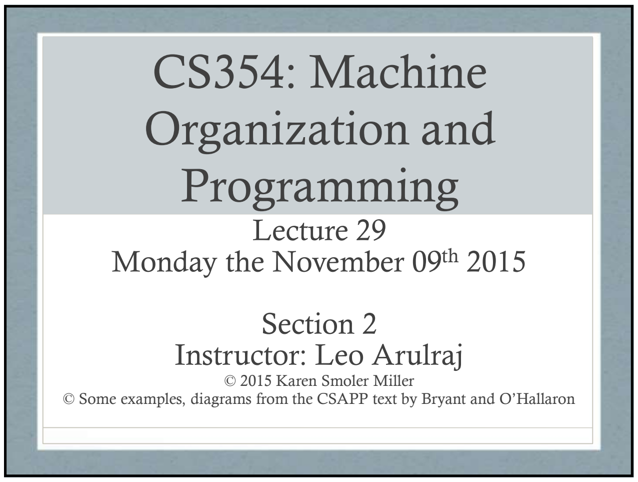
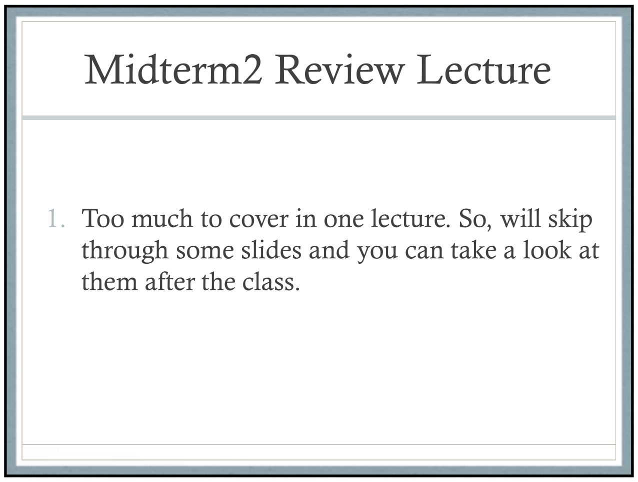


CS354: Machine Organization and Programming

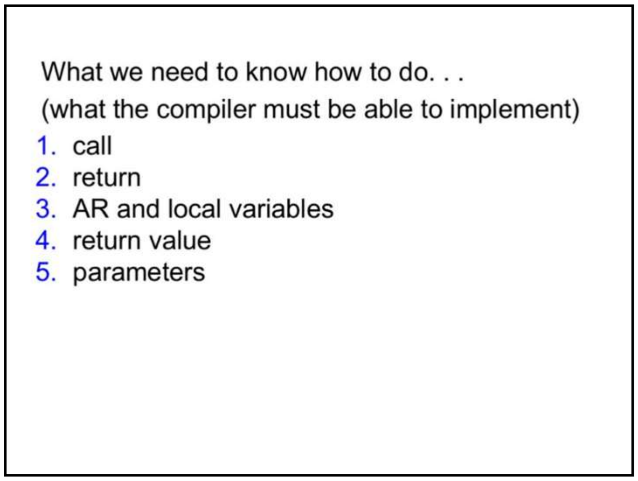
Lecture 29
Monday the November 09th 2015

Section 2
Instructor: Leo Arulraj



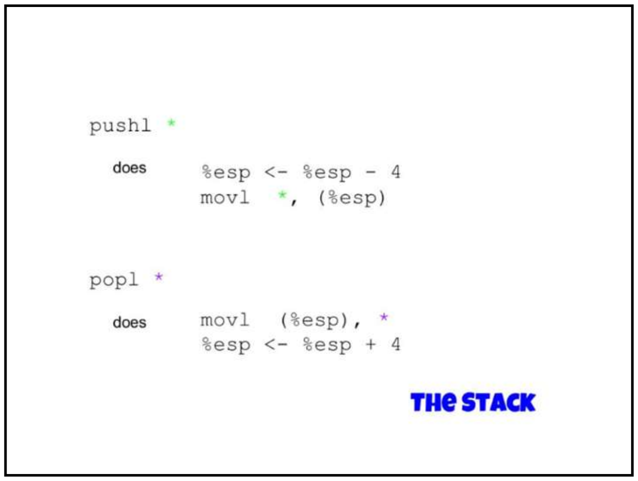
Midterm2 Review Lecture

1. Too much to cover in one lecture. So, will skip through some slides and you can take a look at them after the class.



What we need to know how to do. . .
(what the compiler must be able to implement)

1. call
2. return
3. AR and local variables
4. return value
5. parameters



```
pushl *

  does      %esp <- %esp - 4
            movl   *, (%esp)


popl *

  does      movl   (%esp), *
            %esp <- %esp + 4
```

1. call

- remember the return address
- go to fcn

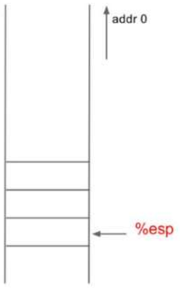
this is such a common operation that the x86 architecture supports it with a single instruction

```
    call fcn
```

does the equivalent of

```
    push %eip
    jmp fcn
```

⇐ PC



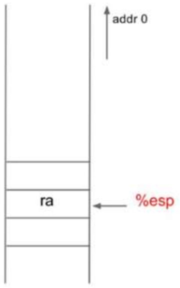
2. return

use the return address pushed onto the stack

```
    ret
```

does the equivalent of

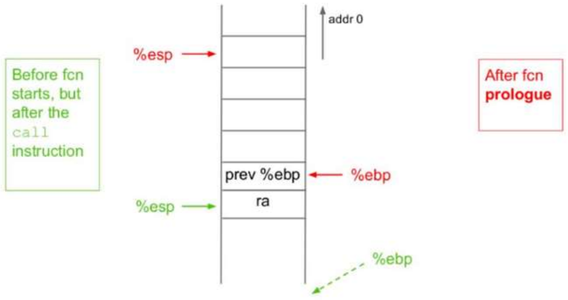
```
    popl %eip
```



3. incorporate AR

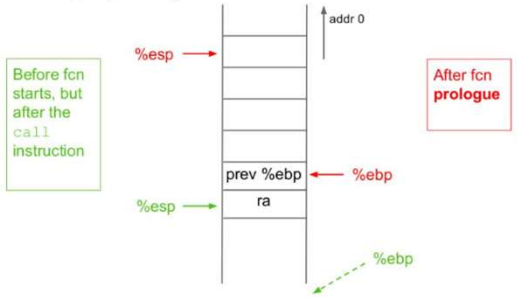
For example, assume we need AR space for 3 ints.

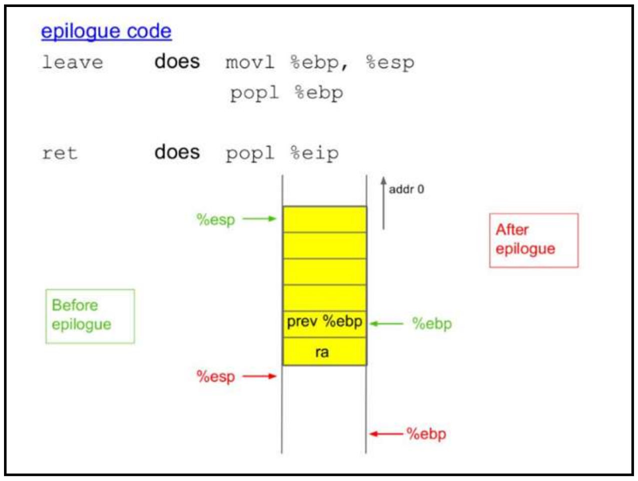
gcc on x86 allocates AR space in multiples of 16 bytes.



prologue code

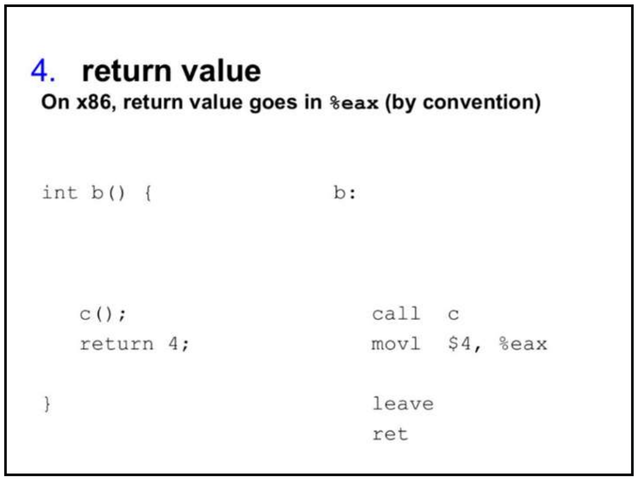
```
pushl  %ebp
movl   %esp, %ebp
subl   $16, %esp
```





epilogue code

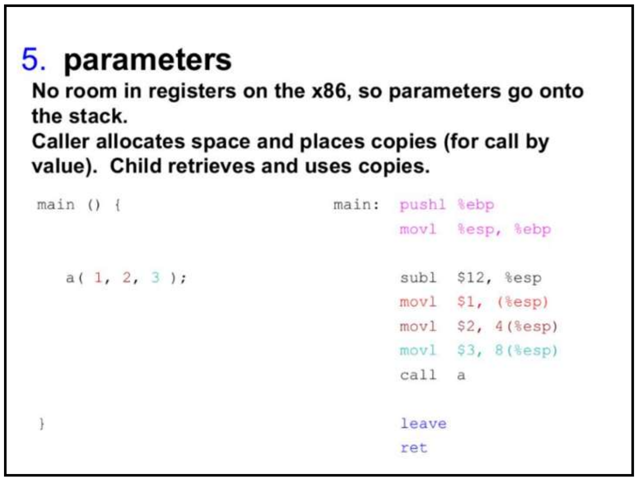
```
leave    does   movl %ebp, %esp
                popl %ebp

ret      does   popl %eip
```



4. return value

On x86, return value goes in `%eax` (by convention)

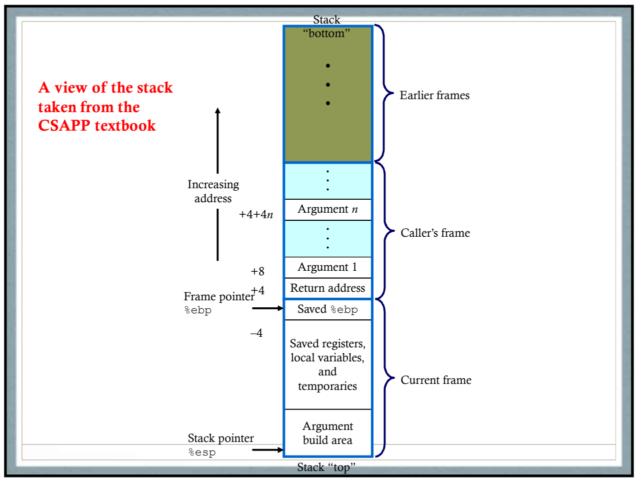
```
int b() {                b:

   c();                     call  c
   return 4;                movl  $4, %eax

}                           leave
                            ret
```



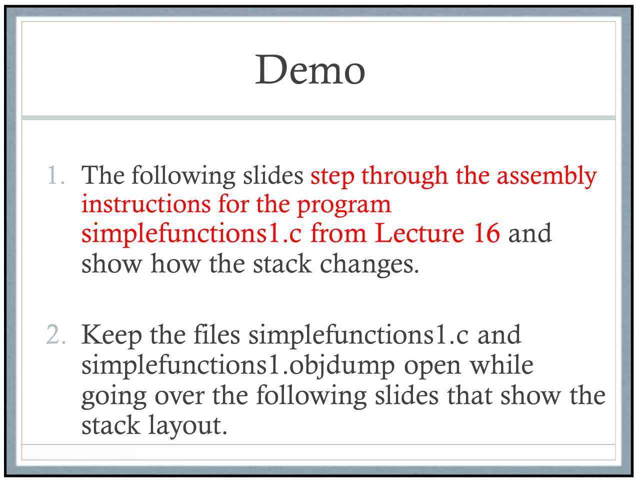
5. parameters

No room in registers on the x86, so parameters go onto the stack.
Caller allocates space and places copies (for call by value). Child retrieves and uses copies.

```
main () {                  main: pushl %ebp
                                 movl  %esp, %ebp

   a( 1, 2, 3 );                 subl  $12, %esp
                                 movl  $1, (%esp)
                                 movl  $2, 4(%esp)
                                 movl  $3, 8(%esp)
                                 call  a

}                                leave
                                 ret
```



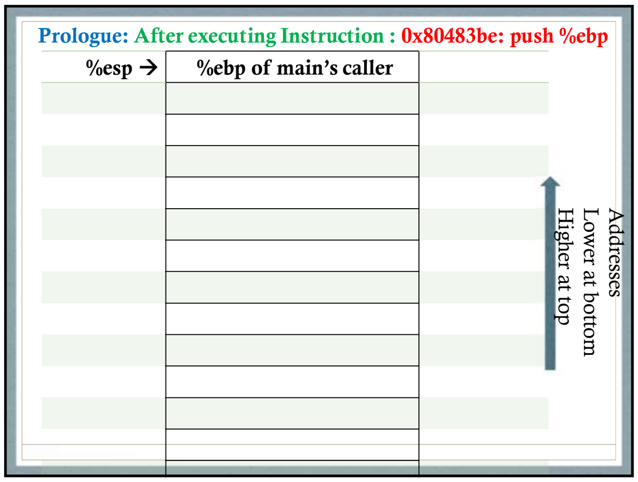
A view of the stack taken from the CSAPP textbook



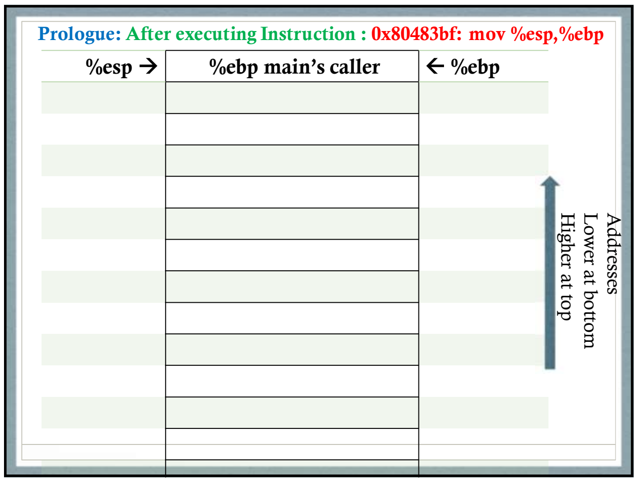
Demo

1. The following slides step through the assembly instructions for the program simplefunctions1.c from Lecture 16 and show how the stack changes.

