Instructions

Data movement

`movq Src, Dest`	Dest = Src
`movsbq Src,Dest`	Dest (quad) = Src (byte), sign-extend
`movzbq Src,Dest`	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove Src, Dest`	Equal / zero
`cmovne Src, Dest`	Not equal / not zero
`cmovs Src, Dest`	Negative
`cmovns Src, Dest`	Nonnegative
`cmovg Src, Dest`	Greater (signed >)
`cmovge Src, Dest`	Greater or equal (signed ≥)
`cmovl Src, Dest`	Less (signed <)
`cmovle Src, Dest`	Less or equal (signed ≤)
`cmova Src, Dest`	Above (unsigned >)
`cmovae Src, Dest`	Above or equal (unsigned ≥)
`cmovb Src, Dest`	Below (unsigned <)
`cmovbe Src, Dest`	Below or equal (unsigned ≤)

Control transfer

`cmpq Src2, Src1`	Sets CCs Src1 Src2
`testq Src2, Src1`	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq Src`	%rsp = %rsp−8, Mem[%rsp] = Src
`popq Dest`	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

popq D =
mov (%rsp), D
add $0x8, %rsp

pushq S =
sub $0x8, %rsp
mov S, (%rsp)

Arithmetic operations

`leaq Src, Dest`	Dest = address of Src
`incq Dest`	Dest = Dest + 1
`decq Dest`	Dest = Dest − 1
`addq Src, Dest`	Dest = Dest + Src
`subq Src, Dest`	Dest = Dest − Src
`imulq Src, Dest`	Dest = Dest ∗ Src
`xorq Src, Dest`	Dest = Dest ^ Src
`orq Src, Dest`	Dest = Dest | Src
`andq Src, Dest`	Dest = Dest & Src
`negq Dest`	Dest = − Dest
`notq Dest`	Dest = ∼ Dest
`salq` k, Dest	Dest = Dest ≪ k
`sarq` k, Dest	Dest = Dest ≫ k (arithmetic)
`shrq` k, Dest	Dest = Dest ≫ k (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer : points to the top of stack
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

%rip program counter : indicates the address in memory of the next instruction to be executed

Big endianess : store most significant byte in highest
little : store least significant byte in highest memory

Virtual address space: $N = 2^n$, as in a **n-bit** address space. Each address can use **n-bit** to represent, each address stores 1 byte. There are total 2^n addresses in the virtual address space, so the virtue address space can represent $N = 2^n$ bytes of memory.

$$\{0, 1, 2, \ldots, N-1\}$$

Physical address space: $M = 2^m$

$$\{0, 1, 2, \ldots, M-1\}$$

"Each **byte of main memory** has **a virtual address** chosen from the virtual address space, and **a physical address** chosen from the physical address space."

Virtue pages (VPs) : VM partitions the virtual memory into fixed-size blocks. Each VP is $P = 2^p$ bytes in size.

Page Table: maps virtue pages to physical pages. Page table is made up of an array of **page table entries (PTEs)**

Page table entries (PTEs): (a valid bit | n-bit address field)

- valid bit: indicates whether the virtual page is currently cached in DRAM.
- n-bit address field: "valid set" indicates the start of the corresponding physical page in DRAM where the virtue page is cached. "valid not set", then a null address indicates that the vir tual page has not been allocated, or the address points to the start of virtual page on disk.

Page Fault: DRAM cache miss, i.e., did not find the virtual page in DRAM.

n-bit virtual address: ((n-p)-bit **virtual page number (VPN)** | p-bit **virtual page offset (VPO)**)

Translation Lookaside Buffer (TLB): "a small, virtually addressed cache where each line holds a block consisting of a single PTE."

If a TLB has $T = 2^t$ sets.

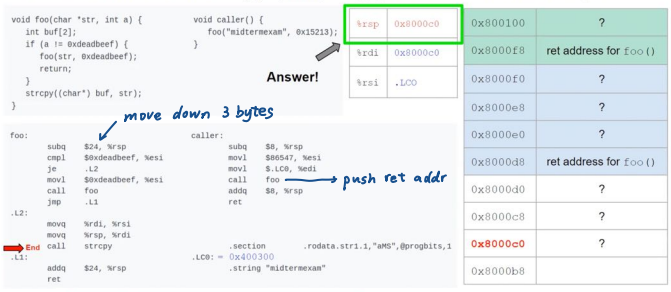
TLB tag (TLBT)	TLB index (TLBI)	VPO
VPN bits: n-1 ~ p+t	VPN bits: p+t-1 ~ p	VPO bits: p-1 ~ 0

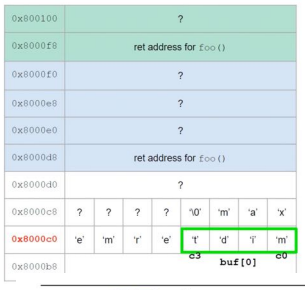
E.g. 8-way associative TLB, with 16 entries ⇒ 16/8 = 2 sets ⇒ TLBI = 1 bit

TLB Hit (VPN match, valid = 1) ⇒ Page in physical mem, No page fault.

