DEC	HEX	BIN	DEC	HEX	BIN	DEC	HEX	BIN
0	00	00000000	43	2B	00101011	86	56	01010110
1	01	00000001	44	2C	00101100	87	57	01010111
2	02	00000010	45	2D	00101101	88	58	01011000
3	03	00000011	46	2E	00101110	89	59	01011001
4	04	00000100	47	2F	00101111	90	5A	01011010
5	05	00000101	48	30	00110000	91	5B	01011011
6	06	00000110	49	31	00110001	92	5C	01011100
7	07	00000111	50	32	00110010	93	5D	01011101
8	08	00001000	51	33	00110011	94	5E	01011110
9	09	00001001	52	34	00110100	95	5F	01011111
10	0A	00001010	53	35	00110101	96	60	01100000
11	0B	00001011	54	36	00110110	97	61	01100001
12	0C	00001100	55	37	00110111	98	62	01100010
13	0D	00001101	56	38	00111000	99	63	01100011
14	0E	00001110	57	39	00111001	100	64	01100100
15	0F	00001111	58	3A	00111010	101	65	01100101
16	10	00010000	59	3B	00111011	102	66	01100110
17	11	00010001	60	3C	00111100	103	67	01100111
18	12	00010010	61	3D	00111101	104	68	01101000
19	13	00010011	62	3E	00111110	105	69	01101001
20	14	00010100	63	3F	00111111	106	6A	01101010
21	15	00010101	64	40	01000000	107	6B	01101011
22	16	00010110	65	41	01000001	108	6C	01101100
23	17	00010111	66	42	01000010	109	6D	01101101
24	18	00011000	67	43	01000011	110	6E	01101110
25	19	00011001	68	44	01000100	111	6F	01101111
26	1A	00011010	69	45	01000101	112	70	01110000
27	1B	00011011	70	46	01000110	113	71	01110001
28	1C	00011100	71	47	01000111	114	72	01110010
29	1D	00011101	72	48	01001000	115	73	01110011
30	1E	00011110	73	49	01001001	116	74	01110100
31	1F	00011111	74	4A	01001010	117	75	01110101
32	20	00100000	75	4B	01001011	118	76	01110110
33	21	00100001	76	4C	01001100	119	77	01110111
34	22	00100010	77	4D	01001101	120	78	01111000
35	23	00100011	78	4E	01001110	121	79	01111001
36	24	00100100	79	4F	01001111	122	7A	01111010
37	25	00100101	80	50	01010000	123	7B	01111011
38	26	00100110	81	51	01010001	124	7C	01111100
39	27	00100111	82	52	01010010	125	7D	01111101
40	28	00101000	83	53	01010011	126	7E	01111110
41	29	00101001	84	54	01010100	127	7F	01111111
42	2A	00101010	85	55	01010101			

negative conversion (bin):

* flip bits (all 0's = 1's, 1's = 0's) = ~(NOT)
* Add 1
* replace first byte w/ 1

bin. addition tip:

* 1 + 1 = 10
* 1 + 1 + 1 = 11

bitwise operators:

"&" AND, "~" NOT, "||" OR
"^" XOR

AND	1	0
1	1	0
0	0	0

NOT	1	0
	0	1

OR	1	0
1	1	1
0	1	0

XOR	1	0
1	0	1
0	1	0

logical operators:

* non-zeros are true
* all zeros is false
(bin) 1 = 0001
(bin) 0 = 0000
* "&&" AND, "||" OR, "!" NOT

Shifting:

* ">>n" RIGHT
* add zeros in empty spaces
* (arithmetic) add first value in number to empty space.
* eq. mult by 2^{-n}

* "<<n" "LEFT"
* add zeros in empty space
* eq. multiply 2^n

Endian order:

* (Big Endian) left to right. BIG to SMALL
Ex) $2 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$
* (Little Endian) SMALL to BIG
$2 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 7$

Binary Numbers:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1024	512	256	128	64	32	16	8	4	2	1		1/2	1/4	1/8	1/16

Hexadecimal:

16^3	16^2	16^1	16^0
4096	256	16	1

* maximum: largest int. generated w/ given bits
* minimum: smallest int. generated w/ given bits