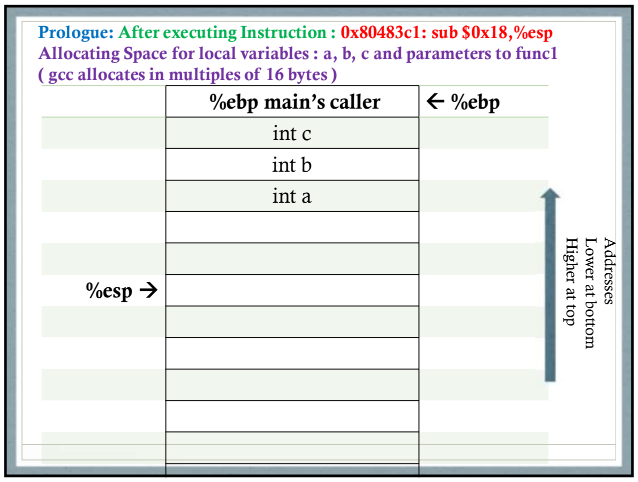
2. Keep the files simplefunctions1.c and simplefunctions1.objdump open while going over the following slides that show the stack layout.



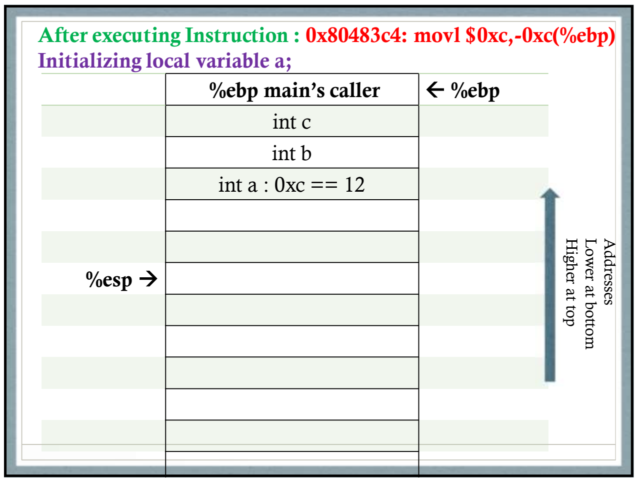
Prologue: After executing Instruction : 0x80483be: push %ebp



Prologue: After executing Instruction : 0x80483bf: mov %esp,%ebp

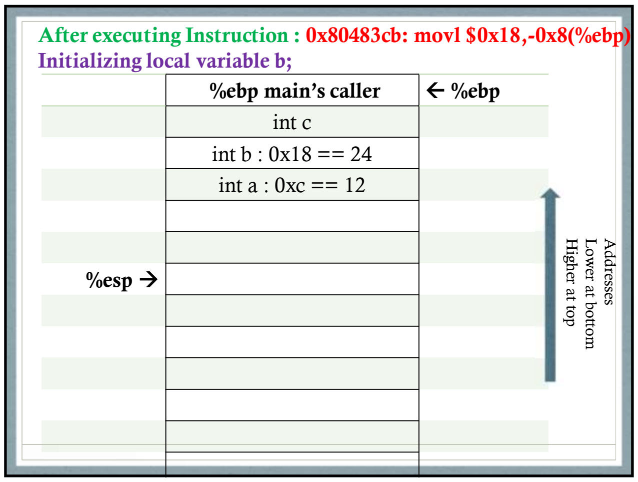


Prologue: After executing Instruction : 0x80483c1: sub $0x18,%esp
Allocating Space for local variables : a, b, c and parameters to func1
(gcc allocates in multiples of 16 bytes)



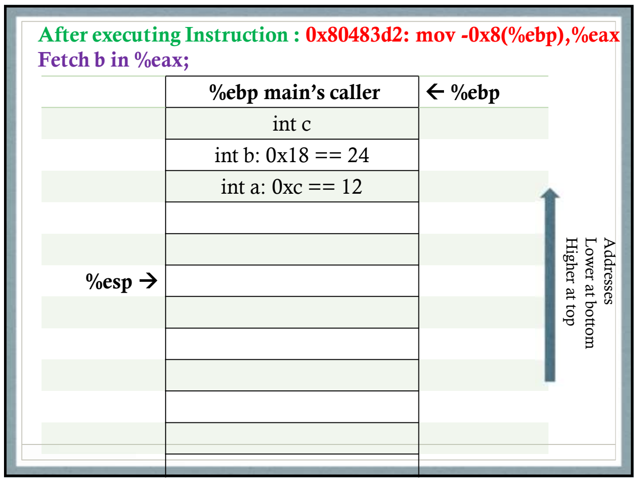
After executing Instruction : 0x80483c4: movl $0xc,-0xc(%ebp)

Initializing local variable a;



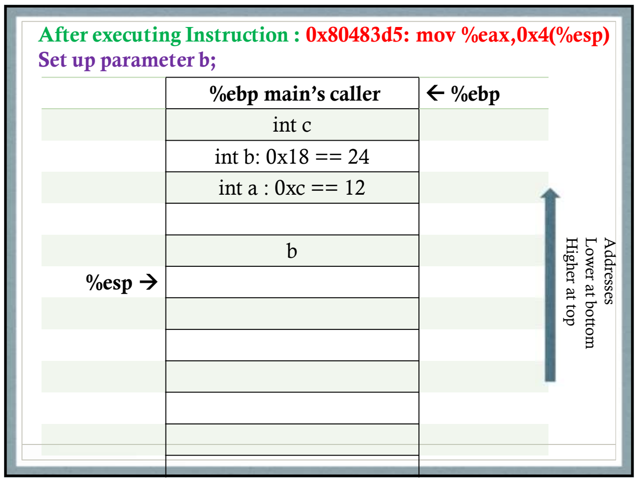
After executing Instruction : 0x80483cb: movl $0x18,-0x8(%ebp)

Initializing local variable b;



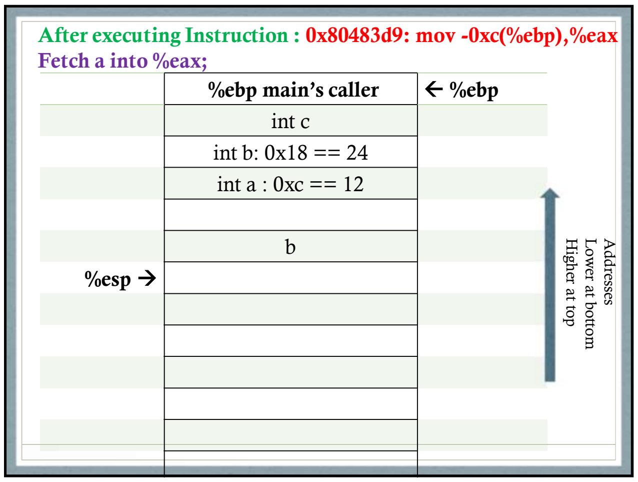
After executing Instruction : 0x80483d2: mov -0x8(%ebp),%eax

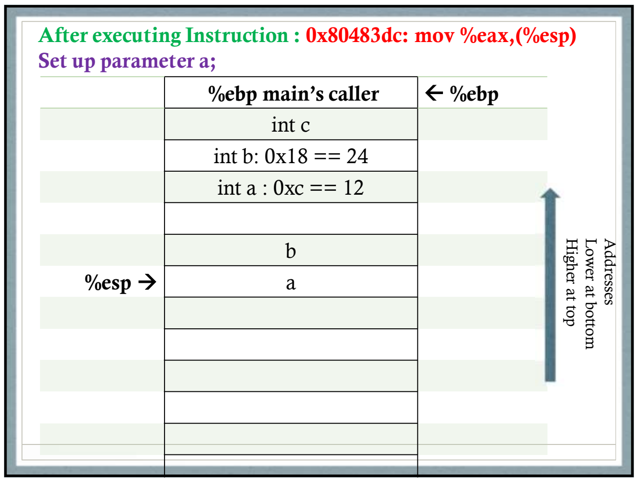
Fetch b in %eax;

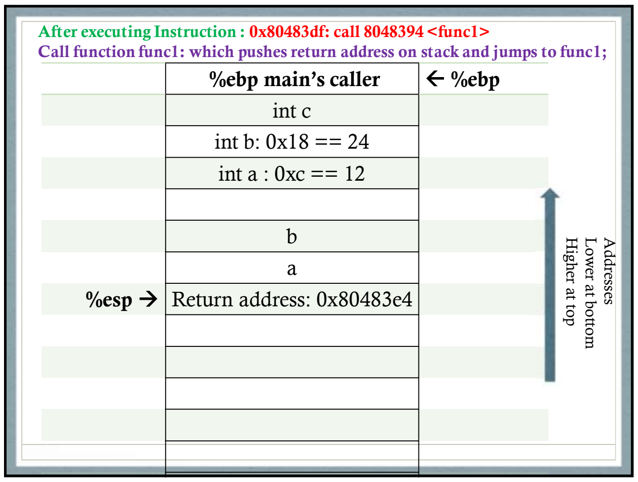


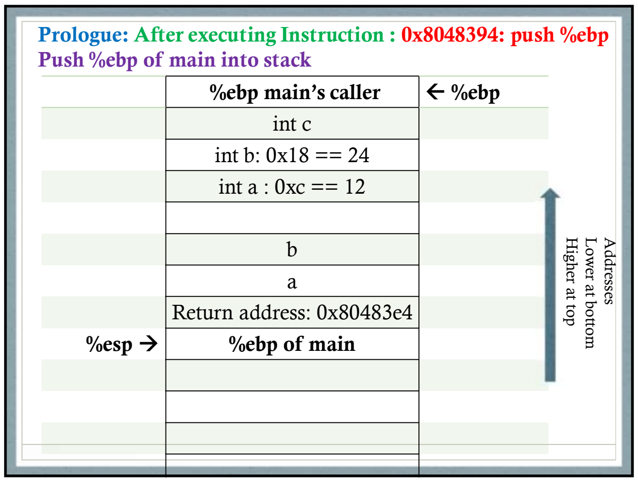
After executing Instruction : 0x80483d5: mov %eax,0x4(%esp)

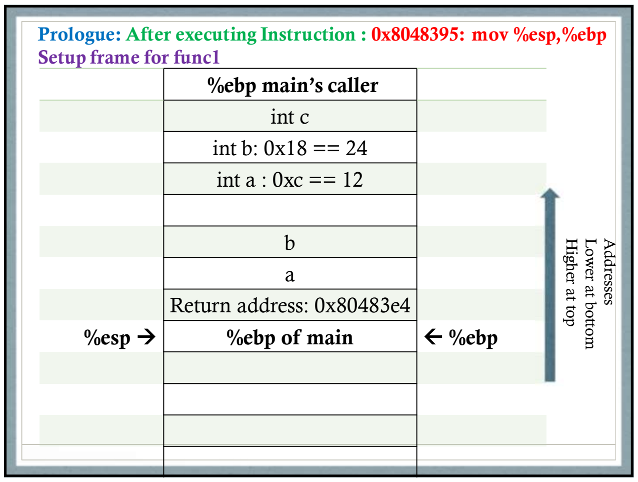
Set up parameter b;


After executing Instruction : 0x80483d9: mov -0xc(%ebp),%eax
Fetch a into %eax;


After executing Instruction : 0x80483dc: mov %eax,(%esp)
Set up parameter a;

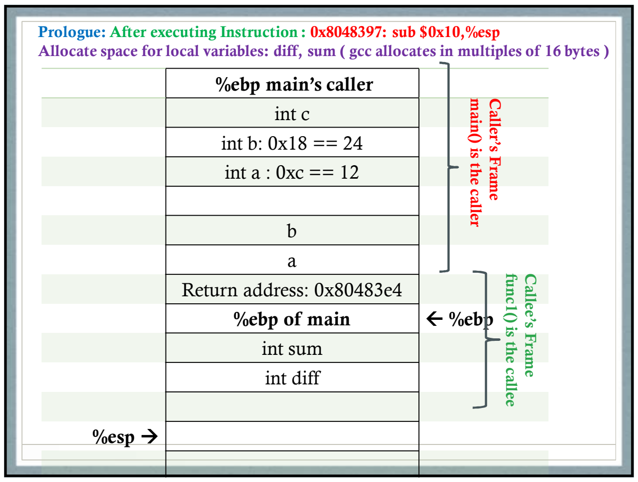

After executing Instruction : 0x80483df: call 8048394 <func1>
Call function func1: which pushes return address on stack and jumps to func1;


Prologue: After executing Instruction : 0x8048394: push %ebp
Push %ebp of main into stack