TLB miss (PPN no match or valid = 0) ⇒ look up page table using VPN < valid = 1 / valid = 0 page fault

Question 1: What is the hex value of `%rsp` just ***before*** `strcpy()` is called for the first time in `foo()`?





buf[0] = 't' 'd' 'i' 'm'

= 74 64 69 6d

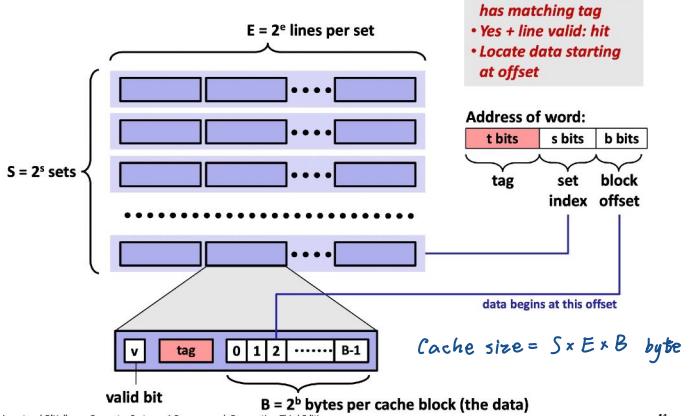
(as int) = 0x7464696d

Char	Hex	Char	Hex
a	61	m	6d
d	64	r	72
e	65	t	74
i	69	x	78

Assembly suffix	C declaration: Signed	C declaration: Unsigned	Bytes: 32-bit	Bytes: 64-bit
b	[signed] char	unsigned char	1	1
w	short	unsigned short	2	2
l	int	unsigned	4	4
q	long	unsigned long	4	8
	int32_t	uint32_t	4	4
	int64_t	uint64_t	8	8
q	char *		4	8
s	float		4	4
l	double		8	8

(signed to unsigned : sign bits extension)

Cache Read



- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Cache size = S x E x B byte

n-way associative n lines per set
direct mapped one line per set
Fully associative one set contains all lines

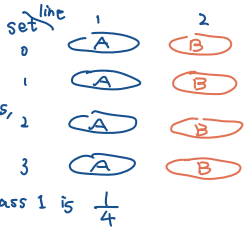
E.g. 2-way associative
4 sets — set index $2^2 \Rightarrow$ 2 bits
64 byte blocks — 2^6 byte $\Rightarrow$ Blk offset 6 bits
tag bits $\Rightarrow$ all the rest

tag | set (2) | blk offset (6)

Ex: A, B are 128 ints

```
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```

① pass 1
64 bytes → 16 ints
after every 16 ints, the set index will increment by 1
miss rate for pass 1 is $\frac{1}{4}$



② pass 2
A, B are 128 ints $\Rightarrow 128 * 4 = 2^9$ bytes
A, B's address is same for last 9 bits
ie. A[0] ~ A[15] & B[0] ~ B[15] mapps to same set
A will not be evicted miss rate : 0
B miss rate : $\frac{1}{4}$
miss rate sum: $\frac{1}{8}$

Processes

- Child calls kill(parent, SIGUSR{1,2}) between 2-4 times.
  What sequence of kills may print 1?
  Can you guarantee printing 2?
  What is the range of values printed?

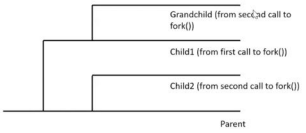
```
int counter = 0;
void handler (int sig) {
  atomically {counter++;}
}
int main(int argc, char** argv) {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);
  int parent = getpid();   int child = fork();
  if (child == 0) {
    /* insert code here */
    exit(0);
  }
  sleep(1);   waitpid(child, NULL, 0);
  printf("Received %d USR{1,2} signals\n", counter);
}
```

- Sending the same signal to the parent in all the calls to kill() may print 1 since there would be no queuing of signals.
- We can guarantee printing 2 if we send precisely one SIGUSR1 and one SIGUSR2.
- We can print 1-4 depending on the manner in which signals are sent and received.