Float:

$$float = (-1)^{S} \cdot 1.frac \cdot 2^{exp-127}$$

Convert int to float

* convert int into binary

$-46.5 = -101110.1$

* move decimal point up (n) to create notation to 2^n

$-1.011101 * 2^5$

* Break it down into function

$(-1)^1 * 1.011101 * 2^{132-127}$

$S = 1$ $frac = 011101....0$ (23 bits) $exp = 132$ → convert to bin → 10000100

* put it together [s exp frac]

1 10000100 01110100000000000000

C 2 3 A 0 0 0 0

* convert to hex via group by 4

0x C23A0000

* Remember division is just multiplying by a fraction, so...

$-46.5/4 = 46.5 * \frac{1}{4} = -46.5 \cdot 2^{-2}$

so....

mult($* 2^n$) multiply by n

div($* 2^{-n}$) divide by n

$(-1)^1 \cdot 1.011101 * [2^5 * 2^{-2}]$

+ simplify

$(-1)^1 \cdot 1.01101 * 2^3$

* continue conversion

convert an int to hex

* find the absolute value of the number

$-93 \rightarrow 93 \rightarrow 1011101$

* then represent it as a float (32 bits) by adding zeros

0000 0000 0000 0000 0000 0000 1011 1101

*(inversion) not (~) the float, all 1's = 0, 0's = 1

1111 1111 1111 1111 1111 111 0100 0010

+1

*(addition) adding 1

1111 1111 1111 1111 1111 111 0100 0011

* group by 4 to convert into hex

1111 1111 1111 1111 1111 111 0100 0011

F F F F F F A 3

Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex
0	00	24	18	48	30	72	48	96	60	120	78	144	90	168	A8
1	01	25	19	49	31	73	49	97	61	121	79	145	91	169	A9
2	02	26	1A	50	32	74	4A	98	62	122	7A	146	92	170	AA
3	03	27	1B	51	33	75	4B	99	63	123	7B	147	93	171	AB
4	04	28	1C	52	34	76	4C	100	64	124	7C	148	94	172	AC
5	05	29	1D	53	35	77	4D	101	65	125	7D	149	95	173	AD
6	06	30	1E	54	36	78	4E	102	66	126	7E	150	96	174	AE
7	07	31	1F	55	37	79	4F	103	67	127	7F	151	97	175	AF
8	08	32	20	56	38	80	50	104	68	128	80	152	98	176	B0
9	09	33	21	57	39	81	51	105	69	129	81	153	99	177	B1
10	0A	34	22	58	3A	82	52	106	6A	130	82	154	9A	178	B2
11	0B	35	23	59	3B	83	53	107	6B	131	83	155	9B	179	B3
12	0C	36	24	60	3C	84	54	108	6C	132	84	156	9C	180	B4
13	0D	37	25	61	3D	85	55	109	6D	133	85	157	9D	181	B5
14	0E	38	26	62	3E	86	56	110	6E	134	86	158	9E	182	B6
15	0F	39	27	63	3F	87	57	111	6F	135	87	159	9F	183	B7
16	10	40	28	64	40	88	58	112	70	136	88	160	A0	184	B8
17	11	41	29	65	41	89	59	113	71	137	89	161	A1	185	B9
18	12	42	2A	66	42	90	5A	114	72	138	8A	162	A2	186	BA
19	13	43	2B	67	43	91	5B	115	73	139	8B	163	A3	187	BB
20	14	44	2C	68	44	92	5C	116	74	140	8C	164	A4	188	BC
21	15	45	2D	69	45	93	5D	117	75	141	8D	165	A5	189	BD
22	16	46	2E	70	46	94	5E	118	76	142	8E	166	A6	190	BE
23	17	47	2F	71	47	95	5F	119	77	143	8F	167	A7	191	BF

Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex	Num	Hex
192	C0	216	D8	240	F0										
193	C1	217	D9	241	F1										
194	C2	218	DA	242	F2										
195	C3	219	DB	243	F3										
196	C4	220	DC	244	F4										
197	C5	221	DD	245	F5										
198	C6	222	DE	246	F6										
199	C7	223	DF	247	F7										
200	C8	224	E0	248	F8										
201	C9	225	E1	249	F9										
202	CA	226	E2	250	FA										
203	CB	227	E3	251	FB										
204	CC	228	E4	252	FC										
205	CD	229	E5	253	FD										
206	CE	230	E6	254	FE										
207	CF	231	E7	255	FF										
208	D0	232	E8	256	100										
209	D1	233	E9												
210	D2	234	EA												
211	D3	235	EB												
212	D4	236	EC												
213	D5	237	ED												
214	D6	238	EE												
215	D7	239	EF												

Assembly Basics:

%rax (function result)	%r8 (fifth argument)
%rbx	%r9 (sixth argument)
%rcx (fourth argument)	%r10
%rdx (third argument)	%r11
%rsi (second argument)	%r12
%rdi (first argument)	%r13
%rsp (stack pointer)	%r14
%rbp	%r15

- `cmp a,b` like computing `b-a` without setting destination
  $b - a == 0$
- `test a,b` like computing `a&b` without setting destination
  $a \& b == 0$
- Test for zero: `test %rax, %rax`
  `%rax`

* src, dest
* "$Imm" - refers to immediates or exact values.
  Ex: $47
* "0xn" - refers to value @ this location in memory
  Ex: 0x0468
* "(%reg)" - refers memory @ address stored in a register.
  - ↳ go to register (%)
  - ↳ get value in register (%)
  - ↳ interpret that value as a memory address
  - ↳ go into memory @ that address
  - ↳ get value in memory @ that address

  Ex: %rax
* "Imm(%rbp)"
  - ↳ go into register (%)
  - ↳ get value in register (%)
  - ↳ interpret it as an address
  - ↳ add constant to that address to get a new one
  - ↳ go into memory @ that new address
  - ↳ get value @ that new address
  - ↳ IF value = src, copy value to this new destination
  - ↳ IF value = dest, override value w/ new destination.
* "Imm(%r1, %r2, c)" - this for accessing values in an array.
  - ↳ compute new memory address by Imm + r1 + (r2 * c)
  - ↳ r1 is a pointer to first element in an array.
  - ↳ r2 is index your trying to access
  - ↳ s size of each element in array

Move Operand Combinations:

	Source	Dest	Src, Dest	C Analog
mov	Imm	Register	mov $0x4, %rax	temp = 4; * move 4 into register %rax
		memory	mov $0x4, (%rax)	*p = 147; * move 4 into address (value in register %rax) in memory
	Register	Reg	mov %rax, %rdx	temp2 = temp; * copy %rax into %rdx
		Mem	mov %rax, (%rdx)	*p = temp; * move %rax into address (value in register %rdx) in memory
	memory	Reg	mov (%rax), %rdx	temp = *p; * move value @ address (value in register %rax) in memory into %rdx.

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

move assembly suffix	move data src → dest
movb	move 1 bytes
movw	move 2 bytes
movl	move 4 bytes
movq	move 8 bytes

leaq Src, Dest

↳ store new computed value (1) in Src (2) into Dest

↳ Example: leaq (%rdi, %rdi, 2), %rax

(1) %rdi + rdi * 2 → new address (or pointer)

(2) store new address into %rax

↳ performs pointer arithmetic → p = &(x[i]); or p = x + i;

Example (leaq):

* pointer arithmetic is just arithmetic expressions of the form x + k*y
  - k can be 1, 2, 4, or 8

So x*12 can be

```
leaq (%rdi,%rdi,2),%rax  # t = x + x*2
                         # performs mult times 3
salq $2,%rax             # return t<<2
                         # perform mult times 4 (2^2)
```

Control Flow:

Jumps:

↳ A jump `jmp` causes the execution of the program to switch to a new position.

↳ `jmp Label` → indicate the address in memory of subsequent instruction.

`jmp *operand` → pass a function as a param.

Condition codes:

`cmpq a, b` → compute b-a

↳ a `cmp` is followed a jump `jmp` instruction on how to compare b-a to zero.

↳ IF true then JUMP to specified Label.

ELSE keep going.