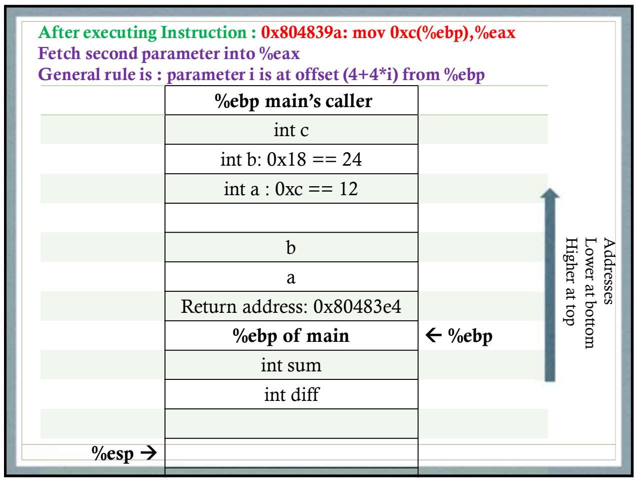
Prologue: After executing Instruction : 0x8048395: mov %esp,%ebp
Setup frame for func1

	%ebp main's caller	
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
%esp →	**%ebp of main**	**← %ebp**



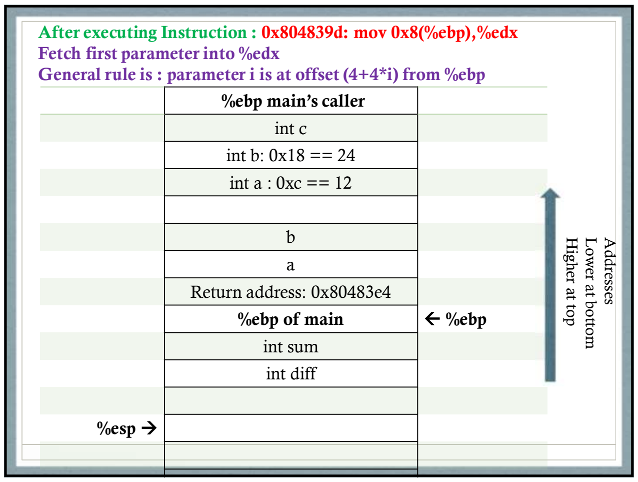
Prologue: After executing Instruction : 0x8048397: sub $0x10,%esp
Allocate space for local variables: diff, sum (gcc allocates in multiples of 16 bytes)

	%ebp main's caller	
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	**← %ebp**
	int sum	
	int diff	
%esp →		



After executing Instruction : 0x804839a: mov 0xc(%ebp),%eax
Fetch second parameter into %eax
General rule is : parameter i is at offset (4+4*i) from %ebp

	%ebp main's caller	
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	**← %ebp**
	int sum	
	int diff	
%esp →		



After executing Instruction : 0x804839d: mov 0x8(%ebp),%edx
Fetch first parameter into %edx
General rule is : parameter i is at offset (4+4*i) from %ebp

	%ebp main's caller	
	int c	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	**← %ebp**
	int sum	
	int diff	
%esp →		

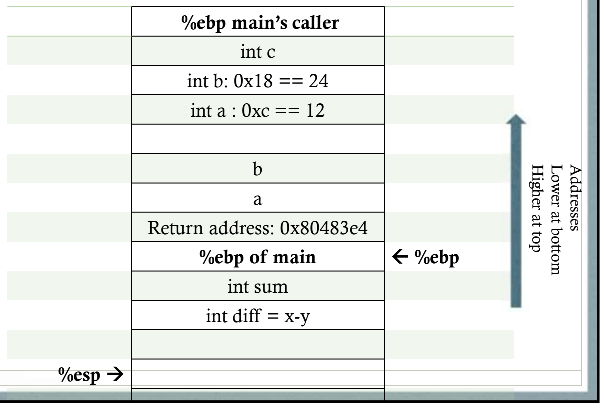
After executing Instructions :

```
0x80483a0: mov %edx,%ecx
0x80483a0: mov %edx,%ecx
0x80483a2: sub %eax,%ecx
0x80483a4: mov %ecx,%eax
```

These instruction calculate x-y and store it in %eax

After executing Instruction : 0x80483a6: mov %eax,-0x8(%ebp)

Store result in diff



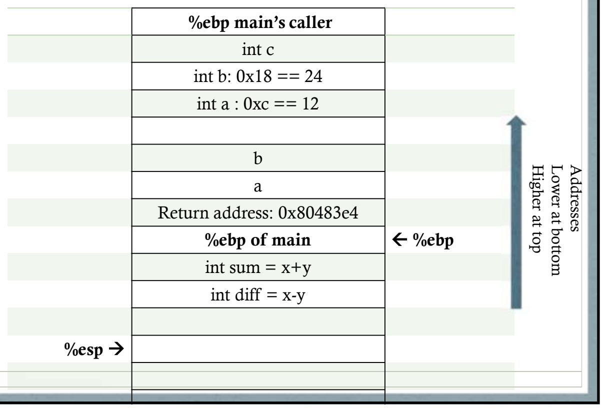
After executing Instructions :

```
0x80483a9: mov 0xc(%ebp),%eax
0x80483ac: mov 0x8(%ebp),%edx
0x80483af: lea (%edx,%eax,1),%eax
```

These instruction fetch parameters x, y into temporary registers, calculate x+y into register %eax

After executing Instruction : 0x80483b2: mov %eax,-0x4(%ebp)

Store result in sum

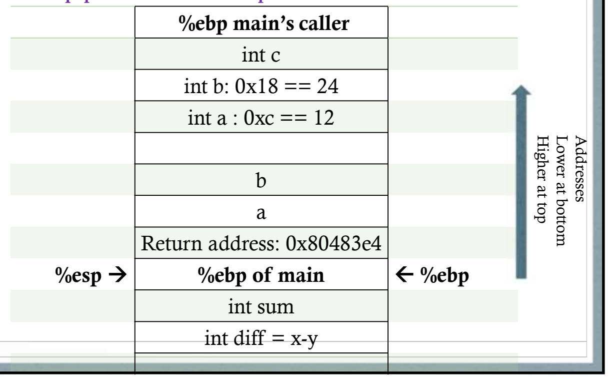


After executing Instructions :
0x80483b5: mov -0x4(%ebp),%eax
0x80483b8: imul -0x8(%ebp),%eax

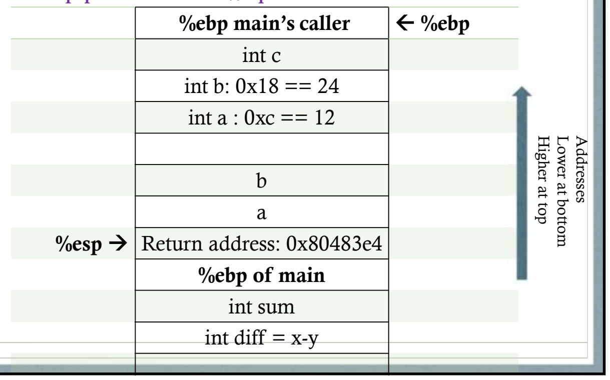
These instructions fetch sum into %eax, and then calculate product of sum and diff into register %eax

Since by x86 conventions, the result of a function is left in %eax, we do not need to anything further.

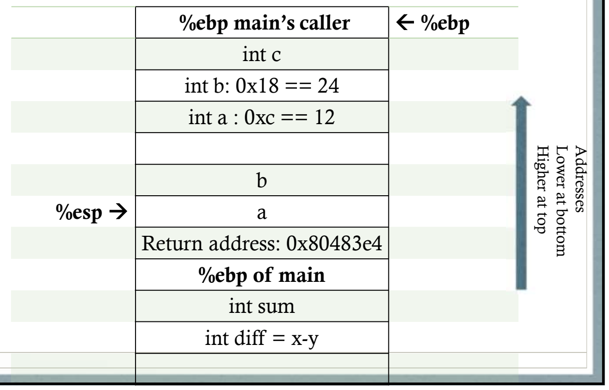
After executing First part of Instruction : 0x80483bc: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

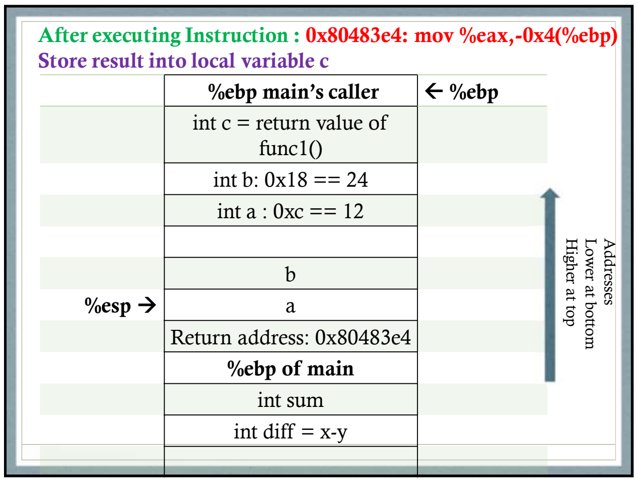


After executing Second part of Instruction : 0x80483bc: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.



After executing Instruction : 0x80483bd: ret
Return to main by poping into %eip

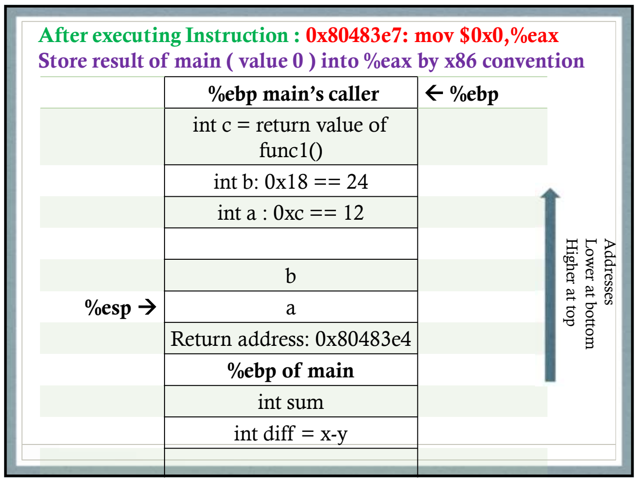




After executing Instruction : 0x80483e4: mov %eax,-0x4(%ebp)
Store result into local variable c

	%ebp main's caller	← %ebp
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
%esp →	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

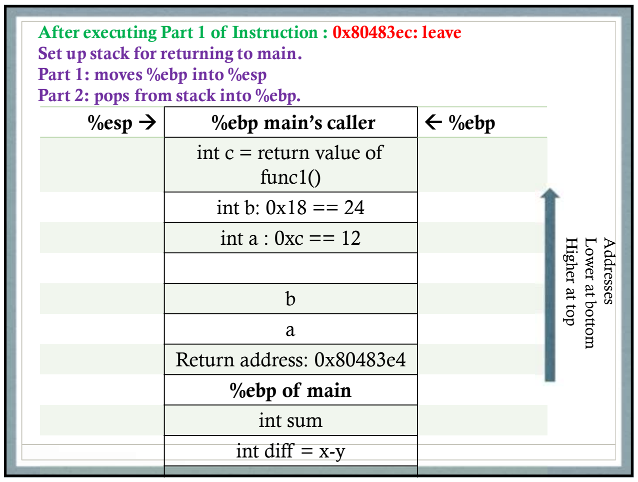
Addresses
Lower at bottom
Higher at top



After executing Instruction : 0x80483e7: mov $0x0,%eax
Store result of main (value 0) into %eax by x86 convention

	%ebp main's caller	← %ebp
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
%esp →	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

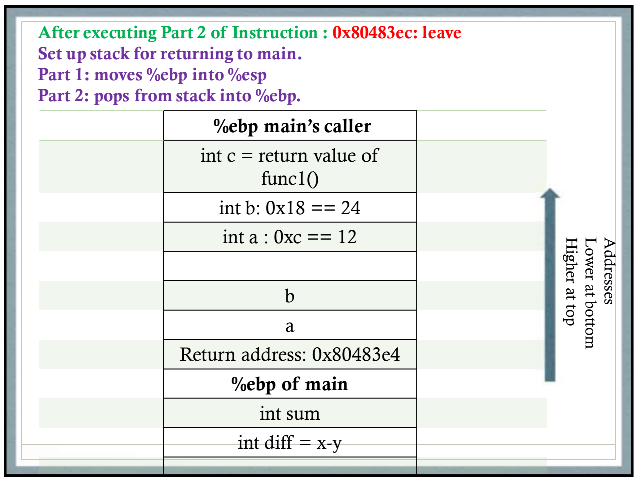
Addresses
Lower at bottom
Higher at top



After executing Part 1 of Instruction : 0x80483ec: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

%esp →	**%ebp main's caller**	← %ebp
	int c = return value of func1()	
	int b: 0x18 == 24	
	int a : 0xc == 12	
	b	
	a	
	Return address: 0x80483e4	
	%ebp of main	
	int sum	
	int diff = x-y	

Addresses
Lower at bottom
Higher at top



After executing Part 2 of Instruction : 0x80483ec: leave
Set up stack for returning to main.
Part 1: moves %ebp into %esp
Part 2: pops from stack into %ebp.

%ebp main's caller
int c = return value of func1()
int b: 0x18 == 24
int a : 0xc == 12
b
a
Return address: 0x80483e4
%ebp of main
int sum
int diff = x-y

Addresses
Lower at bottom
Higher at top

IA 32 convention

caller save

%eax %edx %ecx

callee save

%ebx %esi %edi

Which is %ebp ?

