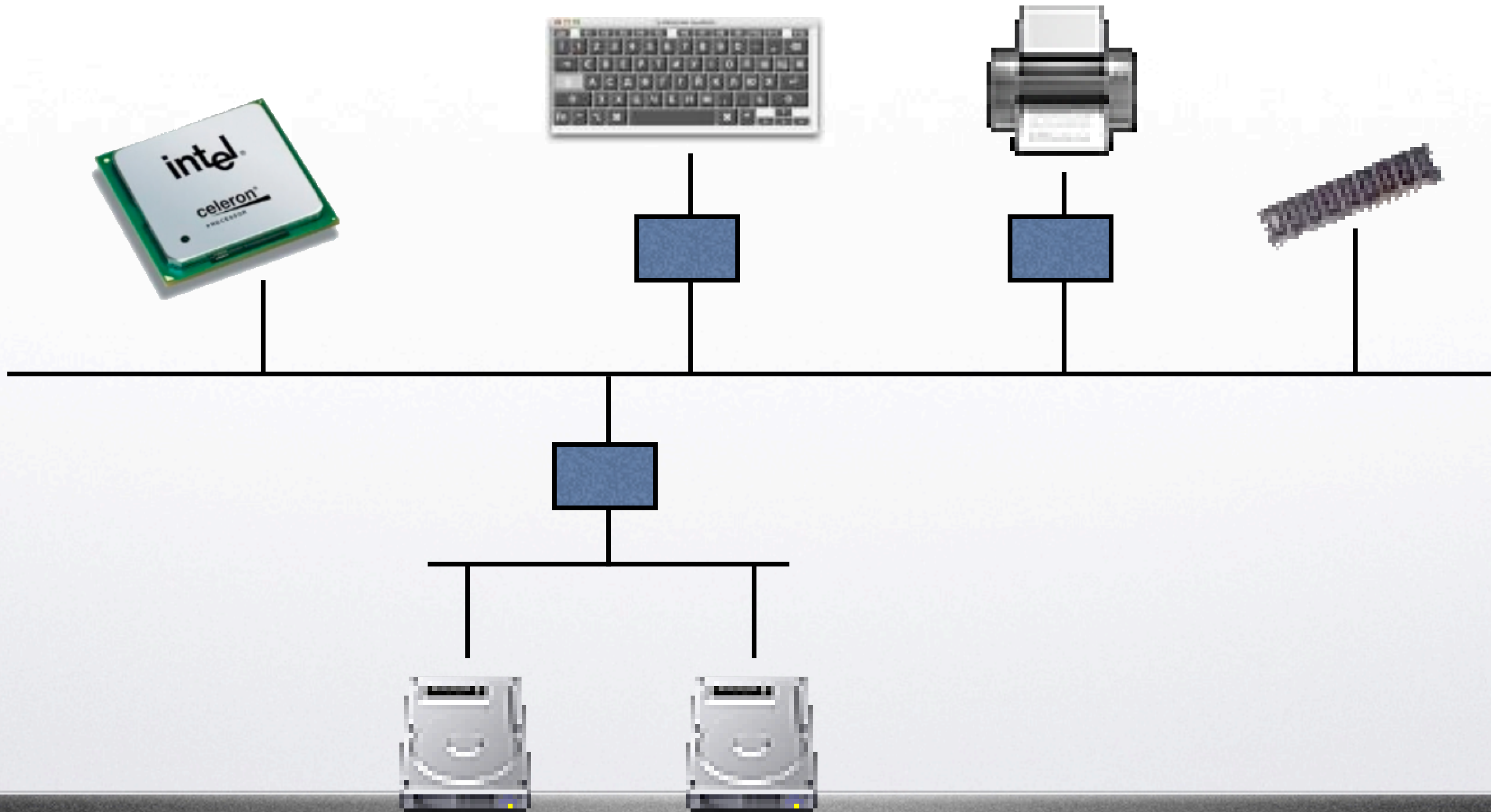
Multitasking



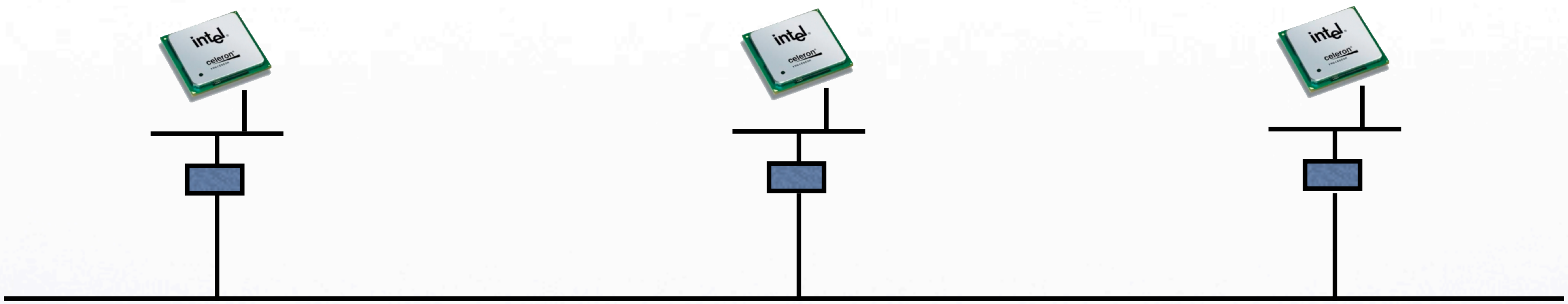
So what's under the hood?



Well... not quite



Networks



A CPU

- Registers:
  - The CPU's short term memory.
- Arithmetic Logic Unit:
  - Where most of the work gets done.
- Floating Point Unit:
  - Handles the "decimal" calculations.
- Caches:
  - Reduce memory access times.

The Pipeline

- A lot of computation goes into a single instruction.
- Can some of this computation be done in parallel?
- Set up an assembly line.
  - Each stage processes a little and passes it on.
  - Utilize hardware more intensively
  - Less work per stage means stages can run faster.
- Why not have lots and lots of stages?
  - What happens if we don't know what will happen next?
  - What happens if one instruction needs data from an earlier instruction?

The Pipeline

- Avoiding delays:
  - Branch Prediction.
  - Instruction Reordering.
- Currently, most pipelines are 10-15 stages in length.
  - Fetch the instruction
  - Decode/Dispatch the instruction.
  - Get necessary data.
  - Perform necessary calculations.
  - Write the results to registers/memory.

The Multicore Revolution

- Moore's law continues, but not like everyone expected.
  - More transistors, but the density is too high.
- How can we use the extra transistors?
  - Make one CPU into two, sixteen, sixty four... or more.
  - Do more at the same speed.