```
int main() {
  int count = 1;
  int pid1 = fork();
  int pid2 = fork();

  if(pid1 == 0)
   count++;
  else{
   if(pid2 == 0)
        count--;
   else
        count += 2;
  }
  printf("%d", count);
}
```

What does the process diagram look like?



FILE IO

```
foo.txt: abcdefgh...xyz
int main() {
     int fd1, fd2, fd3;
     char c;
     pid_t pid;
     fd1 = open("foo.txt", O_RDONLY);
     fd2 = open("foo.txt", O_RDONLY);
     fd3 = open("foo.txt", O_RDONLY);
     read(fd1, &c, sizeof(c));          // c = a
     read(fd2, &c, sizeof(c));          // c = a
     dup2(fd2, fd3);
     read(fd3, &c, sizeof(c));          // c = b
     read(fd2, &c, sizeof(c));          // c = c
```

- H

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c
```

- H

```
pid = fork();
if (pid==0) {
     read(fd1, &c, sizeof(c));
     printf("c = %c\n", c);
     dup2(fd1, fd2);
     read(fd3, &c, sizeof(c));
     printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

Possible output 1:
c = b // in child
c = d // in child
c = c // in child
c = d // in child
c = e // in parent
c = e // in parent

Possible output 2:
c = d // in parent
c = b // in parent
c = c // in child from fd1
c = e // in child from fd3
c = d // in child
c = e // in child

- Child creates a copy of the parent fd table
  - dup2/open/close in parent affect the child
  - dup2/open/close in child do NOT affect the parent
- File descriptors across process share the same file offset.

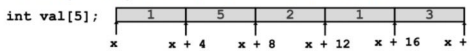
Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - (48-16) + (80-48) = 64
  - external
  - 32

	#1	#2	#3	#4		
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

Bonus Coverage: Arrays

Good toy examples (for your cheatsheet and/or big brain):



- A can be used as the pointer to the first array element: A[0]

	Type	Value
val	int *	x
val[2]	int	2
*(val + 2)	int	2
&val[2]	int *	x + 8
val + 2	int *	x + 8
val + i	int *	x + (4 * i)

Accessing methods:
- val[index]
- *(val + index)

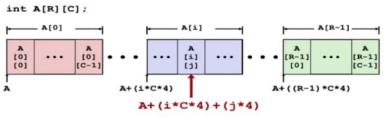
Addressing methods:
- &val[index]
- val + index

Bonus Coverage: Arrays

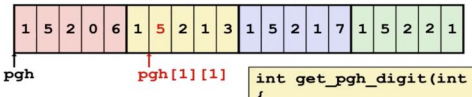
Nested indexing rules (for your cheatsheet and/or big brain):

A[i][j] is element of type T, which requires K bytes

Address A + i * (C * K) + j * K
= A + (i * C + j) * K



Nested Array Element Access Code




```
int get_pgh_digit(int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - pgh[index][dig] is int
  - Address: pgh + 20*index + 4*dig
    = pgh + 4*(5*index + dig)

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Type	Form	Operand value	Name
Immediate	$*Imm*	*Imm*	Immediate
Register	r_a	R[r_a]	Register
Memory	*Imm*	M[*Imm*]	Absolute
Memory	(r_a)	M[R[r_a]]	Indirect
Memory	*Imm*(r_b)	M[*Imm* + R[r_b]]	Base + displacement
Memory	(r_b,r_i)	M[R[r_b] + R[r_i]]	Indexed
Memory	*Imm*(r_b,r_i)	M[*Imm* + R[r_b] + R[r_i]]	Indexed
Memory	(,r_i,s)	M[R[r_i] · s]	Scaled indexed
Memory	*Imm*(,r_i,s)	M[*Imm* + R[r_i] · s]	Scaled indexed
Memory	(r_b,r_i,s)	M[R[r_b] + R[r_i] · s]	Scaled indexed
Memory	*Imm*(r_b,r_i,s)	M[*Imm* + R[r_b] + R[r_i] · s]	Scaled indexed

```
1    movl $0x4050,%eax          Immediate--Register, 4 bytes
2    movw %bp,%sp               Register--Register,  2 bytes
3    movb (%rdi,%rcx),%al       Memory--Register,    1 byte
4    movb $-17,(%esp)           Immediate--Memory,   1 byte
5    movq %rax,-12(%rbp)        Register--Memory,    8 bytes
```

