Contents

1 Integer

- Interpretation of binary representations

$$B2U_w(\overrightarrow{x}) = \sum_{i=0}^{w-1} x_i 2^i \qquad UMin_w = 0 \qquad UMax_w = 2^w - 1$$
$$B2T_w(\overrightarrow{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \quad TMin_w = -2^{w-1} \quad TMax_w = 2^{w-1} - 1$$

- Transformation between unsigned and signed

$$T2U_w(x) = \begin{cases} x & 0 \le x \le TMax_w \\ 2^w + x & TMin_w \le x < 0 \end{cases}$$
$$U2T_w(u) = \begin{cases} u & 0 \le u \le TMax_w \\ u - 2^w & TMax_w < u \le UMax_w \end{cases}$$

- Expansion

$$\overrightarrow{u} = [u_{w-1}, \ldots, u_0] \to \overrightarrow{u}' = [0, \ldots, 0, u_{w-1}, \ldots, u_0] \qquad B2U_w(\overrightarrow{u}) = B2U_{w'}(\overrightarrow{u}')$$
$$\overrightarrow{x} = [x_{w-1}, \ldots, x_0] \to \overrightarrow{x}' = [x_{w-1}, \ldots, x_{w-1}, x_{w-1}, \ldots, x_0] \quad B2T_w(\overrightarrow{x}) = B2T_{w'}(\overrightarrow{x}')$$

- Truncation

$$\overrightarrow{u} = [u_{w-1}, \ldots, u_0] \to \overrightarrow{u}' = [u_{k-1}, \ldots, u_0] \quad B2U_k(\overrightarrow{u}') = B2U_w(\overrightarrow{u}) \mod 2^k$$
$$\overrightarrow{x} = [x_{w-1}, \ldots, x_0] \to \overrightarrow{x}' = [x_{k-1}, \ldots, x_0] \quad B2T_k(\overrightarrow{x}') = U2T_k\left(B2U_w(\overrightarrow{x}) \mod 2^k\right)$$

- Addition

$$x +_w^u y = \begin{cases} x + y & 0 \le x + y < 2^w \\ x + y - 2^w & 2^w \le x + y \le 2^{w+1} - 2 \end{cases} \qquad 0 \le x, y \le 2^w - 1$$
$$x +_w^t y = \begin{cases} x + y + 2^w & -2^w \le x + y < -2^{w-1} \\ x + y & -2^{w-1} \le x + y \le 2^{w-1} - 1 \\ x + y - 2^w & 2^{w-1} \le x + y \le 2^w - 2 \end{cases} \qquad -2^{w-1} \le x, y \le 2^{w-1} - 1$$

- Negation

$$-_w^t x = \begin{cases} TMin_w & x = TMin_w \\ -x & TMin_w < x \le TMax_w \end{cases}$$

- Multiplication

$$x *_w^u y = (x \cdot y) \mod 2^w \qquad u \ll k = u *_w^u 2^k$$
$$x *_w^t y = U2T_w((x \cdot y) \mod 2^w) \qquad x \ll k = x *_w^t 2^k$$

- Division

$$x \gg k = \lfloor x/2^k \rfloor$$
$$(x + (1 \ll k) - 1) \gg k = \lceil x/2^k \rceil$$
$$x/y \equiv \begin{cases} \lfloor x/y \rfloor & x \ge 0, y > 0 \\ \lceil x/y \rceil & x < 0, y > 0 \end{cases}$$

2 Floating Point Number

- Definition $V = (-1)^s \times M \times 2^E$. 1-bit s encodes the sign s, k-bit `exp` encodes the exponent E, and n-bit `frac` encodes the significand M. float: 1, 8, 23; double: 1, 11, 52.

  Normalized `exp` is neither 0 or $2^k - 1$ (all 1).
  - $E = e - Bias$, in which $Bias = 2^{k-1} - 1$.
  - $M = 1.f_{n-1} \dots f_1 f_0$.

  Denormalized `exp = 0`
  - $E = 1 - Bias$.
  - $M = 0.f_{n-1} \dots f_1 f_0$.

  Infinity `exp`=$2^k - 1$ (all 1), `frac` = 0.

  NaN `exp`=$2^k - 1$ (all 1), `frac` $\neq 0$.

- Use round to even.

- $x +^f y \equiv Round(x+y)$. FP addition lacks associativity but features monotonicity: $x+a \geq x+b$ if $a \geq b$.

- $x *^f y \equiv Round(x \times y)$. FP Multiplication is not associative, and it does not distribute over addition. It features monotonicity:

$$a \geq b, c \geq 0 \Rightarrow a *^f c \geq b *^f c$$
$$a \geq b, c \leq 0 \Rightarrow a *^f c \leq b *^f c$$
$$a \neq NaN \Rightarrow a *^f a \geq 0$$

- Type conversions:
  - `int` to `float`: no overflow. Possible to be rounded.
  - `int/float` to `double`: precise conversion.
  - `double` to `float`: possible to overflow to infinity. Possible to be rounded.
  - `float/double` to `int`: round to 0. Possible to overflow.

3 x86-64 Assembly Language

3.1 Registers & operand indicators

Table 1: x86-64 registers & operand indicators

63	31	15	7	function
%rax	%eax	%ax	%al	return value
%rbx	%ebx	%bx	%bl	preserved by callee
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	preserved by callee
%rsp	%esp	%sp	%spl	stack pointer
%r8	%r8d	%r8w	%r8b	5th parameter
%r9	%r9d	%r9w	%r9b	6th parameter
%r10	%r10d	%r10w	%r10b	preserved by caller
%r11	%r11d	%r11w	%r11b	preserved by caller
%r12	%r12d	%r12w	%r12b	preserved by callee
%r13	%r13d	%r13w	%r13b	preserved by callee
%r14	%r14d	%r14w	%r14b	preserved by callee
%r15	%r15d	%r15w	%r15b	preserved by callee

$\$Imm$	Immediate number	Imm
r_a	Value of register r_a	$R[r_a]$
$Imm(r_b, r_s, s)$	Value at memory	$M[Imm+R[r_b]+R[r_s]*s]$, $s = 1, 2, 4, 8$

3.2 Data movement

- Direct move: `MOV S D`, `D`←`S`. `S` is immediate number, register or memory position. `D` is register or memory position. `S,D` cannot both be memory positions.
- Move with zero expansion: `MOVZ S R`, `R`←`Zero expansion(S)`. `S` can be register or memory position. `D` must be register. There is no `movzlq` because `movl` can set upper bits to 0, which is equivalent to zero expansion from 32 bits to 64 bits.
- Move with sign expansion: `MOVS S R`, `R`←`Sign expansion(S)`. `S` can be register or memory position. `D` must be register.
- Push/pop stack: push 4 words (64 bits) on or pop 4 words from the stack. Special cases: `pushq %rsp` pushes the original value of `%rsp` on the stack; `popq %rsp` puts the value read from memory in `%rsp`.