`testq a, b` → compute b&a

↳ a `test` is followed a jump `jmp` instruction on how to compare b&a to zero

↳ IF a=0, b=0 then a&b==0 ✓

↳ ELSE a=1, b=1 then a&b==0 ✗

`jX Label`

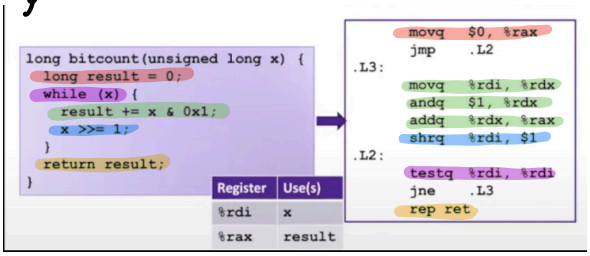
jX	Description
jmp	Unconditional
je	Equal / Zero
jne	Not Equal / Not Zero
jl	Less (Signed)
jle	Less or Equal (Signed)
jg	Greater (Signed)
jge	Greater or Equal (Signed)

cmpq		testq
→ b-a == 0	b==a	b&a == 0
→ b-a != 0	b!=a	b&a != 0
→ b-a < 0	b<a	b&a < 0
→ b-a ≤ 0	b>a	b&a ≤ 0
→ b-a > 0	b>a	b&a > 0
→ b-a ≥ 0	b<a	b&a ≥ 0

While loops:

(*) initialize variable
(*) condition to check to see if it's still true
(*) work to be done to update result
(*) incrementation to update condition.
(*) return result

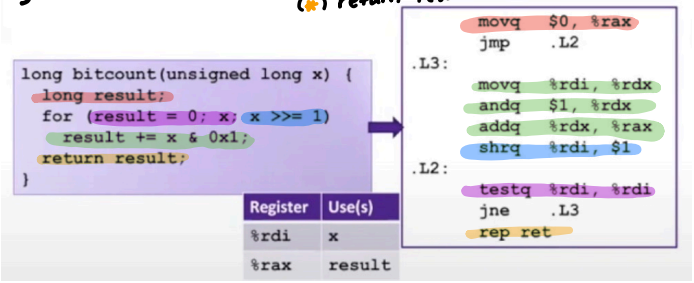
```
while (condition) {
    Body
    Incr
}
```



For Loops:

(*) initialize variable
(*) condition to check to see if it's still true
(*) work to be done to update result
(*) incrementation to update condition.
(*) return result

```
for (Init; Cond; Incr) {
    Body
}
```



DEC	HEX	BIN	DEC	HEX	BIN	DEC	HEX	BIN
0	00	00000000	43	2B	00101011	86	56	01010110
1	01	00000001	44	2C	00101100	87	57	01010111
2	02	00000010	45	2D	00101101	88	58	01011000
3	03	00000011	46	2E	00101110	89	59	01011001
4	04	00000100	47	2F	00101111	90	5A	01011010
5	05	00000101	48	30	00110000	91	5B	01011011
6	06	00000110	49	31	00110001	92	5C	01011100
7	07	00000111	50	32	00110010	93	5D	01011101
8	08	00001000	51	33	00110011	94	5E	01011110
9	09	00001001	52	34	00110100	95	5F	01011111
10	0A	00001010	53	35	00110101	96	60	01100000
11	0B	00001011	54	36	00110110	97	61	01100001
12	0C	00001100	55	37	00110111	98	62	01100010
13	0D	00001101	56	38	00111000	99	63	01100011
14	0E	00001110	57	39	00111001	100	64	01100100
15	0F	00001111	58	3A	00111010	101	65	01100101
16	10	00010000	59	3B	00111011	102	66	01100110
17	11	00010001	60	3C	00111100	103	67	01100111
18	12	00010010	61	3D	00111101	104	68	01101000
19	13	00010011	62	3E	00111110	105	69	01101001
20	14	00010100	63	3F	00111111	106	6A	01101010
21	15	00010101	64	40	01000000	107	6B	01101011
22	16	00010110	65	41	01000001	108	6C	01101100
23	17	00010111	66	42	01000010	109	6D	01101101
24	18	00011000	67	43	01000011	110	6E	01101110
25	19	00011001	68	44	01000100	111	6F	01101111
26	1A	00011010	69	45	01000101	112	70	01110000
27	1B	00011011	70	46	01000110	113	71	01110001
28	1C	00011100	71	47	01000111	114	72	01110010
29	1D	00011101	72	48	01001000	115	73	01110011
30	1E	00011110	73	49	01001001	116	74	01110100
31	1F	00011111	74	4A	01001010	117	75	01110101
32	20	00100000	75	4B	01001011	118	76	01110110
33	21	00100001	76	4C	01001100	119	77	01110111
34	22	00100010	77	4D	01001101	120	78	01111000
35	23	00100011	78	4E	01001110	121	79	01111001
36	24	00100100	79	4F	01001111	122	7A	01111010
37	25	00100101	80	50	01010000	123	7B	01111011
38	26	00100110	81	51	01010001	124	7C	01111100
39	27	00100111	82	52	01010010	125	7D	01111101
40	28	00101000	83	53	01010011	126	7E	01111110
41	29	00101001	84	54	01010100	127	7F	01111111
42	2A	00101010	85	55	01010101			