- Push towards multithreaded programming languages.
  - ... need OS support.

The Memory Hierarchy

- Registers: 8-64 integers/floats at a time.
  - Available immediately.
- L1 Cache: ~32KB Data, ~32KB Instructions.
  - Short access time (2-3 cycles).
- L2 Cache: 1-2 MB.
  - Moderate access time (~10-20 cycles).
- Main Memory: up to 4GB or more.
  - Long access time (on the order of 100 cycles).
- Prefetching is used to increase cache hits.

Why a Hierarchy?

- Tradeoff between speed and expense of hardware: very high-speed memory is expensive. (Also, physical distance can be a limit)
- Programs typically have strong *locality:* most accesses are near previous accesses, in both space and time.
  - Spatial locality: accesses to nearby addresses
  - Temporal locality: same resource accessed twice
- Can get very high cache hit rates with comparatively small cache.
  - Mostly stay on fast path

Stacks

- Functions in most languages execute in a LIFO order:
- Therefore, can store local variables on a stack:
  - Each function allocates an *activation record* by decrementing stack pointer register; can use that area for locals.
  - Increment SP to return.
  - Note that this is just a region of main memory accessed with stack discipline; hardware may not treat it specially
  - Can switch stacks by changing value of SP register.
- Note: not all programs use a stack; ML code typically won't.