Table 2: Data movement instructions

`movb`	Move byte (1B = 8bit, char)
`movw`	Move word (2B = 16bit, short)
`movl`	Move long word (4B = 32bit, int). Also set upper 32 bits of the register to 0
`movq`	Move quad word (8B = 64bit, long, pointer). When `S` is immediate number, only expansion of 32-bit 2's component can be used. For 64-bit immediate value, use `movabsq`
`movabsq`	move 64-bit immediate number to register
`movzbw`	byte→word (1B→2B)
`movzbl`	byte→long word (1B→4B)
`movzbq`	byte→quad word (1B→8B)
`movzwl`	word→long word (2B→4B)
`movzwq`	word→quad word (2B→8B)
`movsbw`	byte→word (1B→2B)
`movsbl`	byte→long word (1B→4B)
`movsbq`	byte→quad word (1B→8B)
`movswl`	word→long word (2B→4B)
`movswq`	word→quad word (2B→8B)
`movslq`	long word→quad word (4B→8B)
`cltq`	`%rax←Sign expansion(%eax)`, i.e. `movslq %eax %rax`
`pushq S`	`R[%rsp]←R[%rsp]-8; M[R[%rsp]]←S`
`popq D`	`D←M[R[%rsp]]; R[%rsp]←R[%rsp]+8`

3.3 Arithmetic & logical operations

- Load address (only 1 version, `q`)
- Unary operations, binary operations, bitwise shifts (4 versions, `bwlq`)
- 128-bit integer manipulation

Table 3: Arithmetic & logical operation instructions

`leaq S D`	`D ← &S`		
`inc D`	`D ← D + 1`	`dec D`	`D ← D - 1`
`neg D`	`D ← -D`	`not D`	`D ← ~D`
`add S D`	`D ← D + S`	`sub S D`	`D ← D - S`
`imul S D`	`D ← D * S`	`or S D`	`D ← D | S`
`and S D`	`D ← D & S`		
`sal k D`	`D ← D ≪ k`	`shl k D`	`D ← D ≪ k`

to be continued

continue sar k D	D ← D $\gg_A$ k	shr k D	D ← D $\gg_L$ k
imulq S	R[%rdx]:R[%rax] ← S × R[%rax]		signed multiplication
mulq S	R[%rdx]:R[%rax] ← S × R[%rax]		unsigned multiplication
cqto	R[%rdx]:R[%rax] ← Signed expansion (R[%rax])		4 words to 8 words
idivq S	R[%rdx] ← R[%rdx]:R[%rax] mod S R[%rax] ← R[%rdx]:R[%rax] ÷ S		signed division
divq S	R[%rdx] ← R[%rdx]:R[%rax] mod S R[%rax] ← R[%rdx]:R[%rax] ÷ S		unsigned division

3.4 Control flow

- All arithemtic & logical operations except `leaq` can make changes to condition codes `CF, ZF, SF, OF`.
- `cmp` and `test` instructions set the condition codes without changing values of registers. Both have 4 versions(`bwlq`).
- `set` instructions can set a **byte** to 0 or 1 according to different combinations condition codes.
- `jump` instructions can make the execution jump to a specified position according to different combinations of condition codes
- `cmov` (conditional move) instructions can move the value at the source (memory position or register) to the destination register. They can be applied to 16, 32 or 64 bits (i.e. single byte conditional move is not supported!).
- For `set, jump` and `cmov` instructions, g/l (greater/less) are for signed integers, while a/b (above/below) are for unsigned integers.

Table 4: Condition codes & control flow instructions

CF	carry	Unsigned overflow (carry at highest bit).		
ZF	zero	Most recent result is 0.		
SF	sign	Most recent result is negative.		
OF	overflow	Complement overflow (+ or -).		
cmp S_1 S_2		Change condition codes according to $S_2 - S_1$.		
test S_1 S_2		Change condition codes according to S_1&S_2.		
	jmp		1	Unconditional jump
sete setz	je jz	cmove cmovz	==	ZF
setne setnz	jne jnz	cmovne cmovnz	!=	~ZF
sets	js	cmovs	negative	SF
setns	jns	cmovns	not negative	~SF
setg setnle	jg jnle	cmovg cmovnle	>	~(SF^OF) & ~ZF

to be continued

continue

setge setnl	jge jnl	cmovge cmovnl	>=	~(SF^OF)
setl setnge	jl jnge	cmovl cmovnge	<	SF^OF
setle setng	jle jng	cmovle cmovng	<=	(SF^OF) | ZF
seta setnbe	ja jnbe	cmova cmovnbe	>	~CF & ~ZF
setae setnb	jae jnb	cmovae cmovnb	>=	~CF
setb setnae	jb jnae	cmovb cmovnae	<	CF
setbe setna	jbe jna	cmovbe cmovna	<=	CF | ZF

Using `jump` and `cmov` instructions, we can translate C structs into structures easier to implement with assembly language.

Table 5: Translation of C constructs

C construct	Assembly code logic	Implementation details

`if (test-expr)`
`  then-statement`
`else`
`  else-statement`
	`t = test-expr;`
	`if (!t)`
	`  goto false;`
	`  then-statement`
	`  goto done;`
	`false:`
	`  else-statement`
	`done:`
		Use `jump` instructions.

`if (test-expr)`
`  then-statement`
`else`
`  else-statement`
	`t = test-expr;`
	`v = then-statement;`
	`ve = else-statement;`
	`if(!t) v = ve;`
		Use `cmov` instructions. Typically only when both statements are easy to calculate and have no side effect.

`do`
`  body-statement`
`  while (test-expr);`
	`loop:`
	`  body-statement;`
	`  t = test-expr;`
	`  if(t) goto loop;`
		Use `jump` instructions.

`while (test-expr)`
`  body-statement;`
	`  goto test;`
	`loop:`
	`  body-statement;`
	`test:`
	`  t = test-expr;`
	`  if(t) goto loop;`
		Use `jump` instructions.

to be continued

continue

`while (test-expr)`
`  body-statement;`
	`  t = test-expr;`
	`  if(!t) goto done;`
	`loop:`
	`  body-statement;`
	`  t = test-expr;`
	`  if(t) goto loop;`
	`done:`
		Use `jump` instructions.

`for(init-expr;`
`  test-expr;`
`  update-expr)`
`  body-statement;`
	`init-expr;`
	`while(test-expr) {`
	`  body-statement;`
	`update:`
	`  update-expr;`
	`}`
		Use `jump` instructions. `update` is useful only when `body-statement` contains `continue`.

`switch(n) {`
`case 100:`
`  statement-0; break;`
`case 101:`
`  statement-1;`
`case 103:`
`case 104:`
`  statement-4; break;`
`default:`
`  statement-default;`
`}`
	`static void *jt[5] = {`
	`  &&loc_0,&&loc_1,`
	`  &&loc_def,&&loc_34,`
	`  &&loc_34`
	`};`
	`unsigned long i = n - 100;`
	`if(i > 4) goto loc_def;`
	`goto *jt[i];`
	`loc_0:`
	`  statement-0; goto done;`
	`loc_1:`
	`  statement-1;`
	`loc_34:`
	`  statement-4; goto done;`
	`loc_def:`
	`  statement-default;`
	`done:`
		1. && (pointer to code location) is an extension defined by GCC.
		2. `unsigned long` handles the case of n < 100 (n - 100 overflows to a large integer.)

3.5 Procedure

Procedure is an important abstraction having different forms: function, subroutine, method, handler, etc. Each procedure has its own stack frame. For most procedures, stack frames are aligned to 16.