how to find data in direct cache via an address?

Address of data: tag | index | offset

* **index:** find which cacheline our data can be stored in.
  - ↳ $\log_2$(# lines in cache)
* **offset:** indentifies which byte in the datablock will correspond to address your looking for
  - ↳ $\log_2$(size of datablock in bytes)
* **tag:** which address is stored in the cacheline currently.

① chop address up into tag, index, and offset.
② utilize index and go to that cacheline.
③ check the tag for a match
- ↳ IF tag matches and valid bit is valid, you have a cache hit. use offset to retrieve your data via indexing in datablock
- ↳ ELSE IF tag does not match or valid bit is not set, then cache miss.
  - ↳ go into memory to access data at that address but bring a copy into the cache and put it @ that index in datablock.

how to find data in S.A. cache given an address?

index: which set the data could be in.
- ↳ $\log_2$(# number of sets in cache) — total entries / type of cache

offset: which bit in the datablock will correspond to the address your looking for.
- ↳ $\log_2$(size of datablock)

tag: which address is stored in the cacheline currently.

① Divide address into tag, index, offset by caculating.
index → $\log_2$(# of sets)
offset → $\log_2$(size of datablock)

② find set the data could be a part of w/ index.

③ find if the tag matches and a valid bit is there.
- ↳ IF tag matches and valid bit is valid (1) then use the offset to find the index of the data bit in that cacheline/ data block.
- ↳ ELSE go in memory using the offset as the index in memory. then copy, that value in memory in the datablock of the cache line.

Writing to cache

wr val, Destination

val: numeric value to be inserted
des: address in main memory

* **write-through + no write-allocate:**

hit: update old value in memory @ des to val in cache.

miss: copy the data @ des in memory into the cache. Ex) mem[des]

* **write-back + write allocate:**

hit: only put the val in cache for byte read.

miss: put the val @ des for the byte read.
Ex) mem[des]