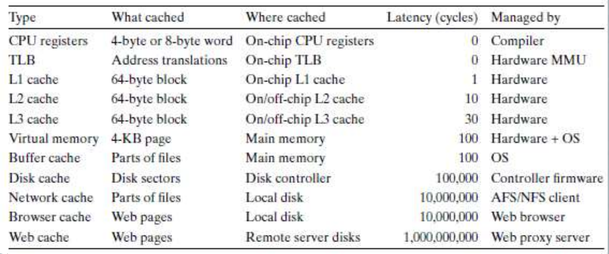
Stack Smashing

- Caused by Buffer overflow
- Attacker can store arbitrary code in the stack and execute it leading to attacks that can comprise privacy of your data or even destroy it.
- As a programmer, avoid using versions of library functions that can cause buffer overflow issues. E.g use strncpy() instead of strcpy()
- Also, check for return values of library functions and handle appropriately.

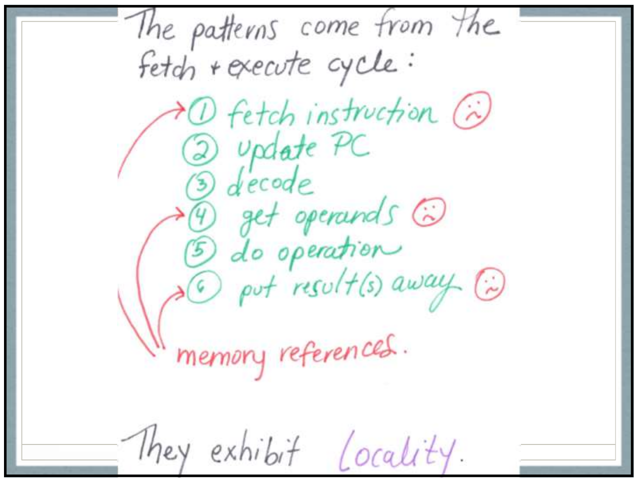
Stack Smashing

- Stack Randomization: Location of the Stack in the memory layout of the program varies between executions of a program. (Other segments can also be relocated: Address Space Layout)
- Reduce executable code locations: mark data, stack , heap as not executable.
- Canary value is inserted by compiler during function call and checked just before return to detect buffer overflow attacks.

Memory Hierarchy Table from CSAPP Textbook



Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte block	On-chip L1 cache	1	Hardware
L2 cache	64-byte block	On/off-chip L2 cache	10	Hardware
L3 cache	64-byte block	On/off-chip L3 cache	30	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

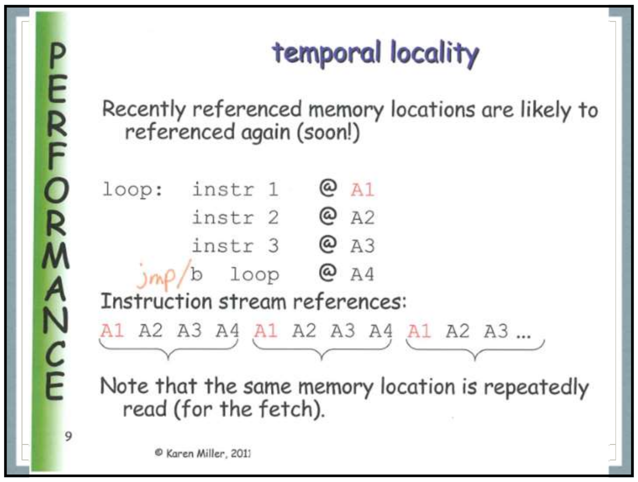


The patterns come from the fetch + execute cycle:

① fetch instruction ☹
② update PC
③ decode
④ get operands ☹
⑤ do operation
⑥ put result(s) away ☹

memory references.

They exhibit locality.



temporal locality

Recently referenced memory locations are likely to referenced again (soon!)

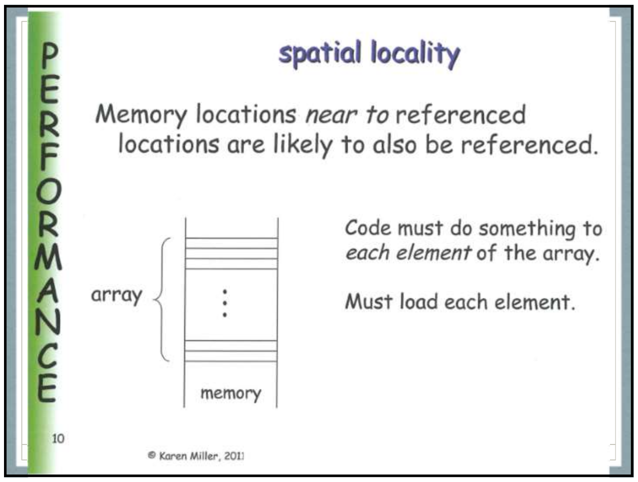
```
loop:  instr 1   @ A1
       instr 2   @ A2
       instr 3   @ A3
   jmp/b loop    @ A4
```

Instruction stream references:

$\underbrace{\text{A1 A2 A3 A4}}\ \underbrace{\text{A1 A2 A3 A4}}\ \underbrace{\text{A1 A2 A3 ...}}$

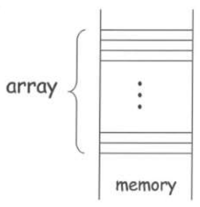
Note that the same memory location is repeatedly read (for the fetch).

9 © Karen Miller, 2011



spatial locality

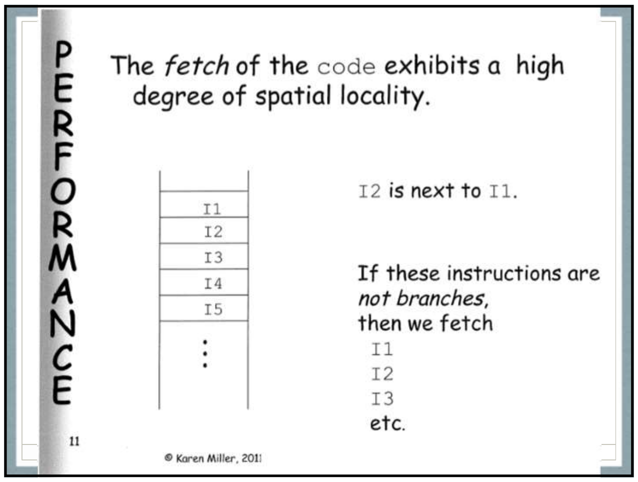
Memory locations *near to* referenced locations are likely to also be referenced.



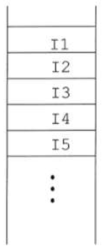
Code must do something to *each element* of the array.

Must load each element.

10 © Karen Miller, 2011



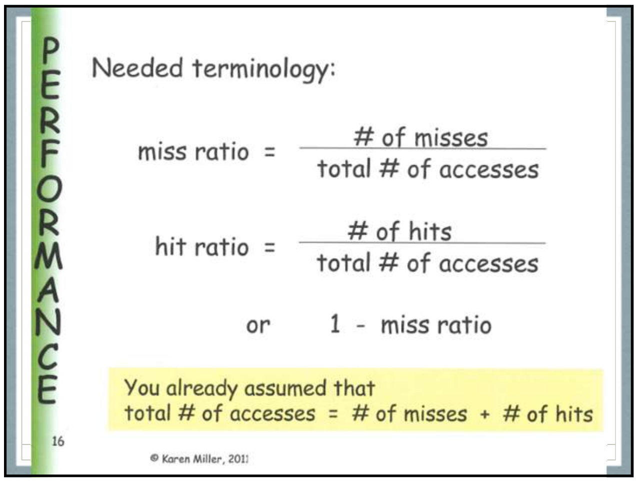
The *fetch* of the code exhibits a high degree of spatial locality.



I2 is next to I1.

If these instructions are *not branches*, then we fetch
I1
I2
I3
etc.

11 © Karen Miller, 2011



PERFORMANCE

Needed terminology:

$$\text{miss ratio} = \frac{\text{\# of misses}}{\text{total \# of accesses}}$$

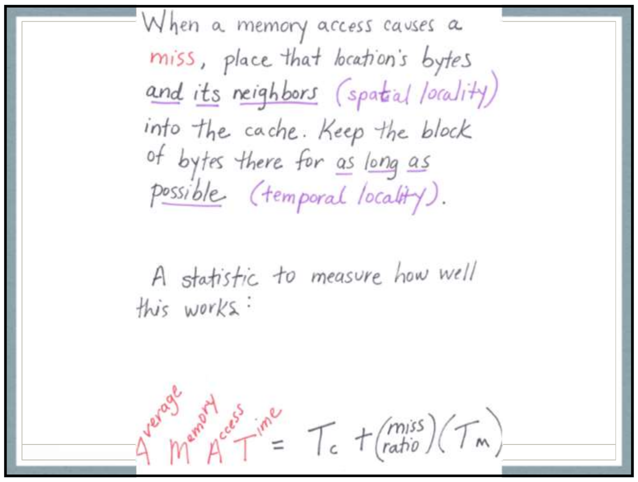
$$\text{hit ratio} = \frac{\text{\# of hits}}{\text{total \# of accesses}}$$

or 1 - miss ratio

You already assumed that
total # of accesses = # of misses + # of hits

16

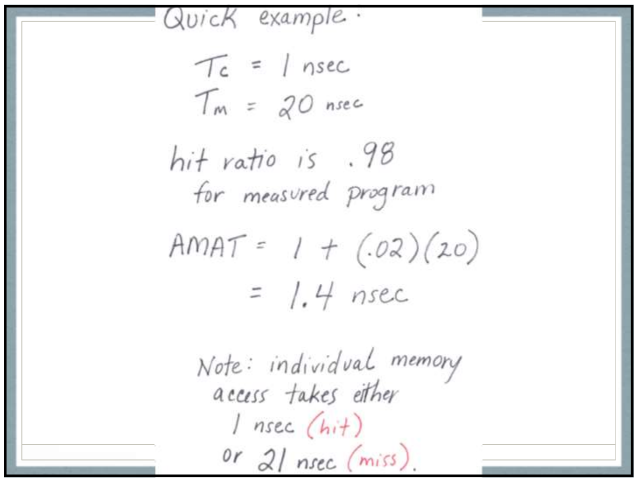
© Karen Miller, 2011



When a memory access causes a miss, place that location's bytes and its neighbors (spatial locality) into the cache. Keep the block of bytes there for as long as possible (temporal locality).

A statistic to measure how well this works:

$$\text{Average Memory Access Time} = T_c + \binom{\text{miss}}{\text{ratio}}(T_m)$$



Quick example:

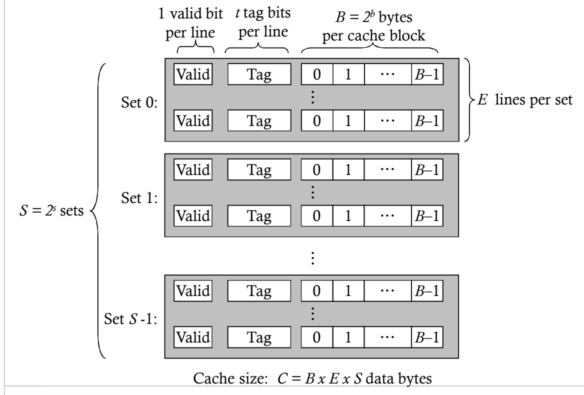
$T_c = 1$ nsec
$T_m = 20$ nsec

hit ratio is .98 for measured program

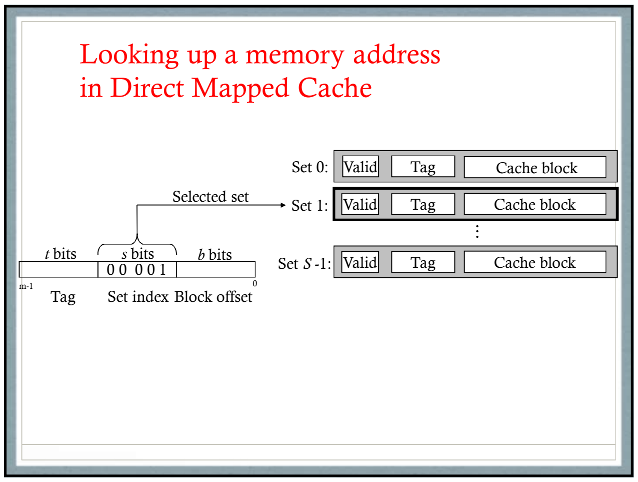
$AMAT = 1 + (.02)(20)$
$= 1.4$ nsec

Note: individual memory access takes either 1 nsec (hit) or 21 nsec (miss).