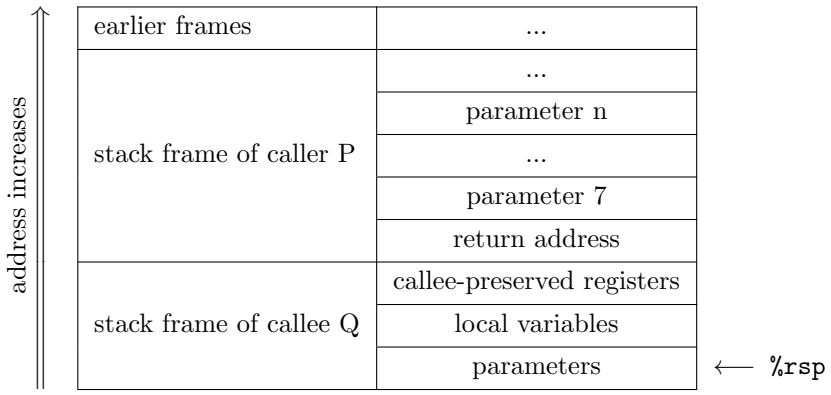


Figure 1: Stack frame structure

3.5.1 Transferring control

Return address is the address of the instruction after `call`.

Table 6: Control transfer instructions

`call`	Push return address onto stack; Set PC(`%rip`) to starting address of callee.
`ret`	Pop return address off stack; Set PC(`%rip`) to return address.

3.5.2 Passing arguments

- Arguments 1-6 are respectively put inside registers `%rdi, %rsi, %rdx, %rcx, %r8, %r9` or their counterparts of smaller sizes.
- Other arguments are on the stack, with argument 7 at the top. All on-stack arguments are aligned to 8.
- Argument k (k>6) is at address `%rsp+8*(k-6)`.

3.5.3 Local storage

Local data needs to be stored on stack in the following cases:

- registers are not enough to hold all local data;
- the & operator is used on a variable;
- arrays or structs are used as local variables.

On-stack local variables do not have to be aligned to 8. In general, basic variables (integers, pointers) of size K bytes should be aligned to K.

Callee-preserved registers (%rbx, %rbp, %r12-%r15) should be saved on stack before being used inside the callee. Other registers are caller-preserved, i.e. if the caller expects their values to be available after calling a procedure, the caller should save them on stack before calling the procedure.

3.6 Floating point instructions

- 16 `%ymm` registers, each of 256 bits, are used to store floating pointer numbers.
- When dealing with scalar FP numbers, we use `%xmm` registers, i.e. lowest 128 bits of `%ymm` registers to store them. Only the lowest 32 (float) or 64 (double) bits are used.
- `%xmm0` is used to store the FP return value.
- `%xmm0-7` are used to store 1st-8th FP arguments.
- `%xmm8-15` are caller-preserved registers.
- `aps` = aligned packed single, `apd` = aligned packed double.
- `vcvttss2si` = Vector ConVerT Truncation Scalar Single-precision 2 Signed Int.
- Floating point comparison sets 3 condition codes: `CF, ZF` and `PF` (`P` = parity). If at least one of the two arguments is `NaN`, then there is no order, and `PF` is set to 1.

Table 7: Condition codes of FP comparison

	CF	ZF	PF
No order	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

Table 8: Floating point instructions

`vmovss`	M_{32}		X	float, memory → register
`vmovss`	X		M_{32}	float, register → memory
`vmovsd`	M_{64}		X	double, memory → register
`vmovsd`	X		M_{32}	double, register → memory
`vmovaps`	X		X	float, register → register.
`vmovapd`	X		X	double, register → register.
`vcvttss2si`	X/M_{32}		R_{32}	float → int.
`vcvttsd2si`	X/M_{64}		R_{32}	double → int.
`vcvttss2siq`	X/M_{32}		R_{64}	float → long.
`vcvttsd2siq`	X/M_{64}		R_{64}	double → long.
`vcvtsi2ss`	M_{32}/R_{32}	X	X	int → float.
`vcvtsi2sd`	M_{32}/R_{32}	X	X	int → double.
`vcvtsi2ssq`	M_{64}/R_{64}	X	X	long → float.
`vcvtsi2sdq`	M_{64}/R_{64}	X	X	long → double.
`vunpcklps %xmm0 %xmm0 %xmm0`				float → double (weird but it is what
`vcvtps2pd %xmm0 %xmm0`				gcc does)

to be continued

continue	
`vmovddup %xmm0 %xmm0` `vcvtpd2psx %xmm0 %xmm0`	double $\rightarrow$ float(weird but it is what gcc does)
`vaddss vaddsd`	$D \leftarrow S_2 + S_1$
`vsubss vsubsd`	$D \leftarrow S_2 - S_1$
`vmulss vmulsd`	$D \leftarrow S_2 \times S_1$
`vdivss vdivsd`	$D \leftarrow S_2 \div S_1$
`vmaxss vmaxsd`	$D \leftarrow \max(S_2, S_1)$
`vminss vminsd`	$D \leftarrow \min(S_2, S_1)$
`sqrtss sqrtsd`	$D \leftarrow \sqrt{S_1}$
`vxorps vxorpd`	$D \leftarrow S_2 \hat{} S_1$
`vandps vandpd`	$D \leftarrow S_2 \& S_1$
`ucomiss`	compare float according to $S_2 - S_1$
`ucomisd`	compare double according to $S_2 - S_1$

4 Processor Design & Implementation

4.1 Y86-64 ISA

Table 9: Y86-64 instructions

Instruction	0		1		2 - 8	9
halt	0	0				
nop	1	0				
rrmovq rA rB	2	0	rA	rB		
irmovq V rB	3	0	F	rB	V	
rmmovq rA D(rB)	4	0	rA	rB	D	
mrmovq D(rB) rA	5	0	rA	rB	D	
OPq rA rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
cmovXX rA rB	2	fn	rA	rB		
call Dest	8	0	Dest			
ret	9	0				
pushq rA	A	0	rA	F		
popq rA	B	0	rA	F		

Table 10: Function codes of Y86-64 instructions

addq	6	0	subq	6	1	andq	6	2	xorq	6	3
jmp	7	0	jle	7	1	jl	7	2	je	7	3
jne	7	4	jge	7	5	jg	7	6			
rrmovq	2	0	cmovle	2	1	cmovl	2	2	cmove	2	3
cmovne	2	4	cmovge	2	5	cmovg	7	6			

Table 11: Registers in Y86-64 ISA

%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
0	1	2	3	4	5	6	7
%r8	%r9	%r10	%r11	%r12	%r13	%r14	None
8	9	A	B	C	D	E	F