Instruction		Effect	Description
leaq	*S*, *D*	*D* ← &*S*	Load effective address
INC	*D*	*D* ← *D*+1	Increment
DEC	*D*	*D* ← *D*−1	Decrement
NEG	*D*	*D* ← −*D*	Negate
NOT	*D*	*D* ← ~*D*	Complement
ADD	*S*, *D*	*D* ← *D* + *S*	Add
SUB	*S*, *D*	*D* ← *D* − *S*	Subtract
IMUL	*S*, *D*	*D* ← *D* * *S*	Multiply
XOR	*S*, *D*	*D* ← *D* ^ *S*	Exclusive-or
OR	*S*, *D*	*D* ← *D* | *S*	Or
AND	*S*, *D*	*D* ← *D* & *S*	And
SAL	*k*, *D*	*D* ← *D* << *k*	Left shift
SHL	*k*, *D*	*D* ← *D* << *k*	Left shift (same as SAL)
SAR	*k*, *D*	*D* ← *D* >>$_A$ *k*	Arithmetic right shift
SHR	*k*, *D*	*D* ← *D* >>$_L$ *k*	Logical right shift

CF	(unsigned) t < (unsigned) a	Unsigned overflow
ZF	(t == 0)	Zero
SF	(t < 0)	Negative
OF	(a < 0 == b < 0) && (t < 0 != a < 0)	Signed overflow

CMP S1, S2 (S2 – S1) TST S1, S2 (S1 & S2)

Instruction		Synonym	Effect	Set condition
sete	*D*	setz	*D* ← ZF	Equal / zero
setne	*D*	setnz	*D* ← ~ZF	Not equal / not zero
sets	*D*		*D* ← SF	Negative
setns	*D*		*D* ← ~SF	Nonnegative
setg	*D*	setnle	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
setge	*D*	setnl	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
setl	*D*	setnge	*D* ← SF ^ OF	Less (signed <)
setle	*D*	setng	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
seta	*D*	setnbe	*D* ← ~CF & ~ZF	Above (unsigned >)
setae	*D*	setnb	*D* ← ~CF	Above or equal (unsigned >=)
setb	*D*	setnae	*D* ← CF	Below (unsigned <)
setbe	*D*	setna	*D* ← CF | ZF	Below or equal (unsigned <=)

jmp	*Label*	1	Direct jump
jmp	**Operand*	1	Indirect jump

Callee Saved – Responsibility of function called to save the value before returning to function that called it. 1-6th arguments are Caller Saved

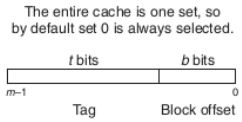
A struct within a struct gets aligned by largest value of struct

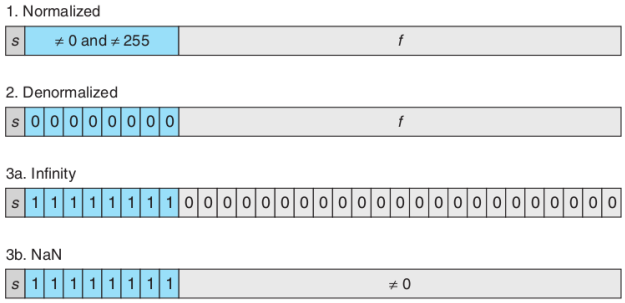
Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as

Set associative cache ($1 < E < C/B$). In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.

Figure 6.35 Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.

< E=2





For normalized numbers:
M = 1.xxxx
E = exp - bias

Bias = 2^(k-1)-1

For denormalized numbers:
M = 0.xxxx
E = 1 - bias

V = (-1)^s*M*2^E

Floating Point: Rounding

1.BGRXXX

In the below examples, imagine the underlined part as a fraction.

- Guard Bit: the least significant bit of the resulting number
- Round Bit: the first bit removed from rounding
- Sticky Bits: all bits after the round bit, OR'd together

Examples of rounding cases, including rounding to nearest even number

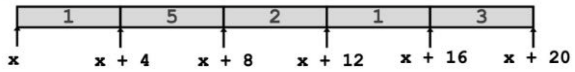
- 1.10 11: More than ½, round up: 1.11
- 1.10 10: Equal to ½, round down *to even*: 1.10
- 1.01 01: Less than ½, round down: 1.01
- 1.01 10: Equal to ½, round up *to even:* 1.10
- 1.01 00: Equal to 0, do nothing: 1.01
- 1.00 00: Equal to 0, do nothing: 1.00

All other cases involve either rounding up or down - *try them*!