DEC	HEX	BIN	DEC	HEX	BIN	DEC	HEX	BIN
0	00	00000000	43	2B	00101011	86	56	01010110
1	01	00000001	44	2C	00101100	87	57	01010111
2	02	00000010	45	2D	00101101	88	58	01011000
3	03	00000011	46	2E	00101110	89	59	01011001
4	04	00000100	47	2F	00101111	90	5A	01011010
5	05	00000101	48	30	00110000	91	5B	01011011
6	06	00000110	49	31	00110001	92	5C	01011100
7	07	00000111	50	32	00110010	93	5D	01011101
8	08	00001000	51	33	00110011	94	5E	01011110
9	09	00001001	52	34	00110100	95	5F	01011111
10	0A	00001010	53	35	00110101	96	60	01100000
11	0B	00001011	54	36	00110110	97	61	01100001
12	0C	00001100	55	37	00110111	98	62	01100010
13	0D	00001101	56	38	00111000	99	63	01100011
14	0E	00001110	57	39	00111001	100	64	01100100
15	0F	00001111	58	3A	00111010	101	65	01100101
16	10	00010000	59	3B	00111011	102	66	01100110
17	11	00010001	60	3C	00111100	103	67	01100111
18	12	00010010	61	3D	00111101	104	68	01101000
19	13	00010011	62	3E	00111110	105	69	01101001
20	14	00010100	63	3F	00111111	106	6A	01101010
21	15	00010101	64	40	01000000	107	6B	01101011
22	16	00010110	65	41	01000001	108	6C	01101100
23	17	00010111	66	42	01000010	109	6D	01101101
24	18	00011000	67	43	01000011	110	6E	01101110
25	19	00011001	68	44	01000100	111	6F	01101111
26	1A	00011010	69	45	01000101	112	70	01110000
27	1B	00011011	70	46	01000110	113	71	01110001
28	1C	00011100	71	47	01000111	114	72	01110010
29	1D	00011101	72	48	01001000	115	73	01110011
30	1E	00011110	73	49	01001001	116	74	01110100
31	1F	00011111	74	4A	01001010	117	75	01110101
32	20	00100000	75	4B	01001011	118	76	01110110
33	21	00100001	76	4C	01001100	119	77	01110111
34	22	00100010	77	4D	01001101	120	78	01111000
35	23	00100011	78	4E	01001110	121	79	01111001
36	24	00100100	79	4F	01001111	122	7A	01111010
37	25	00100101	80	50	01010000	123	7B	01111011
38	26	00100110	81	51	01010001	124	7C	01111100
39	27	00100111	82	52	01010010	125	7D	01111101
40	28	00101000	83	53	01010011	126	7E	01111110
41	29	00101001	84	54	01010100	127	7F	01111111
42	2A	00101010	85	55	01010101			

Coalescing

* for each block we need the size and allocation status.
* the header stores the size of block and a variable a to determine a = 1 (allocated block) and a = 0 (free)
  - ↳ 2 lower order bits of the address of header or footer
* the header and footer are the same, so they contain block size and allocation status for same block
* header of one block is next to footer of another block. vice versa
* IF two or more blocks are adjacent and free then merge them.

Solving Coalescing

① the header is n bytes (n = size of one block) BEFORE the pay load.

② Get the two lower order bits by breaking down last hex in header address.
ex) 0x00000024 —bin→ 0100

③ Observe if block is free or allocated.

④ Calculate the size of a block:
- ↳ make all lower order bits zero
- ↳ this changes the last hex value of the address
- ↳ convert the address to decimal

tip: where numbers start in address is what to convert
ex) 0x00000020 → 32 (decimal)

ex) 0x00000024 → 0100 → 0000 → 0x00000020

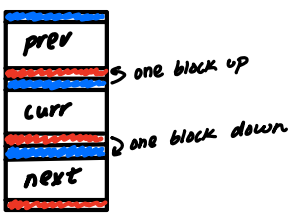
- ↳ divide the decimal by size of a block in stack for # of blocks in that block.

$$\#\text{ of blocks making up a block} = \frac{\text{decimal}}{\text{size of a block}}$$

⑤ Count up (# of blocks in a block) to get the footer.

⑥ Now, repeat ① to find free blocks by using the header of previous block and footer of the next block.

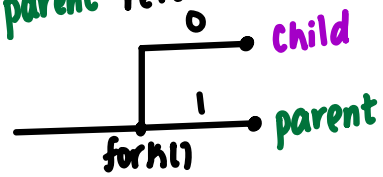
Remember:



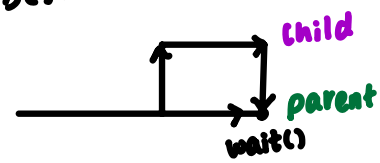
Forking:

fork(): used to initialize a process or create a child and parent process from previous process.

* child returns 0
* parent returns 1



wait(): make a parent process wait to execute before a child process.



* Beware of printf, if(), and returns. Remember this is a program.