4.2 SEQ implementation of Y86-64

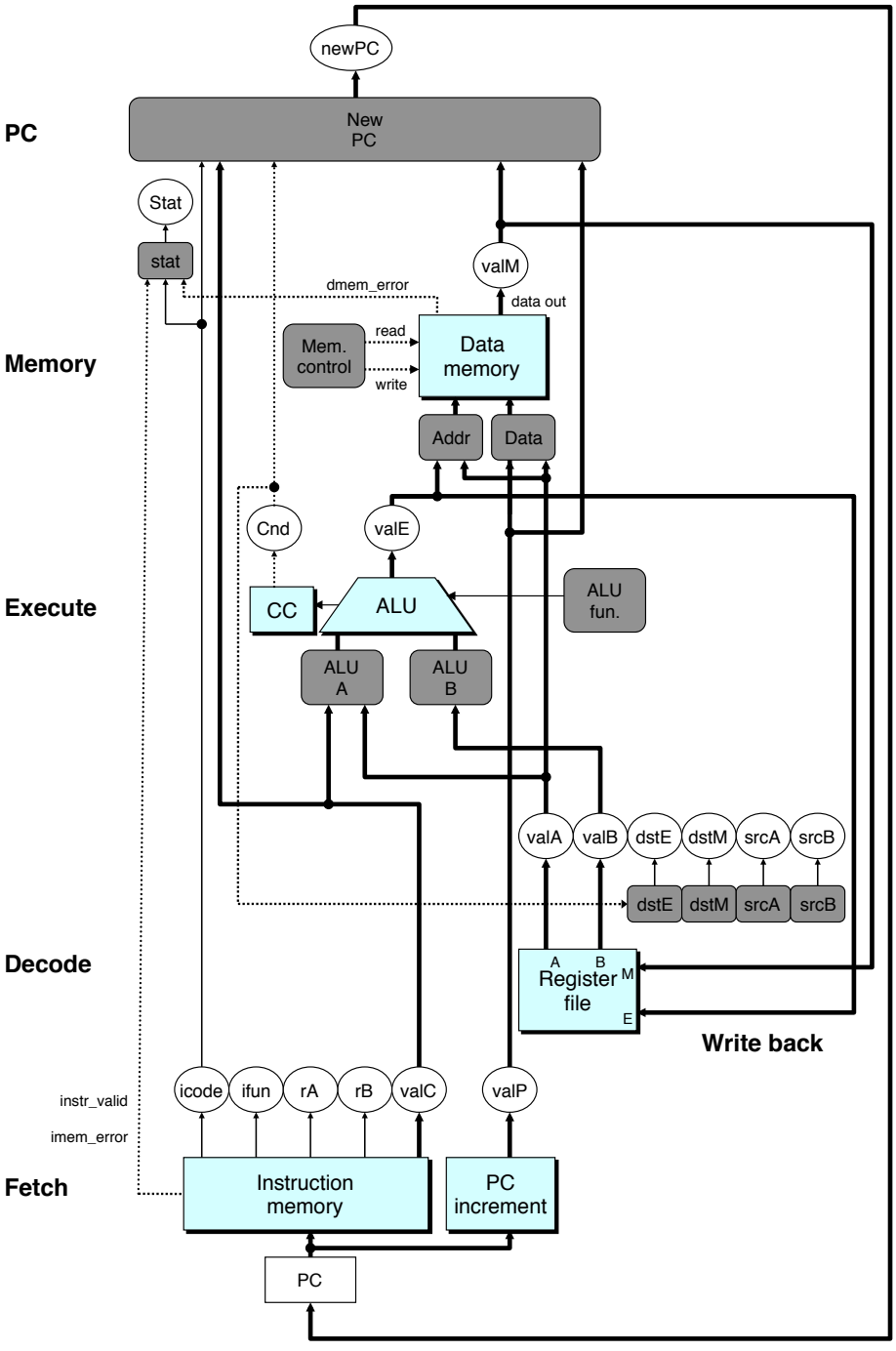


Figure 2: Sequential implementation of Y86-64

Table 12: Sequential implementation of Y86-64

Stage	Fetch	Decode	Execute	Memory	Write Back	PC Update
OPq rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA] valB ← R[rB]	valE ← valB OP valA Set CC		R[rB] ← valE	PC ← valP
rrmovq rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA]	valE ← 0+valA		R[rB] ← valE	PC ← valP
irmovq V rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10		valE ← 0+valC		R[rB] ← valE	PC ← valP
rmmovq rA D(rB)	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valA ← R[rA] valB ← R[rB]	valE ← valB+valC	M_8[valE] ← valA		PC ← valP
mrmovq D(rB) rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valB ← R[rB]	valE ← valB+valC	valM ← M_8[valE]	R[rA] ← valM	PC ← valP
pushq rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA] valB ← R[%rsp]	valE ← valB+(-8)	M_8[valE] ← valA	R[%rsp] ← valE	PC ← valP
popq rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[%rsp] valB ← R[%rsp]	valE ← valB+8	valM ← M_8[valA]	R[%rsp] ← valE R[rA] ← valM	PC ← valP
jXX Dest	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9		Cnd ← Cond(CC, ifun)			PC ← Cnd ? valC : valP
cmoveXX rA rB	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	valA ← R[rA]	valE ← 0+valA Cnd ← Cond(CC, ifun)		if(Cnd) R[rB] ← valE	PC ← valP
call Dest	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	valB ← R[%rsp]	valE ← valB+(-8)	M_8[valE] ← valP	R[%rsp] ← valE	PC ← valC
ret	icode:ifun ← M_1[PC] valP ← PC+1	valA ← R[%rsp] valB ← R[%rsp]	valE ← valB+8	valM ← M_8[valA]	R[%rsp] ← valE	PC ← valM

The following design is taken from both the book and the web aside.

- Fetch

```
word icode = [
  imem_error : INOP;
  1 : imem_icode;
];
word ifun = [
  imem_error : FNONE;
  1 : imem_ifun;
];
bool instr_valid = icode in {IHALT, INOP, IIRMOVQ, IMRMOVQ, IRMMOVQ, IRRMOVQ,
    ICALL, IRET, IPUSHQ, IPOPQ, IOPQ, IJXX};
bool need_regids = icode in {IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ,
    IMRMOVQ};
bool need_valC = icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ, ICALL, IJXX};
```

  valP = PC + 1 + need_regids + 8 * need_valC.

- Decode & Write Back

```
word srcA = [
  icode in {IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ} : rA;
  icode in {IPOPQ, IRET} : RRSP;
  1 : RNONE;
];
word srcB = [
  icode in {IOPQ, IRMMOVQ, IMRMOVQ} : rB;
  icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1 : RNONE;
];
word dstE = [
  icode in {IOPQ, IIRMOVQ} : rB;
  icode == IRRMOVQ && Cnd : rB;
  icode in {ICALL, IRET, IPUSHQ, IPOPQ} : RRSP;
  1 : RNONE;
];
word dstM = [
  icode = {IPOPQ, IMRMOVQ} : rA;
  1 : RNONE;
];
```

- Execute

```
word aluA = [
  icode in {IOPQ, IRRMOVQ} : valA;
  icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : valC;
  icode in {IPUSHQ, ICALL} : -8;
  icode in {IPOPQ, IRET} : 8;
];
word aluB = [
  icode in {IOPQ, IRMMOVQ, IMRMOVQ, IPUSHQ, IPOPQ, ICALL, IRET} : valB;
  icode in {IIRMOVQ, IRRMOVQ} : 0;
];
word alufun = [
  icode == IOPQ: ifun;
  1 : ALUADD;
```

```
];
bool set_cc = icode in {IOPQ};
```

- Memory

```
bool mem_read = icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = icode in {IRMMOVQ, IPUSH, ICALL};
word mem_addr = [
  icode in {IRMMOVQ, IMRMOVQ, IPUSHQ, ICALL} : valE;
  icode in {IPOPQ, IRET} : valA;
];
word mem_data = [
  icode in {IPUSHQ, IRMMOVQ} : valA;
  icode == ICALL : valP;
];
word Stat = [
  imem_error || dmem_error : SADR;
  !instr_valid : SINS;
  icode == IHLT : SHLT;
  1 : SAOK;
];
```

- PC Update

```
word new_pc = [
  icode == ICALL : valC;
  icode == IJXX && Cnd : valC;
  icode == IRET : valM;
  1 : valP;
];
```