Bonus Coverage: Arrays

Good toy examples (for your cheatsheet and/or big brain):

```
int val[5];
```



- A can be used as the pointer to the first array element: A[0]

	Type	Value
val	int *	x
val[2]	int	2
*(val + 2)	int	2
&val[2]	int *	x + 8
val + 2	int *	x + 8
val + i	int *	x + (4 * i)

Accessing methods:
- val[index]
- *(val + index)

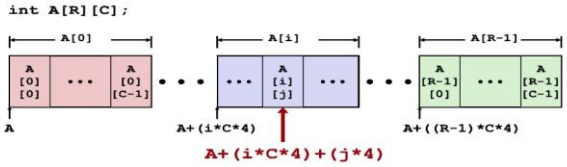
Addressing methods:
- &val[index]
- val + index

Bonus Coverage: Arrays

Nested indexing rules (for your cheatsheet and/or big brain):

A[i][j] is element of type *T*, which requires *K* bytes

Address A + i * (C * K) + j * K
= A + (i * C + j) * K



Bonus Coverage: Arrays

Nested Array Element Access Code


```
int get_pgh_digit(int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - pgh[index][dig] is int
  - Address: pgh + 20*index + 4*dig
    = pgh + 4*(5*index + dig)

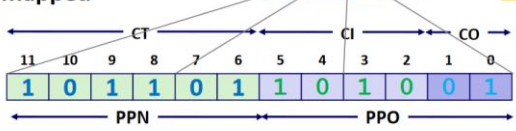
```
1  char c = 0xF0;
2  if(0xF0 == c){
3      //Never Enters unless GCCd
4      //c == 0xFFFFFFF0;
5  }
```

```
1  printf(%x);
2  //Does not include 0x
3  printf("0x%.8x \n")
4  //Prints in Little Endian
5  printf("0x%hx \n")
6  //Prints in Little Endian
```

```
1  Floats are four bytes!
2  Addresses on the stack are in order
3  All other data types on stack little endian (pointers too)
4  ! - > turns thing into signed ints
5  Size then sinage
```

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

V[0b0000110110100 1] = V[0x369]
P[0b101101101001] = P[0xB69] = 0x15



$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$

$$B2U_w(T2B_w(x)) = T2U_w(x) = x + x_{w-1}2^w$$

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u - 2^w, & u > TMax_w \end{cases}$$

$$U2T_w(u) = -u_{w-1}2^w + u$$

$$x *_w^u y = (x \cdot y) \bmod 2^w$$

$$x *_w^t y = U2T_w((x \cdot y) \bmod 2^w)$$

PRINCIPLE: Unsigned division by a power of 2

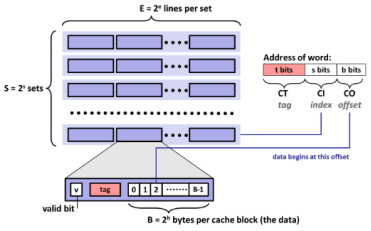
For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression x >> k yields the value $\lfloor x/2^k \rfloor$.

PRINCIPLE: Two's-complement division by a power of 2, rounding up

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that $0 \leq k < w$. The C expression (x + (1 << k) - 1) >> k, when the shift is performed arithmetically, yields the value $\lceil x/2^k \rceil$.

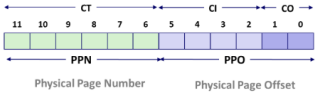
Review of Symbols

- Basic Parameters
  - N = 2^n : Number of addresses in virtual address space
  - M = 2^m : Number of addresses in physical address space
  - P = 2^p : Page size (bytes)



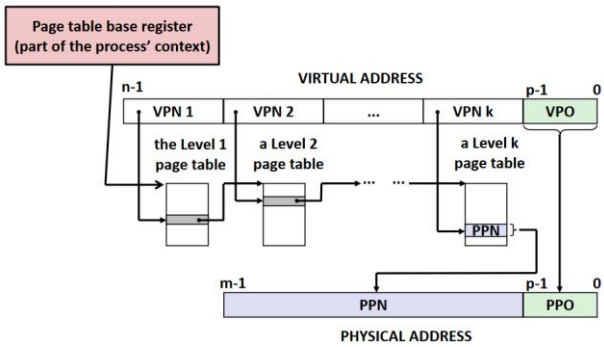
- Components of the *virtual address* (VA)
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number

- Components of the *physical address* (PA) (bits per field for our simple example)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag

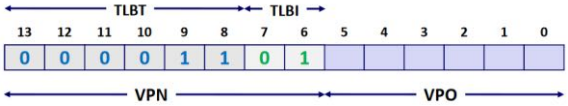


Translating with a k-level Page Table

- Having multiple levels greatly reduces page table size

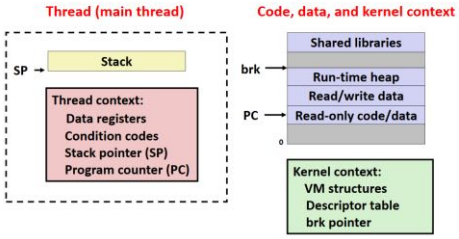


- 16 entries
- 4-way associative



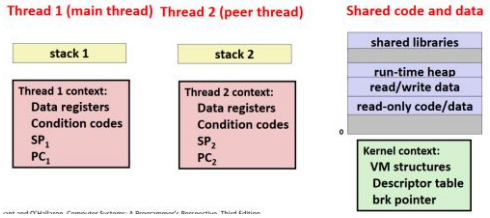
Processes and such

- **Process = thread + code, data, and kernel context**

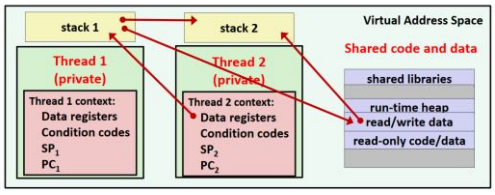


A Process With Multiple Threads

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



- **Separation of data is not strictly enforced:**
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread




```
long* p = Malloc(sizeof(long));
*p = i;
Pthread_create(&tids[i],
               NULL,
               thread,
               (void *)p);

(i = 0; i < N; i++)
```

- **Use malloc to create a per thread heap allocated place in memory for the argument**

- Creating and reaping threads
  - `pthread_create()`
  - `pthread_join()`
- Determining your thread ID
  - `pthread_self()`
- Terminating threads
  - `pthread_cancel()`
  - `pthread_exit()`
  - `exit()` [terminates all threads]
  - `return` [terminates current thread]
- Synchronizing access to shared variables
  - `pthread_mutex_init`
  - `pthread_mutex_[un]lock`

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    return 0;
}
                                          hello.c
```

Thread ID
Thread attributes (usually NULL)
Thread routine
Thread argument (void *p)
Return value (void **p)

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
                                          hello.c
```

- **Global variables**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**
- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

Shared Variable Analysis

- **Which variables are shared?**

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL, thread,(void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n"
         myid, ptr[myid], ++cnt)
  return NULL;
}
```

- **`ptr`, `cnt`, and `msgs` are shared**
- **`i` and `myid` are *not* shared**

Issues With Thread-Based Servers

- **Must run "detached" to avoid memory leak**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable
    - use `pthread_detach(pthread_self())` to make detached
- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread's stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
- **All functions called by a thread must be *thread-safe***

Appendix: Writing signal handlers

- **G1. Call only async-signal-safe functions in your handlers.**
  - Do not call `printf`, `sprintf`, `malloc`, `exit`! Doing so can cause deadlocks, since these functions may require global locks.
  - We've provided you with `sio_printf` which you can use instead.
- **G2. Save and restore `errno` on entry and exit.**
  - If not, the signal handler can corrupt code that tries to read `errno`.
  - The driver will print a warning if `errno` is corrupted.
- **G3. Temporarily block signals to protect shared data.**
  - This will prevent race conditions when writing to shared data.
- **Avoid the use of global variables in tshlab.**

Synchronization: Concurrency Issues

Avoiding Deadlock

Acquire shared resources in same order

```
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    return 0;
}
```

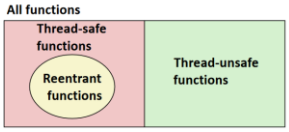
```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:	Tid[1]:
P(s0);	P(s0);
P(s1);	P(s1);
cnt++;	cnt++;
V(s0);	V(s1);
V(s1);	V(s0);

- **Classes of thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions
- **Failing to protect shared variables**
  - Fix: Use *P* and *V* semaphore operations (or mutex)
  - Example: `goodcnt.c`
  - Issue: Synchronization operations will slow down code

Reentrant Functions

- **Def: A function is *reentrant* iff it accesses no shared variables when called by multiple threads.**
  - Important subset of thread-safe functions
    - Require no synchronization operations
    - Only way to make a Class 2 function thread-safe is to make it reetnrant (e.g., `rand_r`)



Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable**
- **Manipulated by *P* and *V* operations:**
  - *P(s):* `[ while (s == 0) wait(); s--; ]`
    - Dutch for "Proberen" (test)
  - *V(s):* `[ s++; ]`
    - Dutch for "Verhogen" (increment)
- **OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
  - Only one *P* or *V* operation at a time can modify *s*.
  - When `while` loop in *P* terminates, only that *P* can decrement `s`
- **Semaphore invariant: *(s >= 0)***
- *Mutex:* binary semaphore used for mutual exclusion
  - P operation: "locking" the mutex
  - V operation: "unlocking" or "releasing" the mutex
  - *"Holding"* a mutex: locked and not yet unlocked.
- **Define and initialize a mutex for the shared variable `cnt`:**

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- **Surround critical section with *P* and *V*:**

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
                                          goodcnt.c
```