Calling Conventions

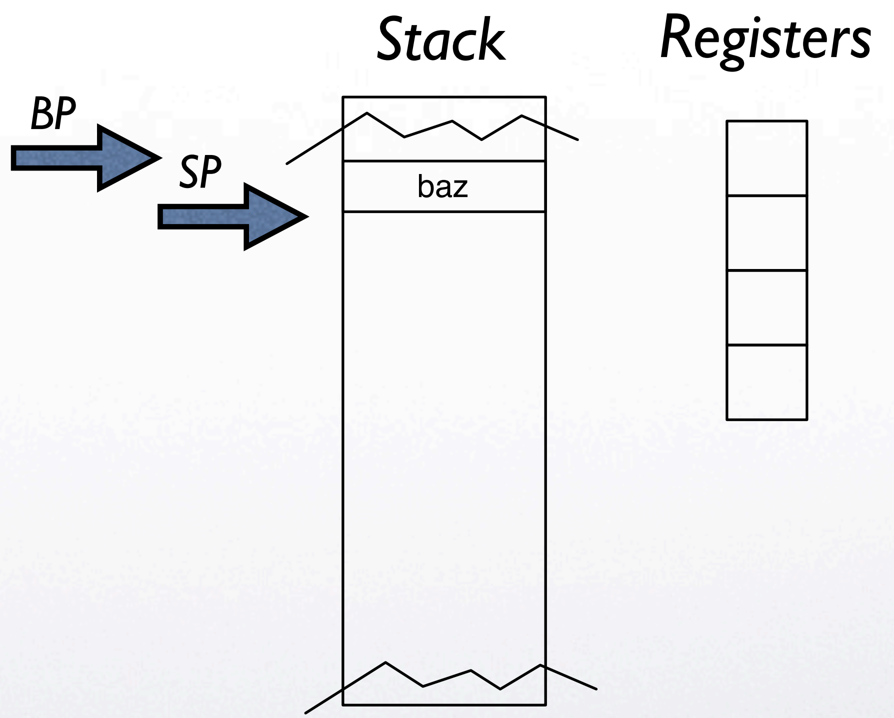
- Which registers can a function use? Where are parameters? Pushed in what order? Who removes them from stack?
- Check **calling convention**
- Example: typical conventions for IA32
  - EAX, ECX, EDX are caller-save, rest are callee-save
  - Args on stack, either left to right (stdcall) or r. to l. (cdecl)
  - In ‘stdcall’ convention, callee pops params from stack. In ‘cdecl’, caller.
    - In C, callee doesn’t always know how many params there are, since some functions are varargs.

Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

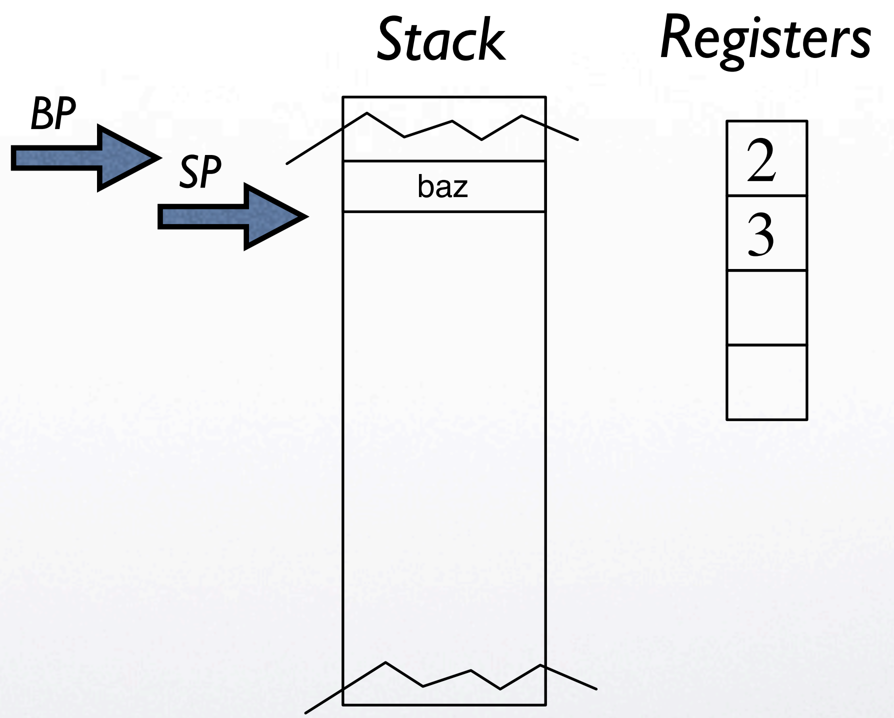


Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

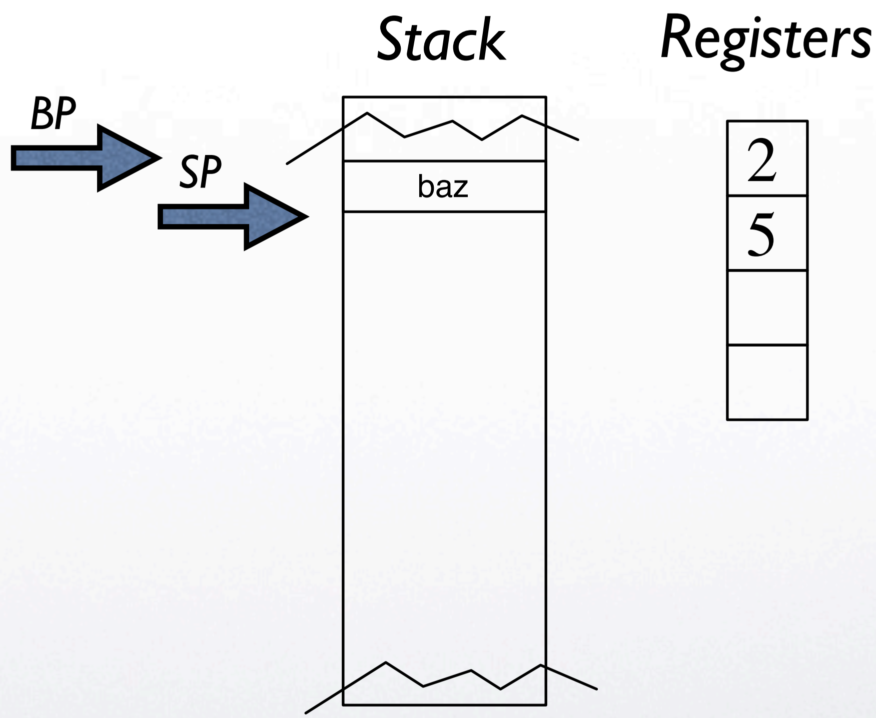


Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

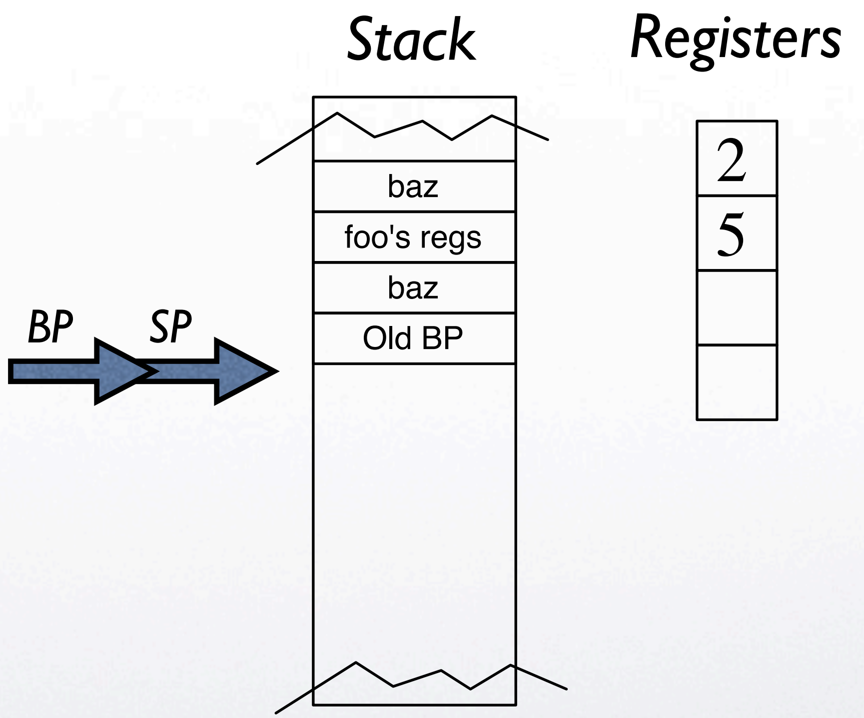


Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

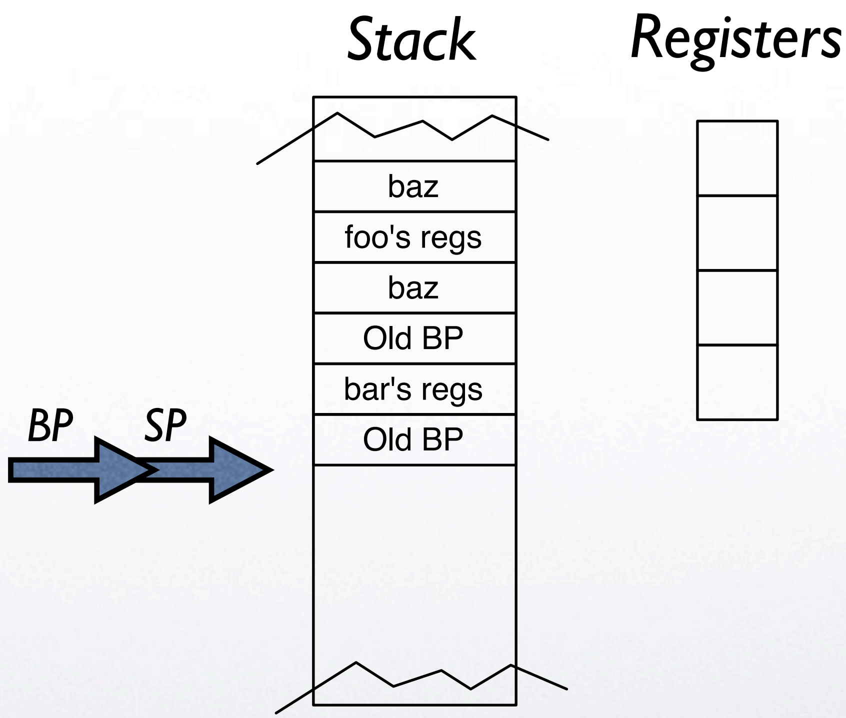


Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

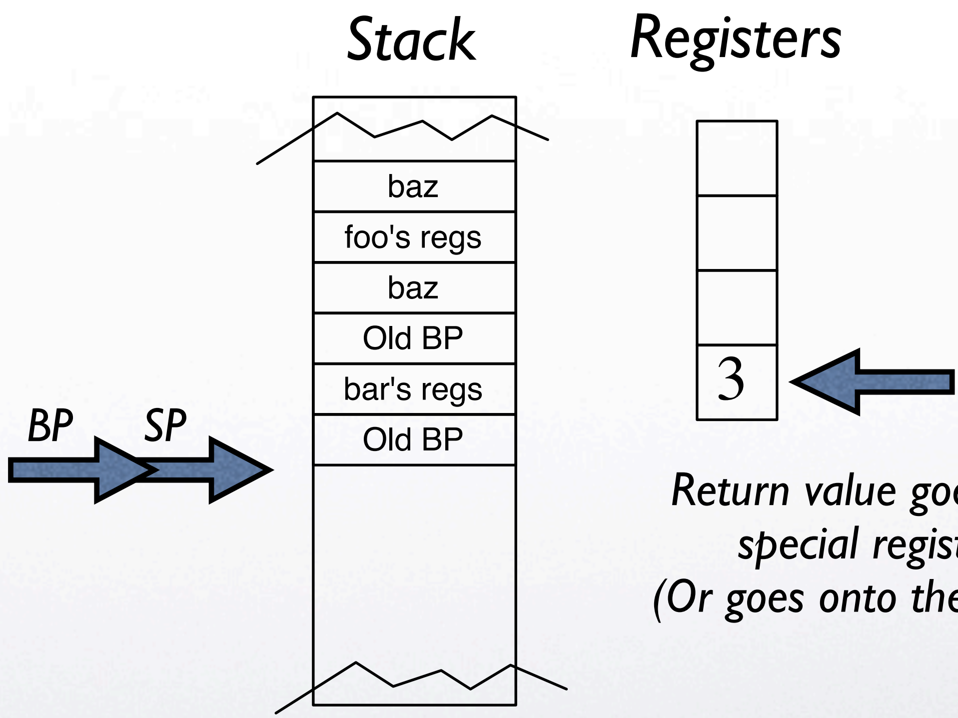


Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