4.3 Pipeline implementation of Y86-64

- Fetch & PC Selection

```
word f_pc = [
  M_icode == IJXX && !M_Cnd : M_valA; //When jump PC prediction is wrong
  W_icode == IRET : W_valM;
  1 : F_predPC;
];
word f_predPC = [
  f_icode in {IJXX, ICALL} : f_valC; //Strategy for jump PC prediction: always
      jump
  1 : f_valP;
];
word f_stat = [
  imem_error : SADR; // dmem_error in memory phase
  !instr_valid : SINS;
  f_icode == IHLT : SHLT;
  1 : SAOK;
];

// similar to SEQ
word f_icode = [
  imem_error : INOP;
  1 : imem_icode;
];
```

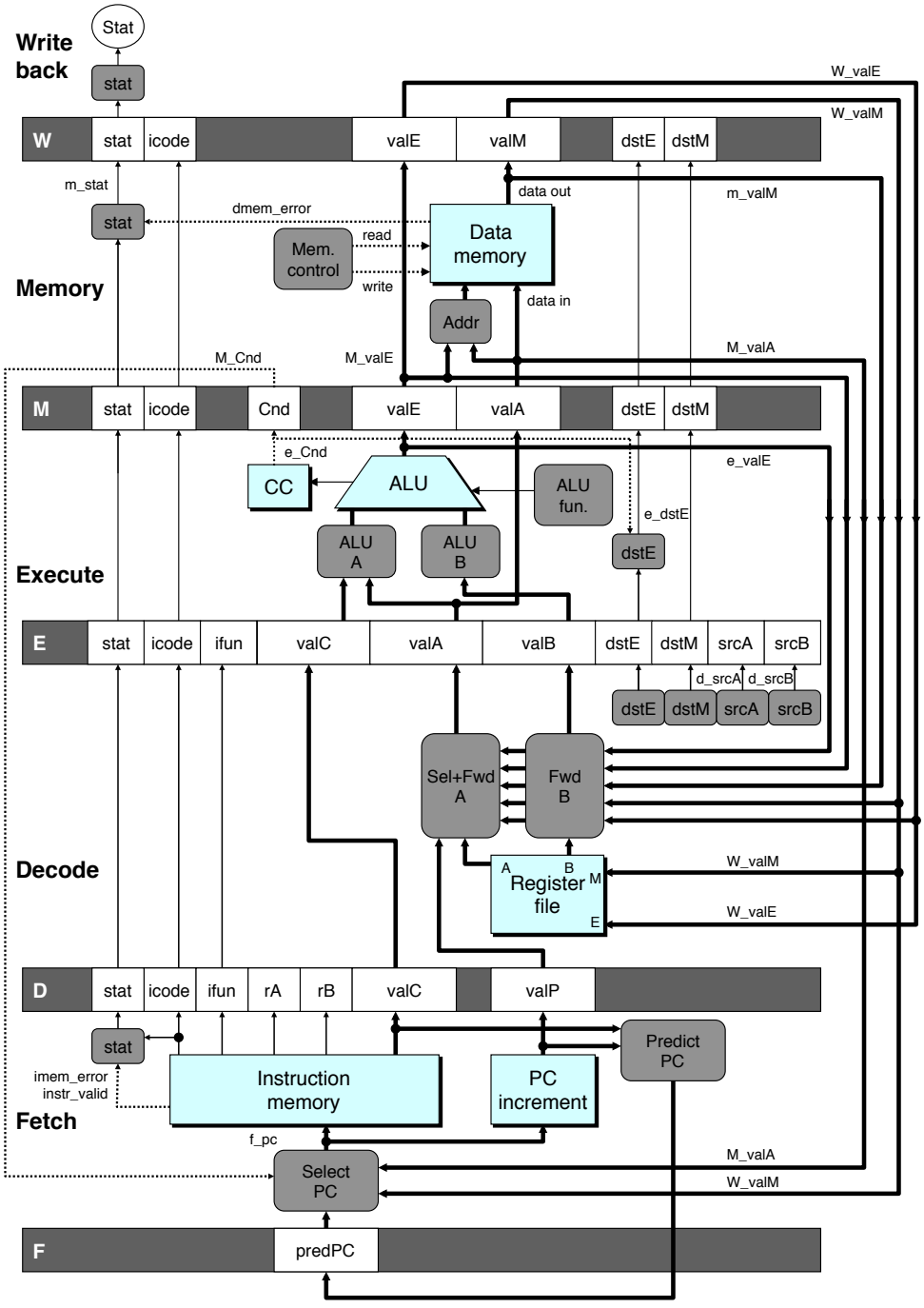


Figure 3: Pipeline implementation of Y86-64

```
word f_ifun = [
  imem_error : FNONE;
  1 : imem_ifun;
];
bool instr_valid = f_icode in {IHALT, INOP, IIRMOVQ, IMRMOVQ, IRMMOVQ, IRRMOVQ,
     ICALL, IRET, IPUSHQ, IPOPQ, IOPQ, IJXX};
bool need_regids = f_icode in {IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ,
    IMRMOVQ};
bool need_valC = f_icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ, ICALL, IJXX};
```

- Decode & Write Back

  Precedence of forwarding sources:

  – Forward data affected by the nearest instruction. Thus E > M > W.

  – For correct behavior of `popq %rsp`, M > E.

```
word d_valA = [
  D_icode in {ICALL, IJXX} : D_valP;
  d_srcA == e_dstE : e_valE;
  d_srcA == M_dstM : m_valM;
  d_srcA == M_dstE : M_valE;
  d_srcA == W_dstM : W_valM;
  d_srcA == W_dstE : W_valE;
  1 : d_rvalA;
];
word d_valB = [
  d_srcB == e_dstE : e_valE;
  d_srcB == M_dstM : m_valM;
  d_srcB == M_dstE : M_valE;
  d_srcB == W_dstM : W_valM;
  d_srcB == W_dstE : W_valE;
  1 : d_rvalB;
];
word w_dstE = W_dstE;
word w_dstM = W_dstM;
word w_valE = W_valE;
word w_valM = W_valM;
word Stat = [
  W_stat = SBUB : SAOK;
  1 : W_stat;
];
// similar to SEQ
word d_srcA = [
  D_icode in {IOPQ, IRMMOVQ, IRRMOVQ, IPUSHQ} : D_rA;
  D_icode in {IRET, IPOPQ} : RRSP;
  1 : RNONE;
];
word d_srcB = [
  D_icode in {IOPQ, IRMMOVQ, IMRMOVQ} : D_rB;
  D_icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1 : RNONE;
];
word d_dstE = [
  D_icode in {IOPQ, IIRMOVQ} : rB;
  D_icode in {ICALL, IRET, IPUSHQ, IPOPQ} : RRSP;
  1 : RNONE;
```

```
];
word d_dstM = [
  D_icode = {IPOPQ, IMRMOVQ} : D_rA;
  1 : RNONE;
];
```