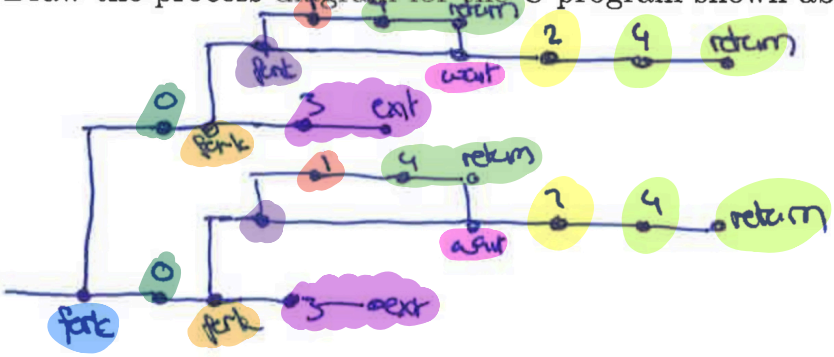
* **printf()**: print statement
* **return or exit()**: end program

Binary

2^{10} 2^{9} 2^{8} 2^{7} 2^{6} 2^{5} 2^{4} 2^{3} 2^{2} 2^{1} 2^{0}

1024 512 256 128 64 32 16 8 4 2 1

Draw the process diagram for the C program shown abc

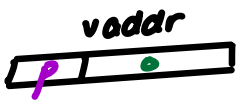



```
int main () {
1   fork();
2   print("0");
    if (fork() == 0) {
        if (fork() == 0){ 3
            printf("1"); 4
        } else {
            int status;
            wait(&status); 6
            printf("2"); 7
        }
    } else {
        printf("3");
        exit(0); 9
    }
5   printf("4");
    return 0;
}
```

8

Paging: (vaddr → paddr)

* Break virtual memory down into binary.
* partition the bits such that,
  - ↳ offset = $\log_2$(size of pages)
  - ↳ page # (frame #) = rest of bits before offset



* If page is valid then, calculate physical address
  - ↳ frame number, followed by offset

paddr

* ELSE page fault

DEC	HEX	BIN	DEC	HEX	BIN	DEC	HEX	BIN
0	00	00000000	43	2B	00101011	86	56	01010110
1	01	00000001	44	2C	00101100	87	57	01010111
2	02	00000010	45	2D	00101101	88	58	01011000
3	03	00000011	46	2E	00101110	89	59	01011001
4	04	00000100	47	2F	00101111	90	5A	01011010
5	05	00000101	48	30	00110000	91	5B	01011011
6	06	00000110	49	31	00110001	92	5C	01011100
7	07	00000111	50	32	00110010	93	5D	01011101
8	08	00001000	51	33	00110011	94	5E	01011110
9	09	00001001	52	34	00110100	95	5F	01011111
10	0A	00001010	53	35	00110101	96	60	01100000
11	0B	00001011	54	36	00110110	97	61	01100001
12	0C	00001100	55	37	00110111	98	62	01100010
13	0D	00001101	56	38	00111000	99	63	01100011
14	0E	00001110	57	39	00111001	100	64	01100100
15	0F	00001111	58	3A	00111010	101	65	01100101
16	10	00010000	59	3B	00111011	102	66	01100110
17	11	00010001	60	3C	00111100	103	67	01100111
18	12	00010010	61	3D	00111101	104	68	01101000
19	13	00010011	62	3E	00111110	105	69	01101001
20	14	00010100	63	3F	00111111	106	6A	01101010
21	15	00010101	64	40	01000000	107	6B	01101011
22	16	00010110	65	41	01000001	108	6C	01101100
23	17	00010111	66	42	01000010	109	6D	01101101
24	18	00011000	67	43	01000011	110	6E	01101110
25	19	00011001	68	44	01000100	111	6F	01101111
26	1A	00011010	69	45	01000101	112	70	01110000
27	1B	00011011	70	46	01000110	113	71	01110001
28	1C	00011100	71	47	01000111	114	72	01110010
29	1D	00011101	72	48	01001000	115	73	01110011
30	1E	00011110	73	49	01001001	116	74	01110100
31	1F	00011111	74	4A	01001010	117	75	01110101
32	20	00100000	75	4B	01001011	118	76	01110110
33	21	00100001	76	4C	01001100	119	77	01110111
34	22	00100010	77	4D	01001101	120	78	01111000
35	23	00100011	78	4E	01001110	121	79	01111001
36	24	00100100	79	4F	01001111	122	7A	01111010
37	25	00100101	80	50	01010000	123	7B	01111011
38	26	00100110	81	51	01010001	124	7C	01111100
39	27	00100111	82	52	01010010	125	7D	01111101
40	28	00101000	83	53	01010011	126	7E	01111110
41	29	00101001	84	54	01010100	127	7F	01111111
42	2A	00101010	85	55	01010101			