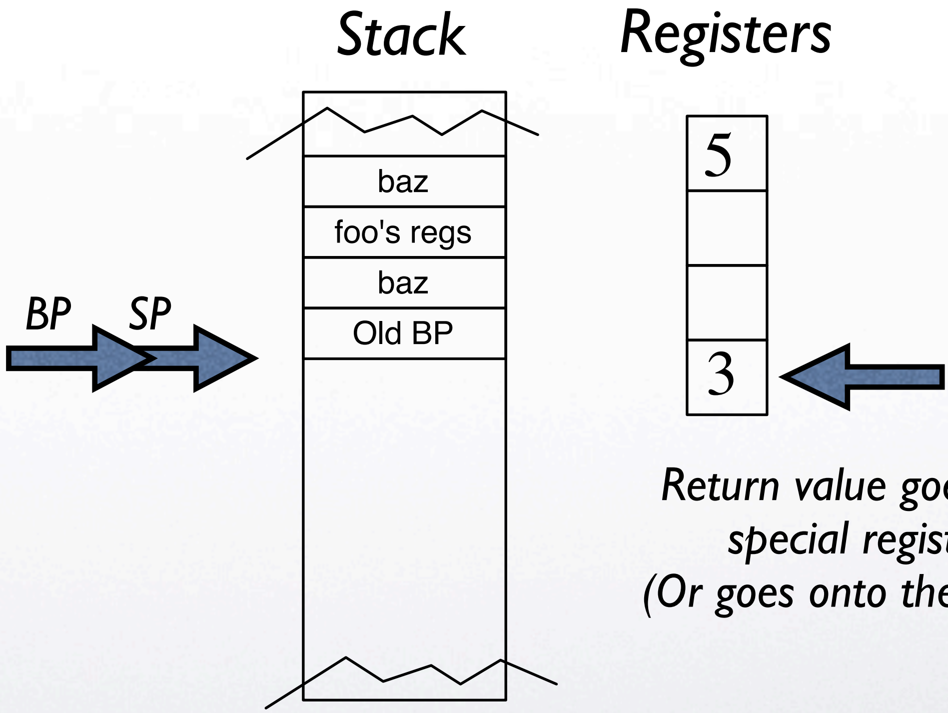
Return value goes in a special register (Or goes onto the stack)

Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



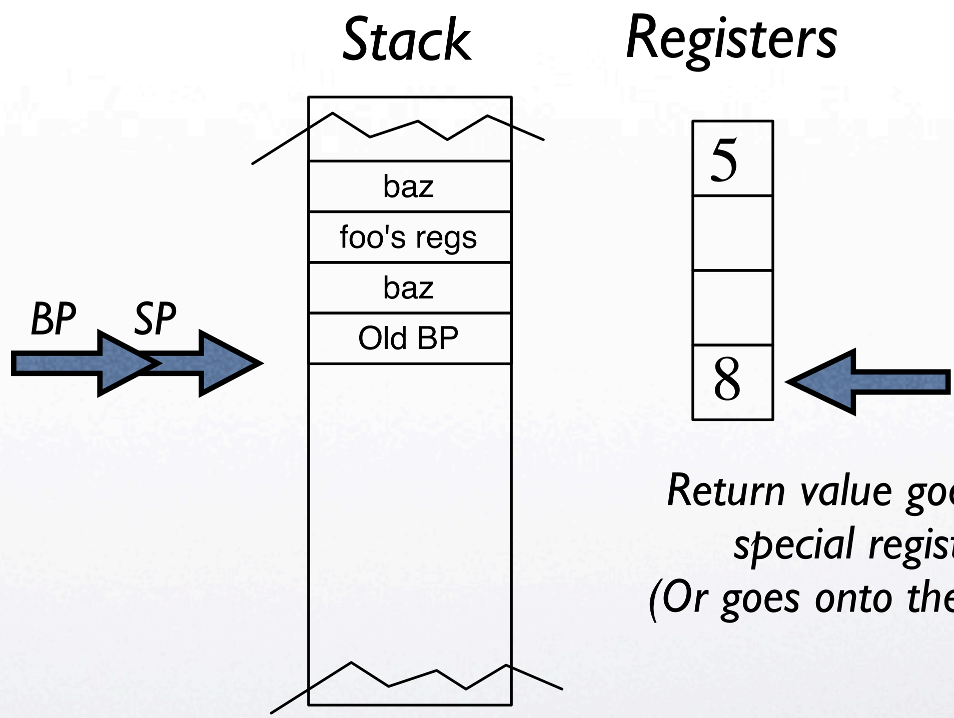
Return value goes in a special register (Or goes onto the stack)

Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



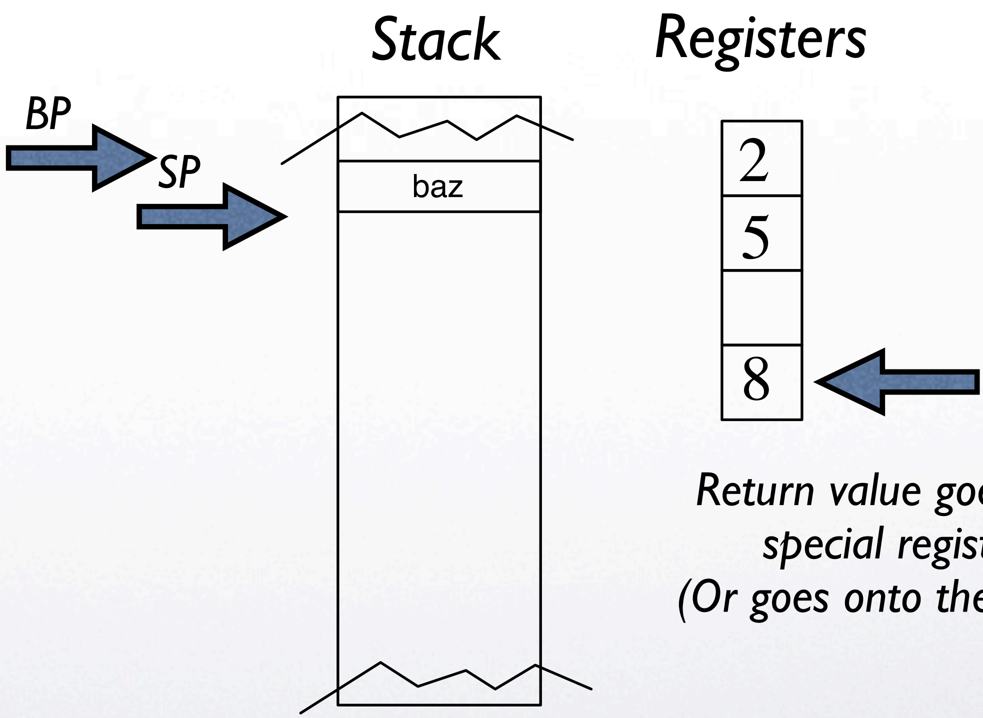
Return value goes in a special register (Or goes onto the stack)

Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



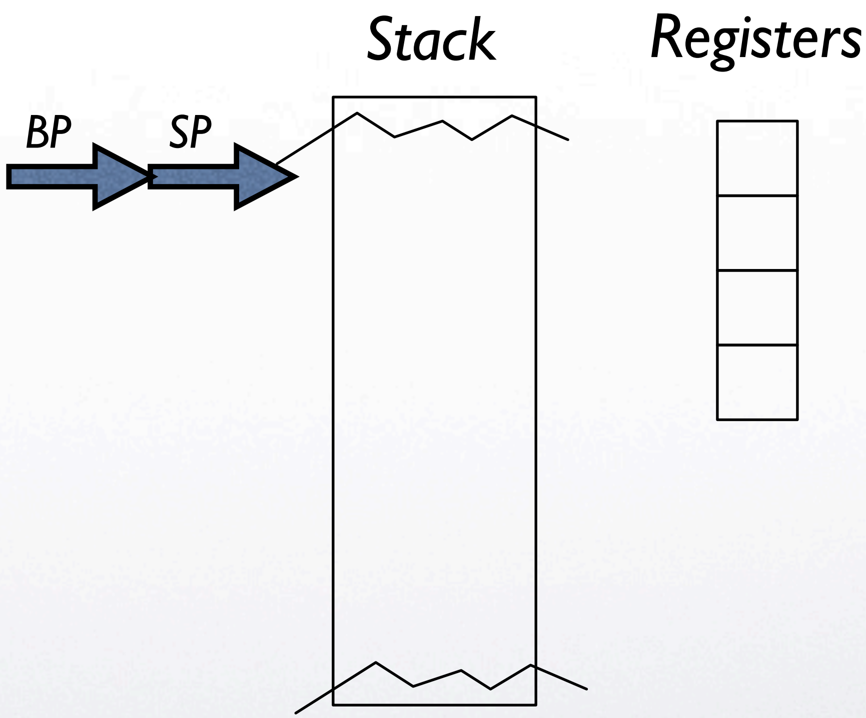
Return value goes in a special register (Or goes onto the stack)

Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



Traps/Interrupts

Traps/Interrupts

- What does the hardware do when something unexpected happens?
  - The software does something wrong. (Divide by 0)
  - The software asks for a wakeup call.
  - The user presses a key on the keyboard.

Traps/Interrupts

- What does the hardware do when something unexpected happens?
  - The software does something wrong. (Divide by 0)
  - The software asks for a wakeup call.
  - The user presses a key on the keyboard.
- It could just set a flag and have the software check for it.
  - Processor intensive.
  - Defeats the point of an operating system.