- Execute

```
bool set_cc = E_icode == IOPQ && !m_stat in {SADR, SHLT, SINS} && !W_stat in {
    SADR, SHLT, SINS};
word e_valA = E_valA;
word e_dstE = [
  E_icode == IRRMOVQ && !e_Cnd : RNONE;
  1 : E_dstE;
];
// similar to SEQ
word aluA = [
  E_icode in {IOPQ, IRRMOVQ} : E_valA;
  E_icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : E_valC;
  E_icode in {IPUSHQ, ICALL} : -8;
  E_icode in {IPOPQ, IRET} : 8;
];
word aluB = [
  E_icode in {IOPQ, IRMMOVQ, IMRMOVQ, IPUSHQ, IPOPQ, ICALL, IRET} : E_valB;
  E_icode in {IIRMOVQ, IRRMOVQ} : 0;
];
word alufun = [
  E_icode == IOPQ: E_ifun;
  1 : ALUADD;
];
```

- Memory

```
word m_stat = [
  dmem_error : SADR;
  1 : M_stat;
];
// similar to SEQ
bool mem_read = M_icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = M_icode in {IRMMOVQ, IPUSH, ICALL};
word mem_addr = [
  M_icode in {IRMMOVQ, IMRMOVQ, IPUSHQ, ICALL} : M_valE;
  M_icode in {IPOPQ, IRET} : M_valA;
];
```

4.4 Special pipeline control logic

```
bool F_bubble = 0;
bool F_stall = E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB} ||
  IRET in {D_icode, E_icode, M_icode};
bool D_stall = E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB} ||;
bool D_bubble = E_code == IJXX && !e_Cnd ||
  IRET in {D_icode, E_icode, M_icode} && !(E_code in {IMRMOVQ, IPOPQ} && E_dstM in {
      d_srcA, d_srcB});
bool E_stall = 0;
bool E_bubble = E_code == IJXX && !e_Cnd ||
```

Table 13: Special pipeline control logic

	Condition	F	D	E	M	W
ret	`IRET ∈ {D_icode, E_icode, M_icode}`	stall	bubble	normal	normal	normal
load / use hazard	`E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srsA, d_srcB}`	stall	stall	bubble	normal	normal
mispredicted branch	`E_icode == IJXX && !e_Cnd`	normal	bubble	bubble	normal	normal
exception	`m_stat ∈ {SADR, SHLT, SINS} || W_stat ∈ {SADR, SHLT, SINS}`	normal	normal	normal	bubble	stall (only W)
ret & misprediction	jumps to `ret`	stall	bubble	bubble	normal	normal
ret & load / use	set `%rsp` followed by `ret`	stall	stall	bubble	normal	normal

```
  E_code in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB};
bool M_bubble = m_stat in {SADR, SHLT, SINS} ||
  W_stat in {SADR, SHLT, SINS};
bool M_stall = 0;
bool W_bubble = 0;
//Do not stall when m_stat is not OK. Otherwise the instruction in W (which does not
    cause exception) won't finish.
bool W_stall = W_stat in {SADR, SINS, SHLT};
```

5 Performance Optimization

5.1 Performance of arithmetic operations on the reference machine

Table 14: Performance & bound of arithmetic operations

Operation	Integer			Floating Number		
	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3-30	3-30	1	3-15	3-15	1
Bound	Integer			Floating Number		
	+	*		+	*	
Latency	1.00	3.00		3.00	5.00	
Throughput	0.50	1.00		1.00	0.50	

5.2 Techniques

Various techniques for performance optimization are illustrated on the following code.

```
#define IDENT 1 //#define IDENT 0
#define OP *    //#define OP +
typedef struct {
  long len;
  data_t *data;
} vec_rec, *vec_ptr;

int get_vec_element(vec_ptr v, long index, data_t *dest) {
  if(index < 0 || index > v->len) return 0;
  *dest = v->data[index];
  return 1;
}

long vec_length(vec_ptr v) { return v->len; }

void combine1(vec_ptr v, data_t *dest) {
  long i;
  *dest = IDENT;
  for(i = 0; i < vec_length(v); ++i) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

5.2.1 Code motion

```
void combine2(vec_ptr v, data_t *dest) {
  long i;
  (*@\color{red}long length = vec\_length(v);@*)
  *dest = IDENT;
  for(i = 0; i < (*@\color{red}length@*); ++i) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

5.2.2 Reduce procedure calls

```
(*@\color{red}data\_t *get\_vec\_start(vec\_ptr v)@*) { return v->data; }

void combine3(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  (*@\color{red}data\_t *data = get\_vec\_start(v);@*)
  *dest = IDENT;
  for(i = 0; i < length; ++i) {
    *dest = *dest OP (*@\color{red}data[i]@*);
  }
}
```

5.2.3 Eliminate unnecessary memory references

```
void combine4(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  (*@\color{red}data\_t acc = IDENT; @*)
  for(i = 0; i < length; ++i) {
    (*@\color{red}acc@*) = (*@\color{red}acc@*) OP data[i];
  }
  (*@\color{red}*dest = acc; @*)
}
```

5.2.4 Loop unrolling (2 * 1 unrolling)

```
void combine5(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  (*@\color{red}long limit = length - 1;@*)//length - k + 1 for k
  for(i = 0; i < (*@\color{red}limit@*); (*@\color{red}i+=2@*)) { //i+=k for k
    (*@\color{red}acc = (acc OP data[i]) OP data[i + 1];@*)
  }
  (*@\color{red}
  for(; i < length; ++i) acc = acc OP data[i];
  @*)
  *dest = acc;
}
```

5.2.5 Multiple accumulator (2 * 2 unrolling)

```
void combine6(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  long limit = length - 1;//length - k + 1 for k
  (*@\color{red}data\_t acc0 = IDENT;@*)
  (*@\color{red}data\_t acc1 = IDENT;@*)
  for(i = 0; i < limit; i+=2) { //i+=k for k
    (*@\color{red}acc0 = acc0 OP data[i];@*)
    (*@\color{red}acc1 = acc1 OP data[i + 1];@*)
  }
  for(; i < length; ++i) (*@\color{red}acc0@*) = (*@\color{red}acc0@*) OP data[i];
  *dest = (*@\color{red}acc0 OP acc1@*);
}
```

5.2.6 Reassociation (2 * 1a unrolling)

```
void combine5(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  long limit = length - 1;//length - k + 1 for k
  for(i = 0; i < limit; i+=2) { //i+=k for k
    acc = (*@\color{red}acc OP (data[i]) OP data[i + 1]@*);
  }
  for(; i < length; ++i) acc = acc OP data[i];
  *dest = acc;
}
```

5.3 Limitations

1. Register spilling. When the degree of parallelism p exceeds the number of registers, some temporaries will have to be stored on stack, which harms the performance of the program.

2. Penalty of wrong branch prediction. Usually we do not have to worry too much about the penalty. But we should write code suitable for conditional moves when possible.

Imperative Code	Functional Code

`if(a[i] > b[i]) {`
`  long t = a[i];`
`  a[i] = b[i];`
`  b[i] = t;`
`}`
	`long min = a[i] < b[i] ? : a[i] : b[i];`
	`long max = a[i] < b[i] ? b[i] : a[i];`
	`a[i] = min;`
	`b[i] = max;`

`if(src1[i1] < src2[i2])`
`  dest[id++] = src1[i1++];`
`else`
`  dest[id++] = src2[i2++];`
	`long v1 = src[i1], v2 = src[i2];`
	`bool choose1 = v1 < v2;`
	`dest[id++] = choose1 ? v1 : v2;`
	`i1 += choose1;`
	`i2 += (1 - choose1);`

5.4 Memory Performance

Try to avoid loading value from a memory position immediately after writing to it, which increases the length of the critical path. For example, this function:

```
void psum1(float a[], float p[], long n) {
  long i;
  p[0] = a[0];
  for(i = 1; i < n; i++) {
    p[i] = p[i - 1] + a[i];//Load from p[i-1], which was written in the last
        iteration.
  }
}
```

should be written as:

```
void psum1_better(float a[], float p[], long n) {
  long i;
  float val = a[0];
  p[0] = val;
  for(i = 1; i < n; i++) {
    val += a[i];
    p[i] = val;//val is in register, not in memory.
  }
}
```

6 Memory Hierarchy

6.1 Storage Technologies

RAM Random Access Memory. SRAM is faster (10×) but more expensive (1000×) than DRAM.