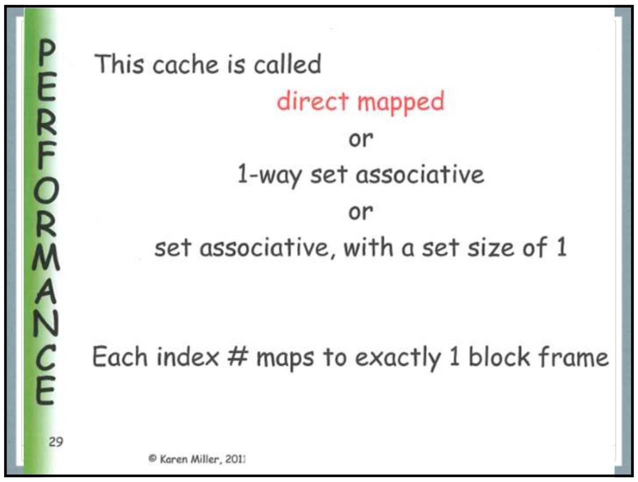
Generic Cache Organization



Cache size: $C = B \times E \times S$ data bytes



Looking up a memory address in Direct Mapped Cache



PERFORMANCE

This cache is called

direct mapped

or

1-way set associative

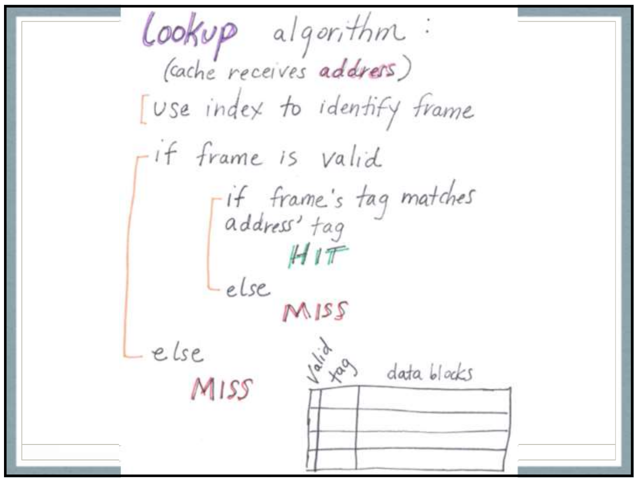
or

set associative, with a set size of 1

Each index # maps to exactly 1 block frame

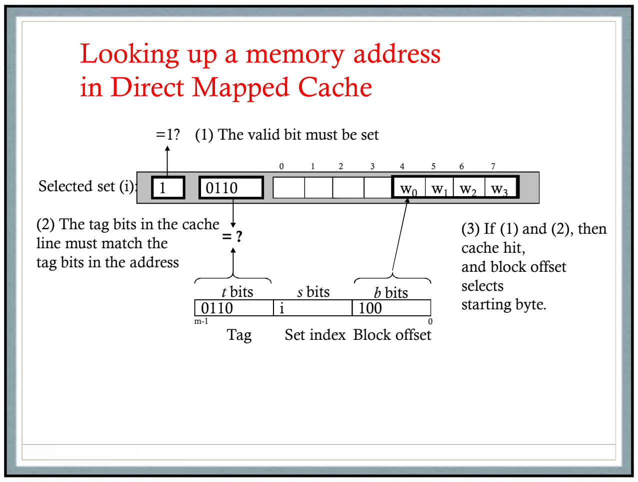
29

© Karen Miller, 2011



```
Lookup algorithm :
(cache receives address)
[use index to identify frame
if frame is valid
    if frame's tag matches
    address' tag
        HIT
    else
        MISS
else
    MISS
```

Valid
tag
data blocks

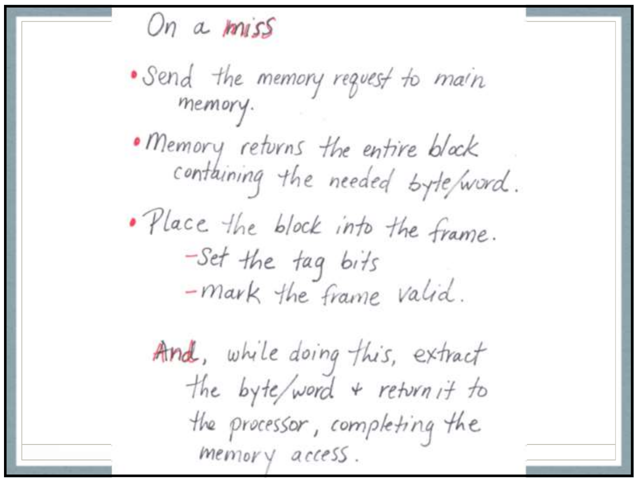


Looking up a memory address in Direct Mapped Cache

=1? (1) The valid bit must be set

(2) The tag bits in the cache line must match the tag bits in the address

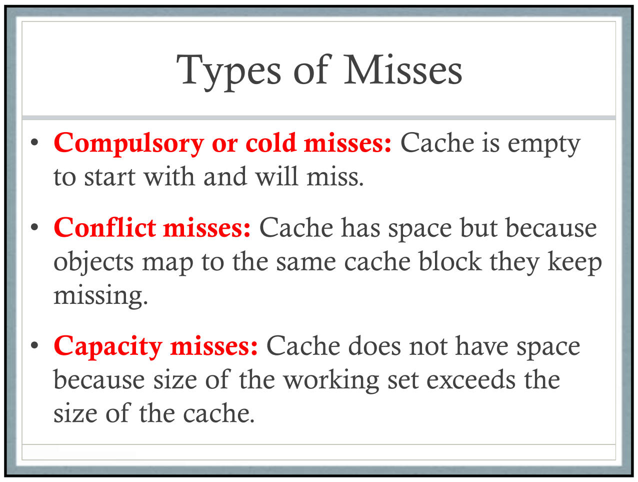
(3) If (1) and (2), then cache hit, and block offset selects starting byte.



On a miss

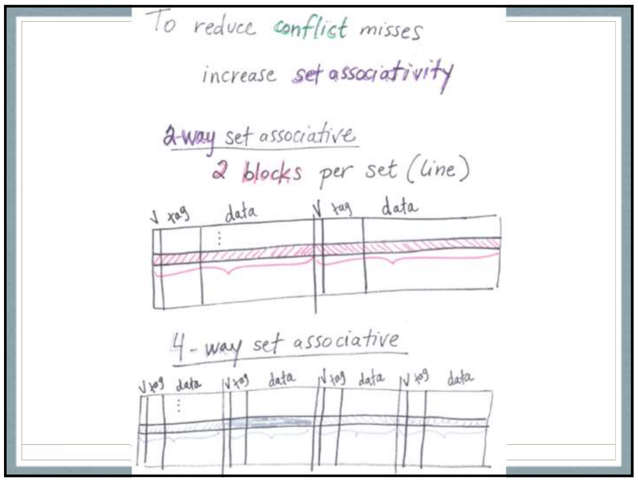
- Send the memory request to main memory.
- Memory returns the entire block containing the needed byte/word.
- Place the block into the frame.
  - Set the tag bits
  - mark the frame valid.

And, while doing this, extract the byte/word + return it to the processor, completing the memory access.



Types of Misses

- **Compulsory or cold misses:** Cache is empty to start with and will miss.
- **Conflict misses:** Cache has space but because objects map to the same cache block they keep missing.
- **Capacity misses:** Cache does not have space because size of the working set exceeds the size of the cache.



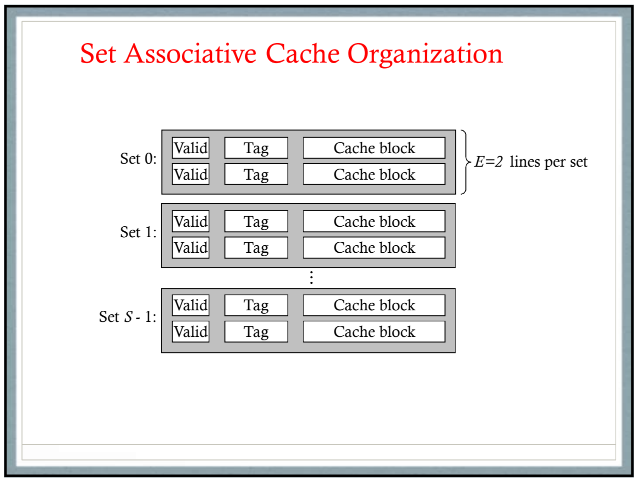
To reduce conflict misses

increase set associativity

2-way set associative

2 blocks per set (line)

4-way set associative



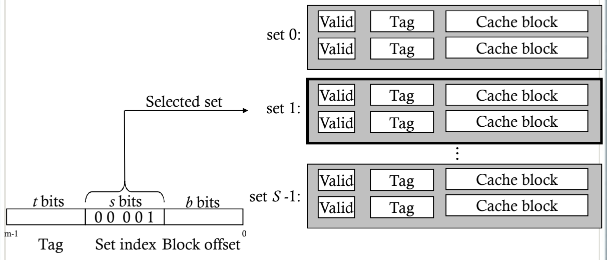
Set Associative Cache Organization

larger set size

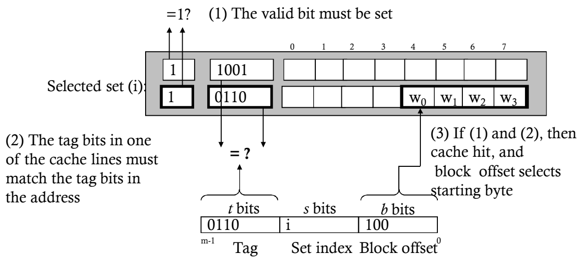
☺ tends to lead to higher hit ratio (due to fewer conflic misses)

☹ amount of circuitry goes up, leading to increase in T_c

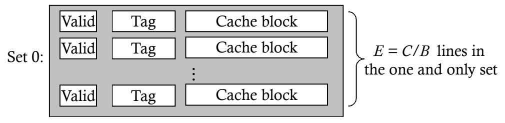
Looking up a memory address in Set Associative Cache



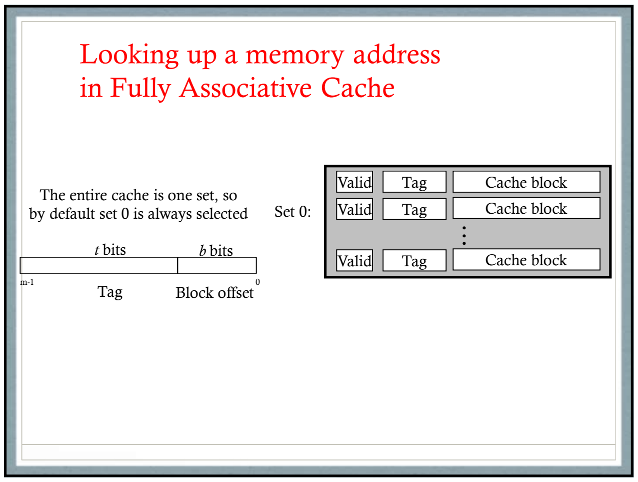
Looking up a memory address in Set Associative Cache


=1?
(1) The valid bit must be set
0 1 2 3 4 5 6 7
Selected set (i):
1
1001
0110
w_0 w_1 w_2 w_3
= ?
(2) The tag bits in one of the cache lines must match the tag bits in the address
(3) If (1) and (2), then cache hit, and block offset selects starting byte
t bits
s bits
b bits
0110
i
100
m-1
0
Tag
Set index
Block offset

Fully Associative Cache Organization

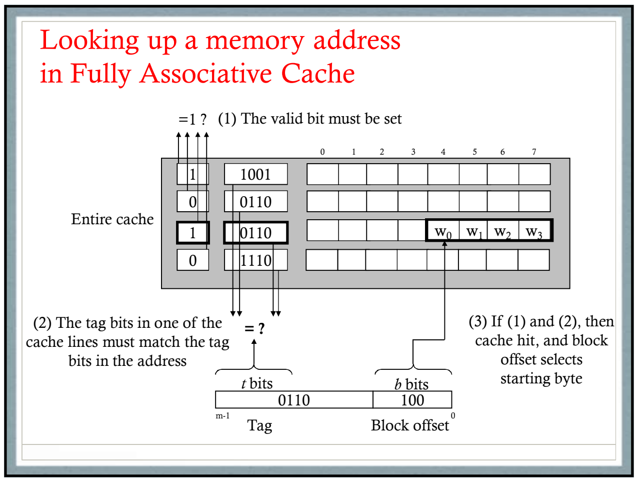


More circuitry and hence more expensive than Direct mapped and Set Associative Caches



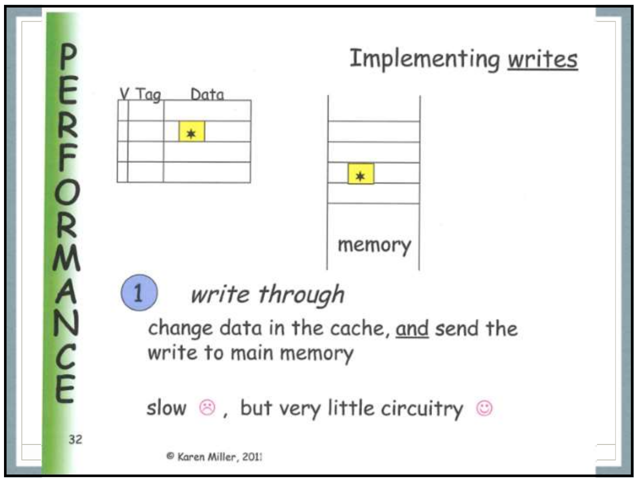
Looking up a memory address in Fully Associative Cache

The entire cache is one set, so by default set 0 is always selected



Looking up a memory address in Fully Associative Cache

=1 ?
(1) The valid bit must be set
Entire cache
0 1 2 3 4 5 6 7
1
1001
0
0110
1
0110
w_0 w_1 w_2 w_3
0
1110
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte
t bits
b bits
0110
100
m-1
0
Tag
Block offset

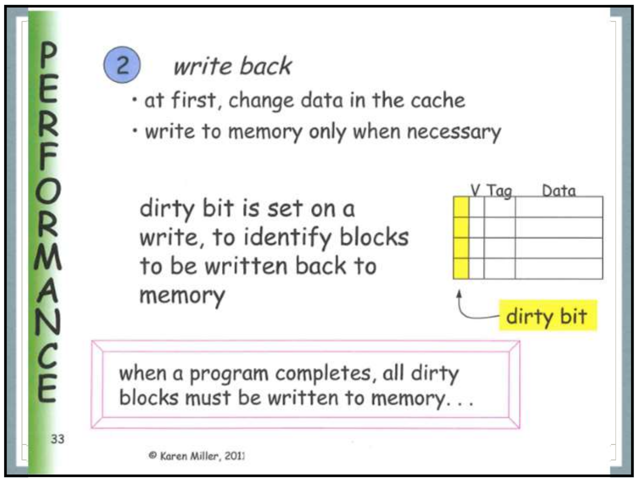


Implementing writes

1 *write through*

change data in the cache, and send the write to main memory

slow ☹ , but very little circuitry ☺

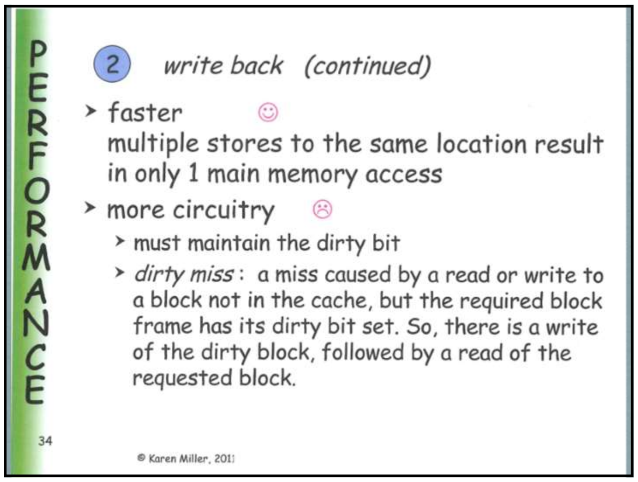


2 *write back*

- at first, change data in the cache
- write to memory only when necessary

dirty bit is set on a write, to identify blocks to be written back to memory

when a program completes, all dirty blocks must be written to memory. . .