Two level Page Tables:

* Breakdown vaddr into binary.
* Partition vaddr into:

  idx1, idx2, offset

  offset = $\log_2$(size of pages)

  idx2 = log(# page entries in one page)

  idx1 = remaining bits

* index into page directory w/ idx1 to find the frame
* now, in that frame or page table use idx2 to find that page
* Calculate the physical memory using frame number @ idx2.

* IF page exists but not valid, then PAGE FAULT
* IF page is NULL or does not exist then SEG FAULT

Translation-Lookaside Buffer (TLB):

* Breakdown vaddr into binary.
* Partition vaddr into:
  - ↳ offset = $\log_2$(size of pages)
  - ↳ (VP) virtual page # = remaining bits before the offset.

* Then partition VP into idx and tag.
  - ↳ idx = $\log_2$(# of sets in TLB) ← total # of entries / type of cache
  - ↳ tag = remaining bits in VP bef. idx

* If valid then calculate physical address PPN then offset

max file size that can be stored in direct pointer

$$(\text{number of direct pointers}) * (\text{size of blocks})$$

maximum number of files that can be stored in a single directory

$$\left(\frac{\text{maximum file size}}{\text{direct entry size}}\right) - 2$$

max file size that can be stored in a file system

n = layer

indirect (n=1)
doubly_ind (n=2)
triply_ind (n=3)

$$(\text{number of direct pointers}) * (\text{size of blocks}) + \left[\frac{(\text{size of blocks})}{(\text{size of block num.})}\right]^{n} * (\text{size of blocks})$$

① make sure you do this for each layer
② then sum everything together

Sequence block Reads

① Caculate the # of data blocks in the file

$$\frac{\text{file size}}{\text{block size}} = \#\ \text{of data blocks}$$

② 3. List (in order) the sequence of block reads that would be required to read the entirety of a 512 byte file `/2023/105/solutions.txt`. Assume that 2023 is the tenth entry in the root directory, that 105 is the second entry in the 2023 directory, and that `solutions.txt` is the first entry in the 105 directory (although of course the OS doesn't know this in advance).

512/64 => 8 data blocks

1. Root dir inode (2)
2. 1st block of /
5. 2nd block of /
4. 2023 inode
5. 1st block of 2023
6. 105 inode
7. 1st block of 105
8. solutions.txt inode
9. }
10. }
11. } 6 direct blocks of solutions.txt
12. }
13. }
14. }
15. indirect block for solutions.txt
16. } 2 indirect data blocks of solutions.txt
17. }

synchronization rules

int global var
lock lock
cv name of condition

* acquire(lock) - acquire lock
* release(lock) - release the lock
* signal(cv) - signal a thread to advance by one
* broadcast(cv) - signal all threads to advance
* wait(cv, lock) - make a thread wait on a waitlist
  * wait() in a while(condition)
  condition - things that need to be met

① acquire(lock) before and release(lock) after global variable

② write the while() condition before the global variable is decremented or incremented.

```
acqq()
while condition:
    wait()

global variable ++ or --
signal()
broadcast()  >or
release()
```

③ Decide if you need to signal or broadcast after global variable

```
int people_dining_at_table = 0;
Lock lock
CV polite_to_leave


void student(){


    acquire(lock)
    people_dining_at_table++;
    signal(polite_to_leave)
    release(lock)



    dine();



    acquire(lock)
    people_dining_at_table--;
    signal(polite_to_leave)
    while(people_at_table == 1){
        wait(polite_to_leave)
    }


    leave();
}
```