SRAM Static Random Access Memory. Each bit is stored in a bistable memory cell. It can stay indefinitely in one of the two voltage configurations. Insensitive to disturbance.

DRAM Dynamic Random Access Memory. Each bit is stored as charge on a capacitor. Sensitive to disturbance.

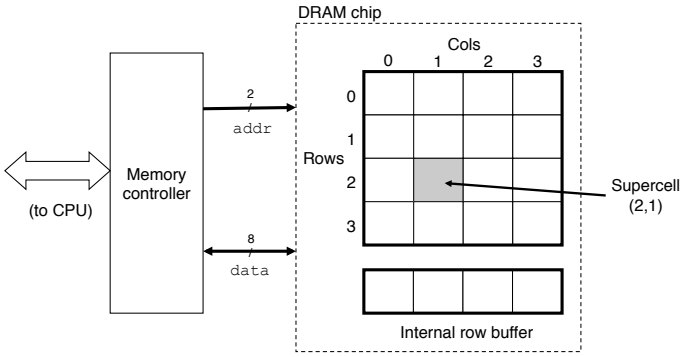


Figure 4: A 128 bit (16×8) DRAM chip

FPM DRAM Fast Page Mode DRAM.

EDO DRAM Extended Data Out DRAM.

SDRAM Synchronous DRAM.

DDR SDRAM Double Data-Rate SDRAM.

VRAM Video RAM.

RAS/CAS Row/Column Access Strobe. Row/Column address sent from the memory controller to the DRAM chip.

ROM Read-Only Memory. (Nonvolatile Memory).

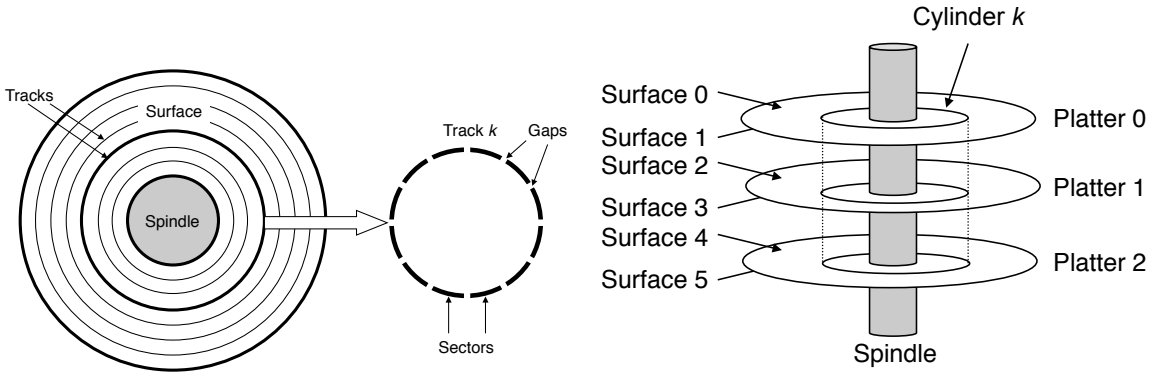
PROM Programmable ROM. Programmable only once.

EPROM Erasable PROM. Programmable ˜1000 times.

EEPROM Electrically Erasable PROM. Programmable ˜10^5 times. **Flash memory** is based on EEPROM. **SSD**(Solid State Disk) is based on flash memory.

Firmware Programs stored in ROM.

Bus Data flows between CPU and DRAM main memory through the buses. Each data transfer is called a **bus transaction**, either **read transaction** or **write transaction**.



(a) Single platter **(b) Multiple platters**

Figure 5: Disk geometry

Disk Platter, surface, spindle, cylinder, track, sector, gap. Rotation rate, RPM(Revolution Per Minute).

$$Capacity = \frac{Bytes}{Sector} \times \frac{Sectors}{Track} \times \frac{Tracks}{Surface} \times \frac{Surfaces}{Platter} \times \frac{Platters}{Disk}$$

Access time:

- Seek time: $T_{\text{avg seek}} = 3 \sim 9\,ms$.
- Rotational latency: $T_{\text{avg rotation}} = \frac{1}{2} \times \frac{1}{RPM} \times \frac{60s}{1min} \sim T_{\text{avg seek}}$
- Transfer time: $T_{\text{avg transfer}} = \frac{1}{RPM} \times \frac{1}{Sectors/Track} \frac{60s}{1min} \ll T_{\text{avg seek}}$.

Disk controller maintains the mapping between logical blocks and physical disk sectors.

I/O Bus Unrelated to CPU. Intel: PCI(Peripheral Component Interconnect) bus. Connects to:

- USB(Universal Serial Bus) controller: connects to USB devices.
- Graphics card/adapter.
- Host bus adapter: connects to one or more disk drives.

Memory mapped I/O CPU uses memory-mapped I/O to send instructions to I/O devices via I/O ports, i.e. a series of special addresses in the memory space.

DMA Direct Memory Access. I/O devices carry out read/write transactions without interference of CPU.

6.2 Locality & memory hierarchy

- Temporal locality & spacial locality
- Stride-k reference pattern: stride-1 reference pattern has the best spacial locality.

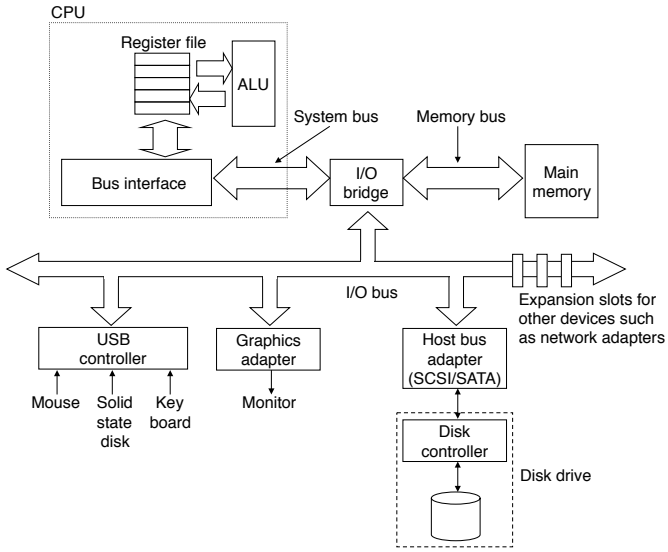


Figure 6: System bus, memory bus, IO bus

- Memory hierarchy & typical visit time: Register (0) - L1 SRAM cache (4) - L2 SRAM cache (10) - L3 SRAM cache (50) - Main DRAM memory (200) - Hard disk (10000000)