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

- **Define and initialize a mutex for the shared variable `cnt`:**

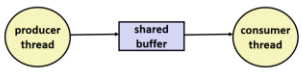
```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

- **Surround critical section with *lock* and *unlock*:**

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
```

```
linux> ./goodmcnt 10000
OK cnt=20000
linux> ./goodmcnt 10000
OK cnt=20000
```

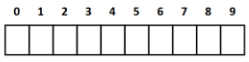
Producer-Consumer Problem



- **Common synchronization pattern:**
  - Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - Consumer waits for ***item***, removes it from buffer, and notifies producer

Circular Buffer (n = 10)

- **Store elements in array of size n**
- **items: number of elements in buffer**
- **Empty buffer:**
  - front = rear
- **Nonempty buffer**
  - rear: index of most recently inserted element
  - front: (index of next element to remove – 1) mod n
- **Initially:**



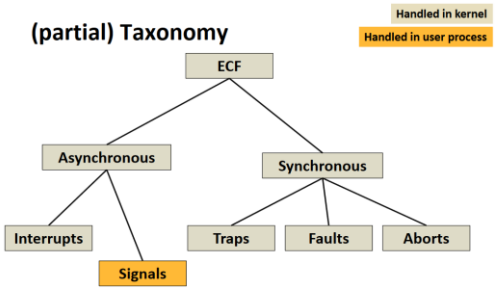

```
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                  /* Buffer holds max of n items */
    sp->front = sp->rear = 0;   /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero items */
}
```