Traps/Interrupts

- What does the hardware do when something unexpected happens?
    - The software does something wrong. (Divide by 0)
    - The software asks for a wakeup call.
    - The user presses a key on the keyboard.
- It could just set a flag and have the software check for it.
    - Processor intensive.
    - Defeats the point of an operating system.
- Instead, the processor pauses what it's doing and and calls a callback.

Traps/Interrupts

Traps/Interrupts

- How does the processor know what code to execute?

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?
  - Disable interrupts while processing one.

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?
  - Disable interrupts while processing one.
  - Have multiple levels of interrupt.

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?
  - Disable interrupts while processing one.
  - Have multiple levels of interrupt.
  - Most architectures keep a short queue of interrupts waiting to be delivered.

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?
  - Disable interrupts while processing one.
  - Have multiple levels of interrupt.
  - Most architectures keep a short queue of interrupts waiting to be delivered.
- But what about traps?

Traps/Interrupts

- How does the processor know what code to execute?
  - Most architectures define a datastructure for a Interrupt Vector Table.
- What happens if an interrupt is interrupted?
  - Disable interrupts while processing one.
  - Have multiple levels of interrupt.
  - Most architectures keep a short queue of interrupts waiting to be delivered.
- But what about traps?
  - Identical to interrupts, except triggered by code (/ by 0).

10

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.
  - Also works with networks. (a packet just arrived)

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.
  - Also works with networks. (a packet just arrived)
- The hard drive's connected to the disk controller...

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.
  - Also works with networks. (a packet just arrived)
- The hard drive's connected to the disk controller...
  - A small piece of hardware (the controller) controls the IO device and communicates with the processor via a bus.

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.
  - Also works with networks. (a packet just arrived)
- The hard drive's connected to the disk controller...
  - A small piece of hardware (the controller) controls the IO device and communicates with the processor via a bus.
  - A small piece of software (a driver) interprets the data sent to the processor by the controller and communicates this information to the OS.

IO

- Hard disks are slow. Do we want to wait idly for data to arrive?
  - Signal data availability with an interrupt.
  - Also works with networks. (a packet just arrived)
- The hard drive's connected to the disk controller...
  - A small piece of hardware (the controller) controls the IO device and communicates with the processor via a bus.
  - A small piece of software (a driver) interprets the data sent to the processor by the controller and communicates this information to the OS.
  - Finally, the OS notifies the program that data is available.

Protection Levels

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?
  - Have multiple access levels. (user, system, etc..)

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?
  - Have multiple access levels. (user, system, etc..)
  - Restrict access to certain instructions in less privileged levels.

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?
  - Have multiple access levels. (user, system, etc..)
  - Restrict access to certain instructions in less privileged levels.
- Most architectures have an instruction to give up privileges, but no instruction to regain them.

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?
  - Have multiple access levels. (user, system, etc..)
  - Restrict access to certain instructions in less privileged levels.
- Most architectures have an instruction to give up privileges, but no instruction to regain them.
  - But how can the OS access the privileged instructions?

Protection Levels

- If the OS is going to be managing multiple programs, how does it stop them from misbehaving?
  - Have multiple access levels. (user, system, etc..)
  - Restrict access to certain instructions in less privileged levels.
- Most architectures have an instruction to give up privileges, but no instruction to regain them.
  - But how can the OS access the privileged instructions?
    - Traps! (The famous INT 21)

Memory Management: Segments

- The 80286 was a 16 bit processor.
  - It could address 2^16 (64k) of memory.
  - 64k is tiny! Couldn't it use more?
    - It divided memory up into 64k segments. Applications that needed more could set a segment register to change which segment their addresses pointed to.
- The 80386 was a 32 bit processor.
  - It could address 2^32 (4 gb) of memory.
  - Segments were still convenient for isolating applications from each other. (But it was messy)

Memory Management: Pages

- Segmentation is messy
  - All memory needs to be allocated upfront
  - Processes entering and leaving the system create holes.
- Instead, let's break memory up into a large number of equally-sized chunks (pages) and hand them to processes as needed.
- Create a Page Table
  - When the CPU is told to access an address in memory, it consults this table and translates the virtual address to a physical address.
  - A process still thinks it has the whole address space.

Memory Management: DMA

- Interrupts are slow.
  - Pausing running code takes a lot of work.
  - ... doubly so if you need to do anything complex.
- Transferring data from disk to memory involves a lot of interrupts.
- Let the IO controller write directly to memory.
  - This is called Direct Memory Access
- Another twist: Memory Mapped IO.
  - Let the disk controller pretend to be a portion of memory.