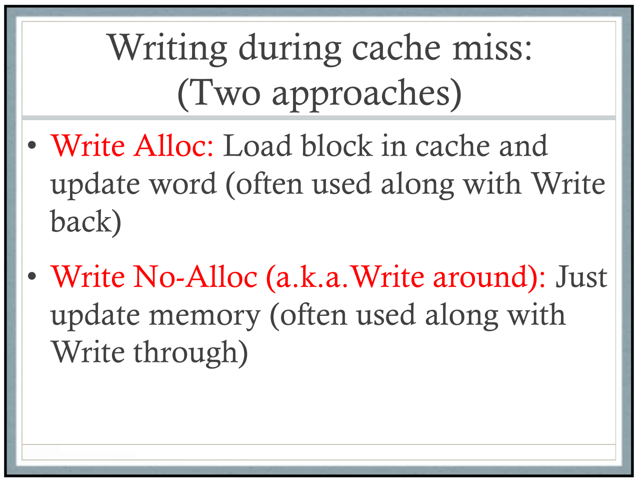
PERFORMANCE

② *write back* *(continued)*

- faster ☺
  multiple stores to the same location result in only 1 main memory access
- more circuitry ☹
  - must maintain the dirty bit
  - *dirty miss* : a miss caused by a read or write to a block not in the cache, but the required block frame has its dirty bit set. So, there is a write of the dirty block, followed by a read of the requested block.

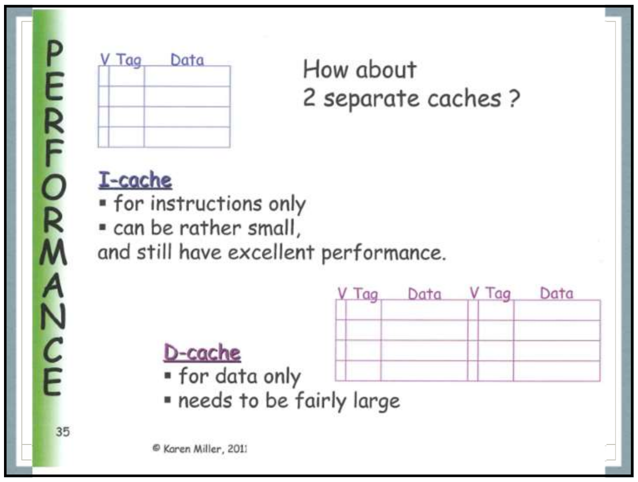
34

© Karen Miller, 2011



Writing during cache miss: (Two approaches)

- Write Alloc: Load block in cache and update word (often used along with Write back)
- Write No-Alloc (a.k.a.Write around): Just update memory (often used along with Write through)



PERFORMANCE

How about
2 separate caches ?

V Tag Data
V Tag Data V Tag Data

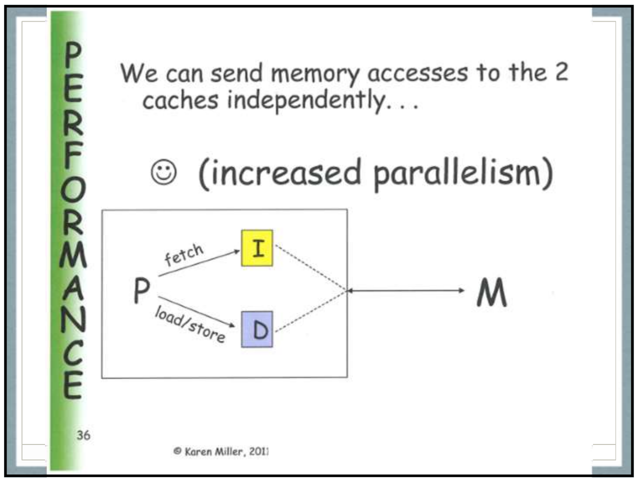
I-cache
- for instructions only
- can be rather small,

and still have excellent performance.

D-cache
- for data only
- needs to be fairly large

35

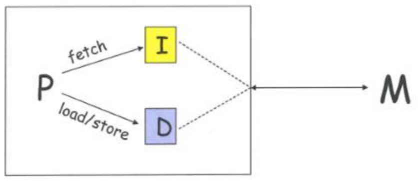
© Karen Miller, 2011



PERFORMANCE

We can send memory accesses to the 2 caches independently. . .

☺ (increased parallelism)



36

© Karen Miller, 2011

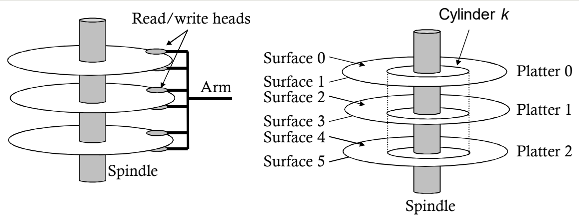
Strided Access Patterns

```
int i, j, sum =0;
for(i=0;i<16;i++)
    for(j=0;j<16;j++)
        sum += a[i][j]
```

What if: sum += a[j][i] ?

Writing Cache Friendly Code

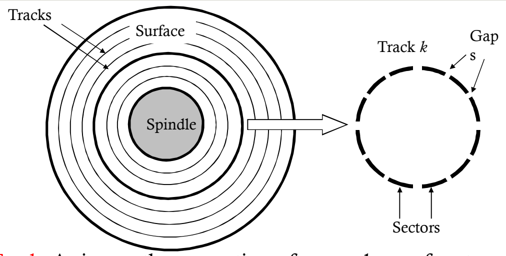
1. Focus on the inner loops where bulk of computation and memory accesses occur
2. Maximize spatial locality by reading data objects sequentially with stride 1
3. Maximize temporal locality by reading a data object as often as possible once it has been read from memory.



Hard Disk Overview

Platters: Consists of two sides or surfaces coated with magnetic recording material

Spindle: Spins the platter at fixed rotational rate (5400 to 15K revolutions per minute - RPM)



Track: A ring on the magnetic surface made up of sectors.

Sectors: Each track is partitioned into a collection of Sectors. Gaps in between sectors store formatting bits that identify sectors.

Cylinder: Collection of tracks on all surfaces that are equidistant from the center of the spindle.

Access Time for a Sector

$T_{I/O} = T_{seek} + T_{rot} + T_{xfr}$

Seek Time: Time taken to position the head over the track that contains the target sector.

Rotational Latency: Time taken for the first bit of the target sector to pass under the head.

Transfer time: Time taken to read or write the contents once the first bit of the target sector is under the head.

Accessing Disks

Memory mapped I/O: A block of addresses in the address space is reserved for communicating with the I/O devices.

Each of these addresses is called an I/O Port.

Every device attached to a bus will have a corresponding I/O Port.

Small Communication with the I/O devices happens through the I/O port.

Textbook Example

Suppose magnetic disk is mapped to I/O port 0xa0

CPU initiates disk read by 3 store instructions:

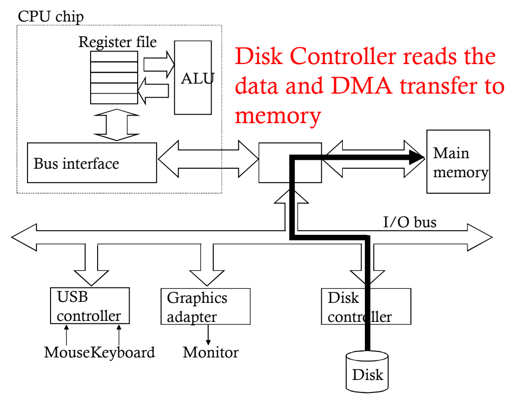
1) Send a command word indicating to perform a read along with other parameters like interrupt on completion of DMA.
2) Send the logical block number that must be read.
3) Indicate the main memory address where the contents of the disk sector should be stored.

Direct Memory Access

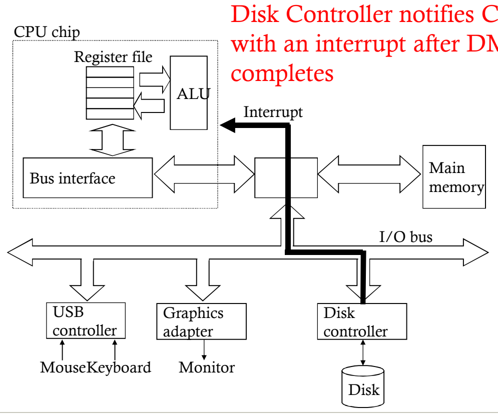
After issuing the request, CPU executes other instructions. Note: A 3GHz processor with a 0.33 ns clock cycle can execute 9 million instructions in 3ms.

After receiving the read command from the CPU, the disk controller fetches data from the right sector and transfers it directly to the memory without the involvement of the CPU.

After DMA is complete, disk controller notifies by sending interrupt signal to the CPU.



Disk Controller reads the data and DMA transfer to memory

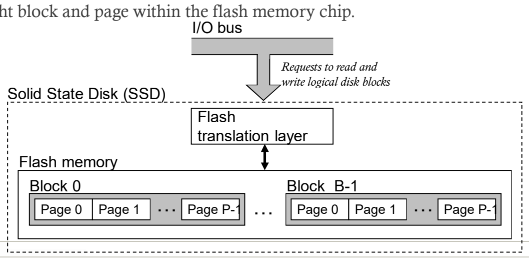


Disk Controller notifies CPU with an interrupt after DMA completes

Solid State Disks

Made up of flash memory chips that store data instead of the magnetic surfaces in a conventional disk.

A Flash Translation Layer translates logical block addresses to accesses to the right block and page within the flash memory chip.



SSD vs. Magnetic Disks

Advantages: SSDs have no moving parts and hence are more quiet during operation, faster, need less power and are more rugged.

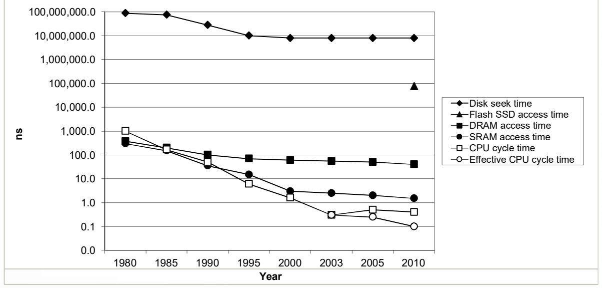
Disadvantages:

1. Possibility of wear out after several program-erase cycles. This is mitigated by the flash translation layer.

2. More expensive than magnetic disks.

Storage Trends

DRAM and disk performance are lagging behind CPU performance. Though SRAM performance also lags, it is roughly keeping up.



Unix I/O

Unix file is a sequence of bytes.

All I/O devices including networks, terminals, disks are modeled as files. E.g. /dev/sda for disk, /dev/tty for terminal

This way, programmers use a simple low-level interface called the Unix I/O to interact with all I/O devices

Some examples of operations on files:

- Opening and Closing files
- Reading and writing files
- Changing the current file offset
- Read metadata about a file

Unix I/O APIs

int open(char *filename, int flags, mode_t mode);

ssize_t read(int fd, void *buf, size_t n);

ssize_t write(int fd, const void *buf, size_t n);

off_t lseek(int fd, off_t offset, int whence);

int fstat(int fd, struct stat *buf);

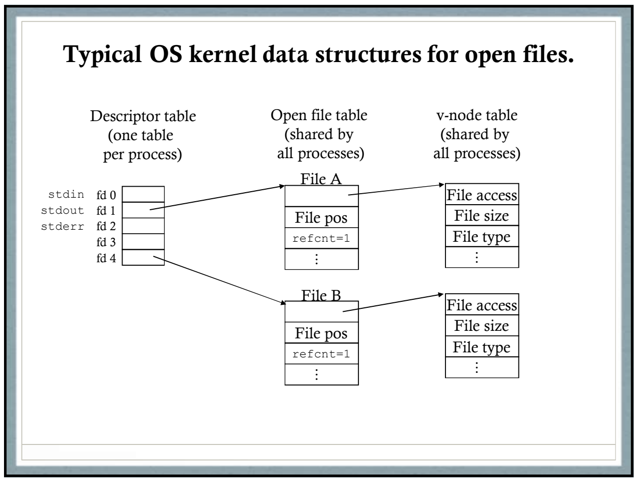
Each process created by a Unix shell begins life with three open files: standard input (descriptor 0), standard output (descriptor 1), and standard error (descriptor 2)

Sharing Files
(Descriptions from CSAPP Text)

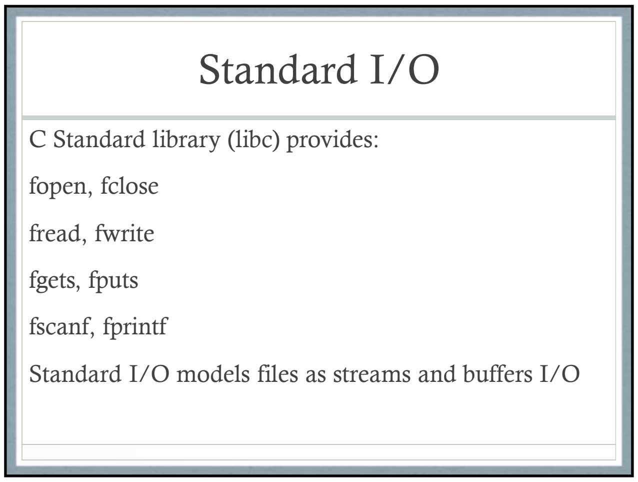
Descriptor table: Each process has its own separate descriptor table whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the file table.