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);              /* Wait for available slot */
    P(&sp->mutex);              /* Lock the buffer         */
    if (++sp->rear >= sp->n)    /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item;   /* Insert the item         */
    V(&sp->mutex);              /* Unlock the buffer       */
    V(&sp->items);              /* Announce available item */
}
                                          sbuf.c
```

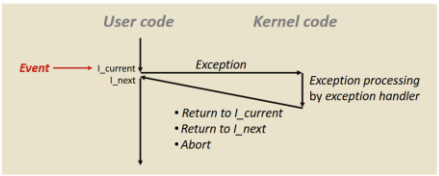
```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);              /* Wait for available item */
    P(&sp->mutex);              /* Lock the buffer         */
    if (++sp->front >= sp->n)   /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];  /* Remove the item         */
    V(&sp->mutex);              /* Unlock the buffer       */
    V(&sp->slots);              /* Announce available slot */
    return item;
}
                                          sbuf.c
```

ECF

(partial) Taxonomy

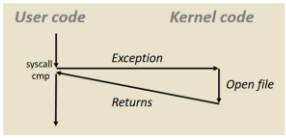


- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



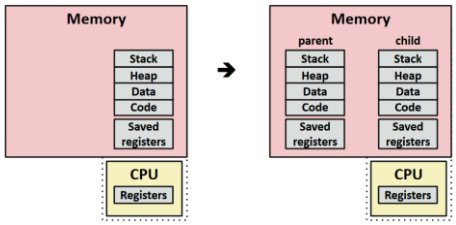
- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
0000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:  0f 05               syscall         # Return value in %rax
e5d80:  48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:  c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Conceptual View of `fork`



WAITPID

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t *pid*, **int** *stat_loc*, **int** *options*);

If *pid* is equal to (**pid_t**)-1, *status* is requested for any child process. In this respect, *waitpid*() is then equivalent to *wait*().

If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is requested.

If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than (**pid_t**)-1, *status* is requested for any child process whose process group ID is equal to the absolute value of *pid*.

WIFEXITED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)

If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-orde 8 bits of the *status* argument that the child process passed to *_exit*() or *exit*(), or the value the child process returned from *main*().

WIFSIGNALED(*stat_val*)

Evaluates to a non-zero value if *status* was returned for a child process that terminated due to the receipt of a signal that was not caught (see <*signal.h*>).

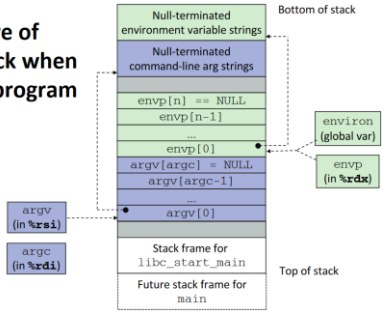
Return Value

If *wait*() or *waitpid*() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*() or *waitpid*() returns due to the delivery of a signal to the calling process, -1 shall be returned and *errno* set to [EINTR]. If *waitpid*() was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, (**pid_t**)-1 shall be returned, and *errno* set to indicate the error.

EXECVE

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
  - Executable file `filename`
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - ...with argument list `argv`
    - By convention `argv[0]==filename`
  - ...and environment variable list `envp`
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv`, `putenv`, `printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - ...except if there is an error

Structure of the stack when a new program starts



Signal Handling

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`
  - Otherwise, `handler` is the address of a user-level *signal handler*
- Implicit blocking mechanism
  - Kernel blocks any pending signals of type currently being handled
  - e.g., a SIGINT handler can't be interrupted by another SIGINT
- Explicit blocking and unblocking mechanism
  - `sigprocmask` function

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    /* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

Guidelines for Writing Safe Handlers

- G0: Keep your handlers as simple as possible
  - e.g., set a global flag and return
- G1: Call only async-signal-safe functions in your handlers
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- G2: Save and restore `errno` on entry and exit
  - So that other handlers don't overwrite your value of `errno`
- G3: Protect accesses to shared data structures by temporarily blocking all signals
  - To prevent possible corruption
- G4: Declare global variables as `volatile`
  - To prevent compiler from storing them in a register
- G5: Declare global flags as `volatile sig_atomic_t`
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals
- Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals
- Posix guarantees 117 functions to be async-signal-safe
  - Source: "`man 7 signal-safety`"
  - Popular functions on the list:
    - `_exit`, `write`, `wait`, `waitpid`, `sleep`, `kill`

```
while (n--) {
    Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
    if ((pid = Fork()) == 0) { /* Child process */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
        Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
    addjob(pid);  /* Add the child to the job list */
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
}
exit(0);
```

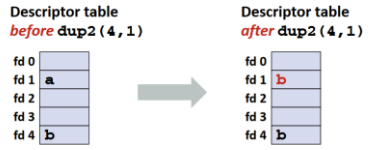
Preventing Races

- `int sigsuspend(const sigset_t *mask)`
- Equivalent to atomic (uninterruptable) version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

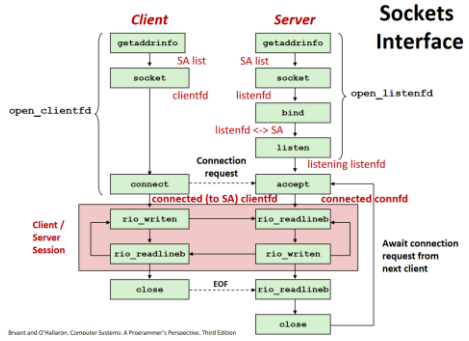
FILE IO

- Answer: By calling the `dup2(oldfd, newfd)` function
  - Copies (per-process) descriptor table entry `oldfd` to entry `newfd`

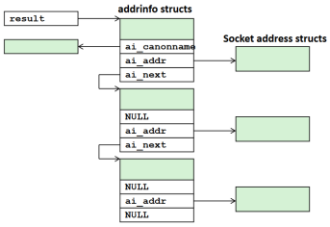


- When to use standard I/O
  - When working with disk or terminal files
- When to use raw Unix I/O
  - *Inside signal handlers, because Unix I/O is async-signal-safe*

Network



Review: `getaddrinfo` Linked List



- Clients: walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- Servers: walk the list until calls to `socket` and `bind` succeed.

OTHER

Snoopy Caches
- invalid : cannot use value
- shared : readable copy
- exclusive : writable copy

start with E → S

VM locality
- working set size < main memory
  good performance
- working set size > main memory
  thrashing

KB ⇒ 2^{10}

- deadlock : nothing can proceed (need locks)
- livelock : infinite loop
- starvation : unfairness, one has priority

Garbage collection
- mark & sweep : mark all connected to roots
  sweep all allocated and not marked.