File table: The set of open files is represented by a file table that is shared by all processes. Each file table entry consists of (for our purposes) the current file position, a reference count of the number of descriptor entries that currently point to it, and a pointer to an entry in the v-node table. Closing a descriptor decrements the reference count in the associated file table entry. The kernel will not delete the file table entry until its reference count is zero.

v-node table: Like the file table, the v-node table is shared by all processes. Each entry contains most of the information in the stat structure, including the st_mode and st_size members. v-node table and the related VFS(Virtual File System) interface is the separation between specific file system implementations and the generic file system operations.



Typical OS kernel data structures for open files.



Standard I/O

C Standard library (libc) provides:

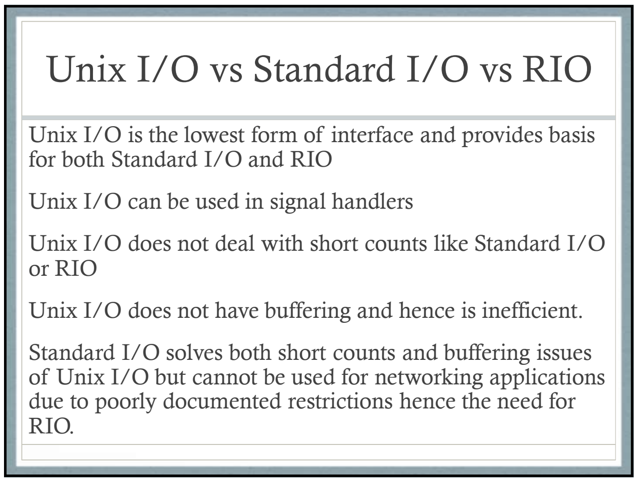
fopen, fclose

fread, fwrite

fgets, fputs

fscanf, fprintf

Standard I/O models files as streams and buffers I/O



Unix I/O vs Standard I/O vs RIO

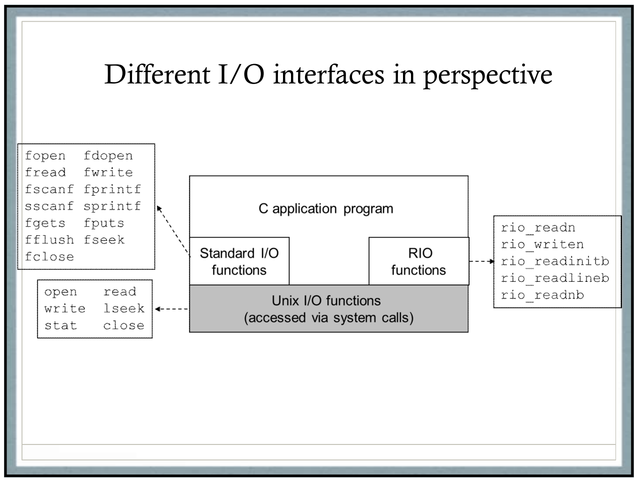
Unix I/O is the lowest form of interface and provides basis for both Standard I/O and RIO

Unix I/O can be used in signal handlers

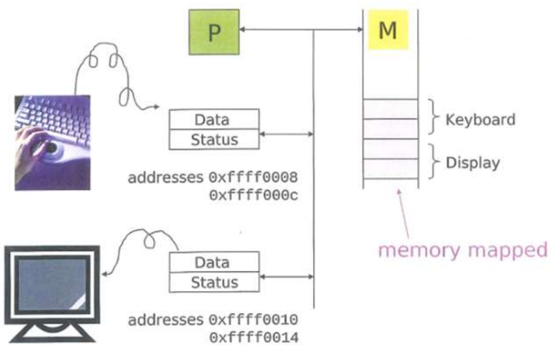
Unix I/O does not deal with short counts like Standard I/O or RIO

Unix I/O does not have buffering and hence is inefficient.

Standard I/O solves both short counts and buffering issues of Unix I/O but cannot be used for networking applications due to poorly documented restrictions hence the need for RIO.

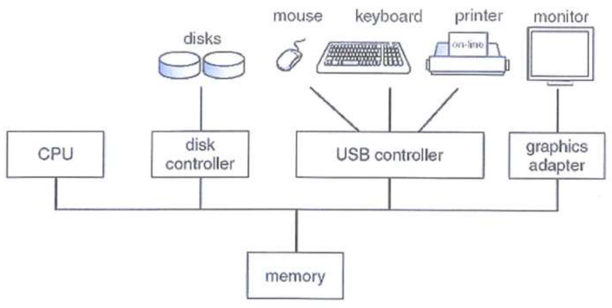


Different I/O interfaces in perspective



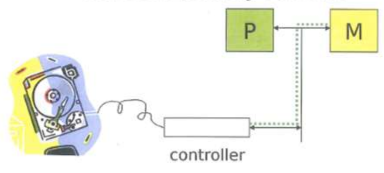
Now, driver code uses a
spin wait loop
(to implement blocking I/o)

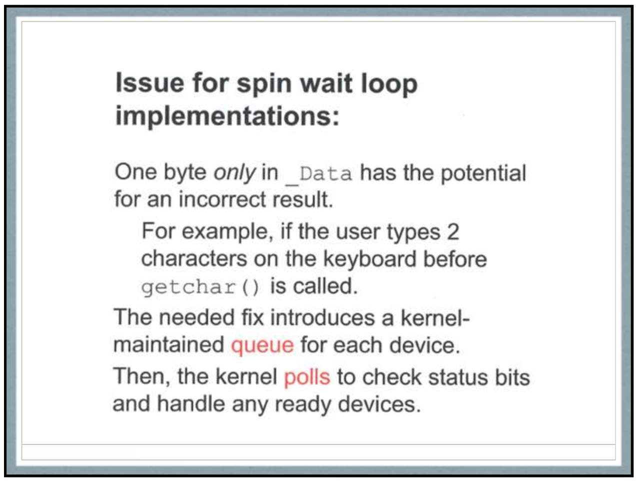
```
kb_spin:   testl  Keyboard_Status, Keyboard_Status
           jz     kb-spin
           movl   Keyboard-Data, %eax
           retfrom~~syscall~~int

disp-spin: testl  Display_Status, Display-Status
           jz     disp-spin
           movl   %eax, Display.Data
           retfrom~~syscall~~int
```



Byte transfers are OK,
But, what about faster devices that like to transfer more than a byte ?

the solution: DMA
Direct Memory Access





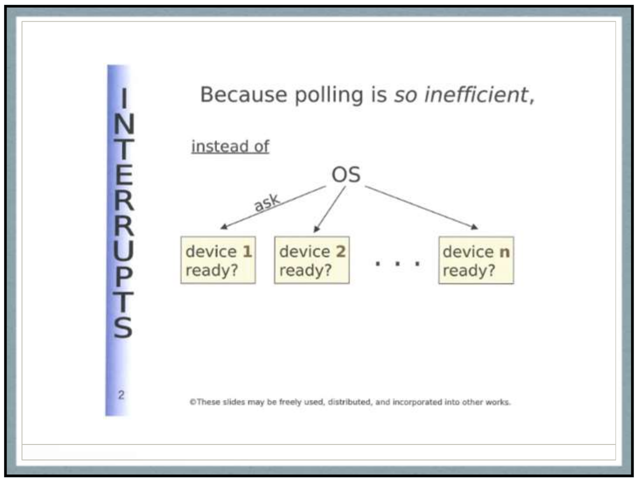
Issue for spin wait loop implementations:

One byte *only* in `_Data` has the potential for an incorrect result.

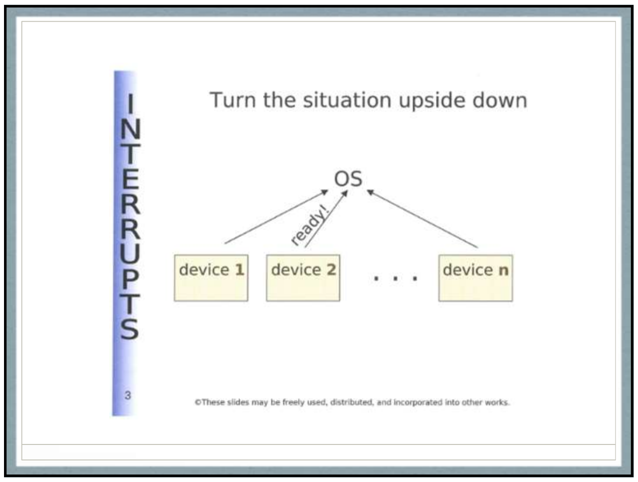
For example, if the user types 2 characters on the keyboard before `getchar()` is called.

The needed fix introduces a kernel-maintained queue for each device.

Then, the kernel polls to check status bits and handle any ready devices.



Because polling is *so inefficient*,

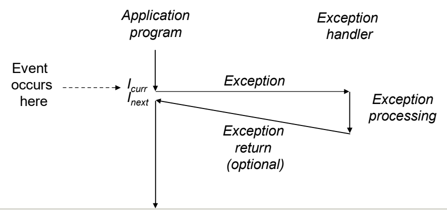


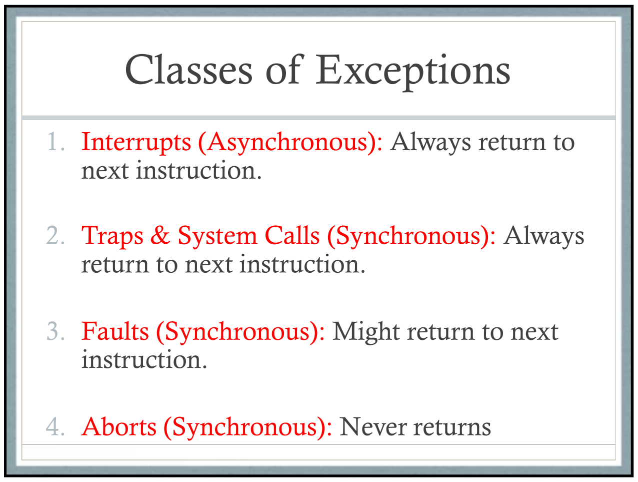
Turn the situation upside down

Anatomy of an exception

An exceptions is an abrupt change in control flow.

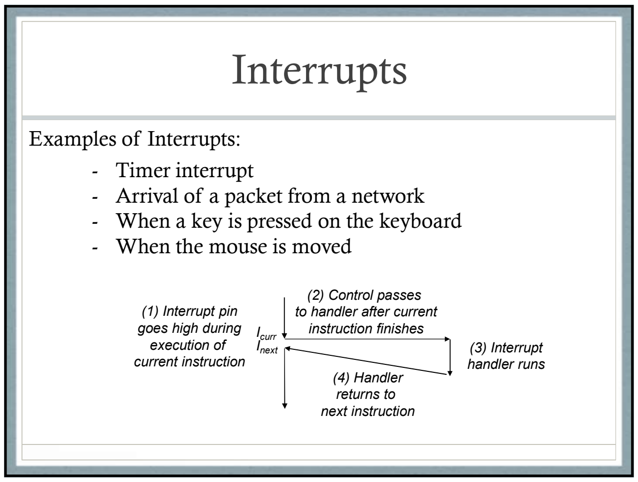
Examples: div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C





Classes of Exceptions

1. Interrupts (Asynchronous): Always return to next instruction.
2. Traps & System Calls (Synchronous): Always return to next instruction.
3. Faults (Synchronous): Might return to next instruction.
4. Aborts (Synchronous): Never returns

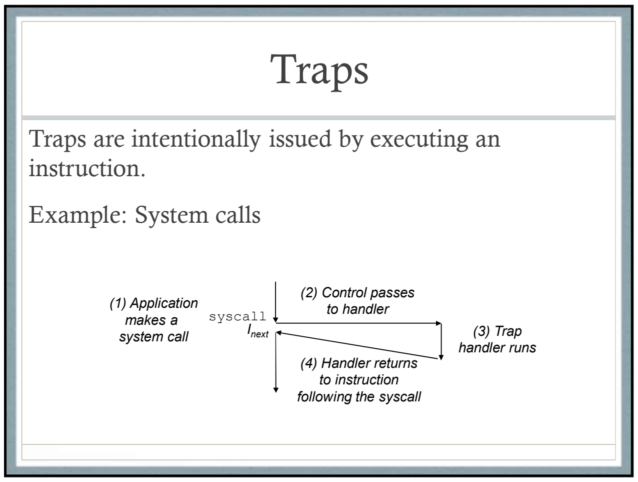


Interrupts

Examples of Interrupts:

- Timer interrupt
- Arrival of a packet from a network
- When a key is pressed on the keyboard
- When the mouse is moved

(1) Interrupt pin goes high during execution of current instruction
I_{curr}
I_{next}
(2) Control passes to handler after current instruction finishes
(3) Interrupt handler runs
(4) Handler returns to next instruction

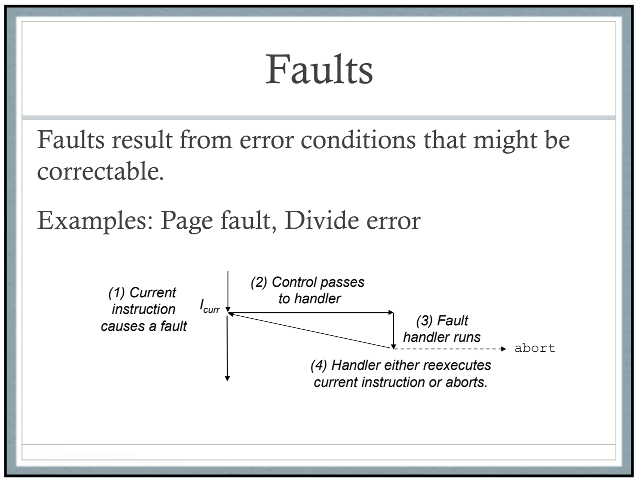


Traps

Traps are intentionally issued by executing an instruction.

Example: System calls

(1) Application makes a system call
syscall
I_{next}
(2) Control passes to handler
(3) Trap handler runs
(4) Handler returns to instruction following the syscall



Faults

Faults result from error conditions that might be correctable.

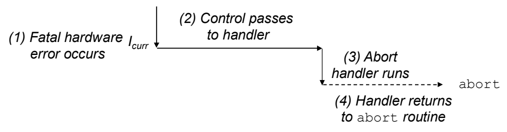
Examples: Page fault, Divide error

(1) Current instruction causes a fault
I_{curr}
(2) Control passes to handler
(3) Fault handler runs
abort
(4) Handler either reexecutes current instruction or aborts.

Aborts

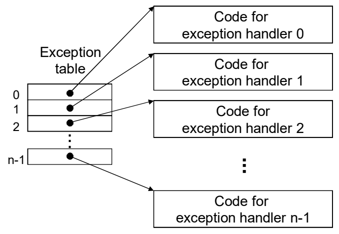
Aborts result from unrecoverable fatal errors.

Example: parity errors due to DRAM bit corruption


(1) Fatal hardware error occurs
I_{curr}
(2) Control passes to handler
(3) Abort handler runs
abort
(4) Handler returns to abort routine

Exception Table

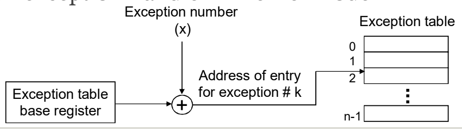
Exception table


Exception table
0
1
2
n-1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
Code for exception handler n-1

Exception Table lookup

Exception is similar to procedure calls except for some important differences:

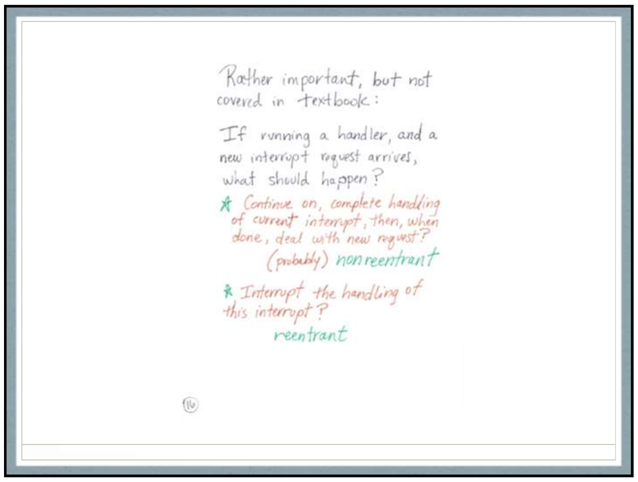
- Return address is not the next instruction always
- Push EFLAGS register also onto kernel stack
- Run exception handler in kernel mode


Exception number (x)
Exception table
0
1
2
n-1
Address of entry for exception # k
Exception table base register

IA32 Exception Table

From CSAPP text book:

Exception Number	Description	Exception Class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-127	OS-defined	Interrupt or trap
128 (0x80)	System call	Trap
129-255	OS-defined	Interrupt or trap



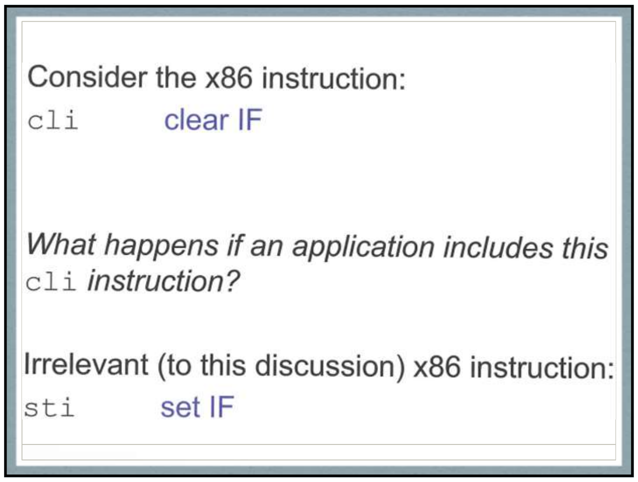
Rather important, but not covered in textbook:

If running a handler, and a new interrupt request arrives, what should happen?

* Continue on, complete handling of current interrupt, then, when done, deal with new request?
  (probably) nonreentrant

* Interrupt the handling of this interrupt?
  reentrant

(16)



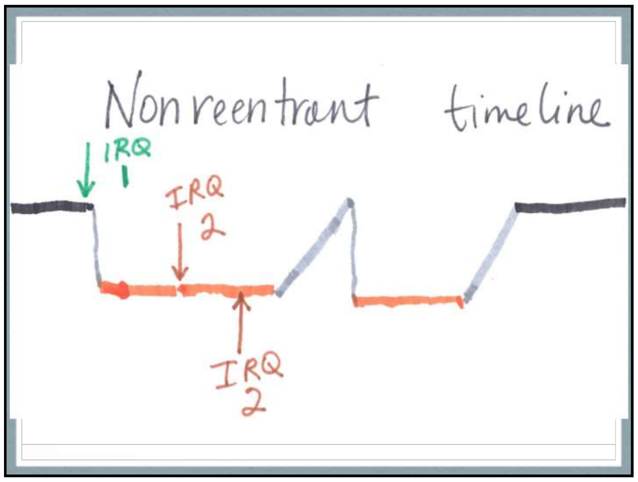
Consider the x86 instruction:

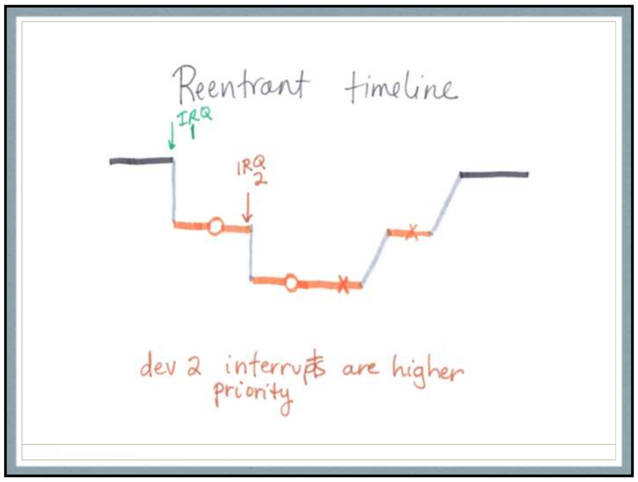
`cli` clear IF

What happens if an application includes this `cli` *instruction?*

Irrelevant (to this discussion) x86 instruction:

`sti` set IF



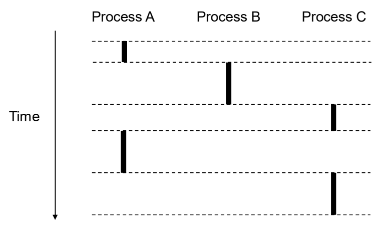


Processes

1. Process: an instance of a program in execution
2. Give the illusion that our program is the only one currently running in the system
3. Process Context consists of state including: Virtual Memory Layout, CPU registers, file descriptors, environment variables etc.
4. Key abstraction provided by a process:
   a. Independent logical control flow
   b. Private address space

Logical Control Flow

The single physical control flow of CPU is partitioned into logical control flows of several processes.



Concurrent and Parallel Flows

1. A logical flow whose execution overlaps in time with another flow is called a concurrent flow.
2. E.g. A & B are concurrent in previous slide, A & C are also concurrent while B & C are not concurrent.
3. Parallel flows: A subset of concurrent flows where the individual flows run on multiple cores or machines in parallel.

Private Address Space

1. A process provides each program the illusion that it has exclusive use of the system's address space through virtual memory.

2. This concept of private address space per process makes writing programs much easier rather than dealing with physical memory addresses. (e.g. frees the programmer from managing the physical memory resources)

Privileged Mode

1. User Mode: Cannot execute privileged instructions like one that halts the CPU. Also cannot access kernel area of address space.

2. Kernel Mode (Privileged/Supervisor Mode): Can execute any instruction and access any memory location.

Process runs application code in user mode and switches to kernel mode only via an exception like interrupt, system call etc.

Signals

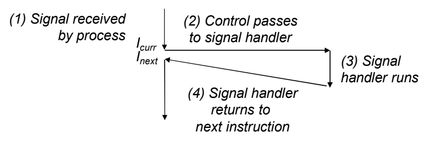
Unix Signal is a higher level software form of exceptional control flow.

A Signal is a small message that notifies a process that an event of some type has occurred in the sytem.

Processes and the Operating System can interrupt other processes using Signals.

Signal Handling

Control flow while handling signals.


(1) Signal received by process
I_{curr}
I_{next}
(2) Control passes to signal handler
(3) Signal handler runs
(4) Signal handler returns to next instruction

Signals

Low level exception we discuss in last lecture are handled by the Operating System's exception handlers and are not visible to user level processes.

Signals provide a mechanism for exposing these low level exceptions to user processes.

E.g. - If a process executes an illegal instruction, then OS kernel sends a SIGILL signal.

- If a process divides by zero, then the OS kernel sends the process a SIGFPE signal.

Signals

List of Linux Signals in "man 7 signal"

The transfer of a signal occurs in two distinct steps:

1) Sending a signal
2) Receiving a signal

Sending Signals

OS Kernel sends/delivers a signal to a destination process by updating the process context.

A signal can be sent in two ways:

1) Kernel has detected an event like divide-by-zero or termination of a child process
2) A process has invoked the kill function to explicitly request the kernel to send a signal to the destination process.

Four ways of Sending Signals

1. With /bin/kill program:
   a. "/bin/kill -9 pid" sends signal 9 (SIGKILL) to process 15213
   b. "/bin/kill -9 -pid" sends signal 9 to all processes in process group 15213
2. Sending signals from the keyboard:
   a. Typing Ctrl-C on shell sends SIGINT signal to every process in the foreground process group.
   b. Typing Ctrl-Z sends SIGTSTP to every foreground process and the result is to suspend them.

Four ways of Sending Signals

3. Sending signals with the kill function:
   int kill(pid_t pid, int sig);
   - positive pid sends signal to that process
   - negative pid sends signal to every process in process group abs(pid)
4. Sending signals with the alarm function:
   unsigned int alarm(unsigned int secs)
   - A process can send SIGALRM signals to itself by calling the alarm function.

Receiving Signals

Before kernel returns control to a process after executing a exception handler, it checks the set of unblocked pending signals.

- If the set is empty(the usual case), then control goes to the next instruction.
- If the set is not empty, then OS kernel chooses one of the pending signals and forces the process to receive the signal.

Receiving Signals

Each signal has a predefined default action which is one of:

1) The process terminates
2) The process terminates and dumps core
3) The process stops until restarted by a SIGCONT signal
4) The process ignores the signal

Receiving Signals

However, a process can choose to install its own modified default action for all signal except SIGSTOP and SIGKILL using:

sighandler_t signal(int signum, sighandler_t handler);

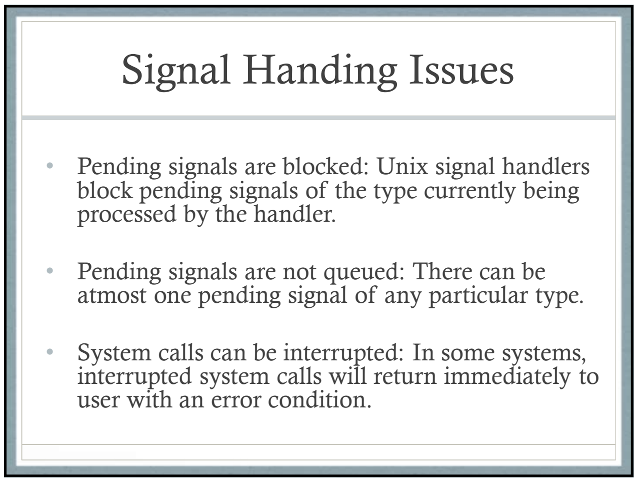
Signal handlers are yet another example of concurrency.

Receiving Signals

The signal function can change the action associated with a signal in one of three ways:

1) If handler is SIG_IGN, then signals of type signum are ignored.
2) If handler is SIG_DFL, then the action for signals of type signum reverts to the default action.
3) Otherwise, handler is the address of a user defined function called signal handler that will be invoked whenever the process receives a signal of type signum.

Example program for user defined signal handler function.



Signal Handing Issues

- Pending signals are blocked: Unix signal handlers block pending signals of the type currently being processed by the handler.
- Pending signals are not queued: There can be atmost one pending signal of any particular type.
- System calls can be interrupted: In some systems, interrupted system calls will return immediately to user with an error condition.