6.3 Cache memories

- m-bit address: t-bit tags + s-bit set index + b-bit block offset.
- Direct-mapped cache: $E = 1$. E-way set associative cache: $1 < E < C/B$. Full associative cache: $S = 1$.
- 3 steps: set selection; line matching; word extraction. In case of cache miss: line replacement.
- Cache thrash: cache keeps loading & evicting the same blocks.
- Handle write: write-through (directly write to lower layer) v.s. write-back (put off write until line gets replaced).
- Handle write cache miss: write-allocate (load from lower layer then update) v.s. not-write-allocate (directly write to lower layer).
- Usually: write-through + not-write-allocate; write-back + write-allocate.
- Write cache-friendly code: improve cache hit rate. For example, to write code for matrix multiplication $C = A \times B$, instead of straightforwardly writing:

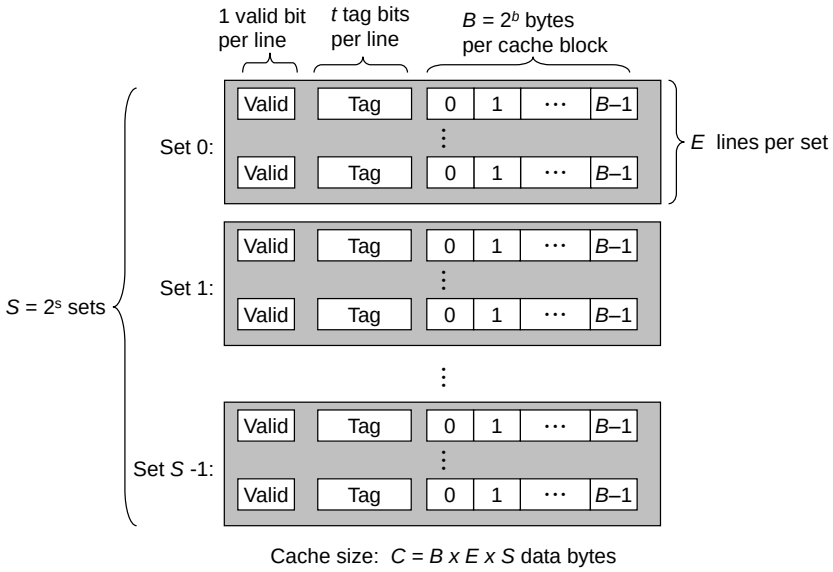


Figure 7: Organization of cache memory(S,E,B,m)

```
//ijk
for(i = 0; i < n; ++i)
  for(j = 0; j < n; ++j) {
    sum = 0.0;
    for(k = 0; k < n; ++k)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum;
  }
```

```
//kji
for(k = 0; k < n; ++k)
  for(j = 0; j < n; ++j) {
    r = B[k][j];
    for(i = 0; i < n; ++i)
      C[i][j] += A[i][k] * r;
  }
```

we should write:

```
//ikj
for(i = 0; i < n; ++i)
  for(k = 0; k < n; ++k) {
    r = A[i][k];
    for(j = 0; j < n; ++j)
      C[i][j] += r * B[k][j];
  }
```

Table 15: Analysis of different inner loops

version	load	store	A miss	B miss	C miss	total miss
ijk & jik	2(AB)	0	0.25	1.00	0.00	1.25
kji & jki	2(AC)	1(C)	1.00	0.00	1.00	2.00
ikj & kij	2(AC)	1(C)	0.00	0.25	0.25	0.50

7 Linking[1]

7.1 Basics

1. C compiler driver:

   cpp C preprocessor. *.c → *.i

   cc1 C compiler. *.i → *.s

   as assembler. *.s → *.o

   ld linker. *.o → exec

2. Task of linker:

   Symbol resolution Associate each symbol reference with one symbol definition.

   Relocation Associate a memory location with each symbol definition, and modify all references to the symbol so that they point to this memory location.

3. Object files:

   relocatable object file Binary code + data.

   executable object file Can be directly copied into memory and executed.

   shared object file A special type of relocatable object file. Can be loaded into memory and linked dynamically, at either load time or run time.

7.2 Relocatable object file

ELF header 16-byte sequence. Word size & byte ordering of the system. Information for linker to parse & interpreting the object file: size of ELF header; object file type; machine type; file offset, size, number of entries of the section header table.

section header table a fixed-sized entry for each section. Locations & sizes of the sections.

.text Machine code of the compiled program.

.rodata Read-only data.

.data Initialized non-zero global / static variables.

.bss Merely place holder (does not occupy actual space). Uninitialized or zero-initialized global / static variables.

symtab Symbol table.

.rel.text Relocation information. A list of locations in `.text` that need to be modified when the linker combines this object file with others. Any instruction that calls an external function or references a global variable needs to be modified.

[1] 7.8-7.14 omitted for the moment.

.rel.data Relocation information. Initialized global variable whose initial value is the address of a global variable or an externally defined function.

.debug A debugging symbol table only present with `-g` option.

.line A mapping between line numbers in the original C code and machine code instructions in `.text`.

.strtab A string table for the symbol tables in `.symtab` and `.debug`, and section names in the section headers.

7.3 Symbol table

global symbols Defined by this module and can be used by other modules. Non-static C functions & global variables.

external symbols C functions & global variables defined by other modules and used by this module.

local symbols C functions & variables defined and referenced exclusively by this module. `static` C functions & global variables.

Entry in symbol table:

```
typedef struct {
  int name;   /*String table offset. Name of the symbol.*/
  char type:4, /*Function or data (4 bits)*/
       binding:4; /*Local or global*/
  char reserved; /*Unused*/
  short section; /*Section header index*/
  long value; /*Address. Section offset (relocatable) or absolute address (
      executable)*/
  long size; /*Object size in bytes.*/
} Elf64_Symbol;
```

Pseudo sections:

ABS Should not be relocated

UNDEF Undefined symbols (referenced in this module but defined elsewhere).

COMMON Uninitialized data objects that are not yet allocated. In this case, `value` gives alignment requirement and `size` gives the minimum size. Difference from `.bss`:

- COMMON: uninitialized global variables.
- .bss: uninitialized static variables; global / static variables initialized to 0.

7.4 Symbol resolution

7.4.1 Multiply defined global variables

- Strong symbols: Functions; initialized variables.
- Weak symbols: Uninitialized variables.

Rules to resolve multiply defined global symbols:

- Multiple strong symbols are not allowed.
- Given a strong symbol and multiple weak symbols, choose the strong one.
- Given multiple weak symbols, choose any of them.

Error due to 2nd rule:

```c
/*foo.c*/
#include <stdio.h>
void f(void);
int y = 15212;
int x = 15313;
int main() {
  f();
  printf("x=0x%x, y=0x%x\n", x, y);
  return 0;
}

/*bar.c*/
double x;
void f() { x = -0.0; }
```

Running the program will give `x=0x0, y=0x80000000`.

7.4.2 Static libraries

During the symbol resolution phase, the linker scans the relocatable object files and archives (static libraries) from left to right according to their order in the command line. During the scan, the following sets are maintained:

E Relocatable object files to be merged into the executable.

U Unresolved symbols.

D Symbols defined in previous input files.

- Initially **E, U, D** are all empty.
- For each input file *f*, if *f* is an object file, the linker adds *f* to **E** and updates **U, D** according to symbol definitions and references in *f*.
- If *f* is an archive, the linker attempts to match symbols in **U** against symbols defined by the members of the archive. If a referenced symbol in **U** is defined in a member *m* of the archive, then *m* is added to **E** and **U, D** are updated accordingly. Repeat until **U, D** no longer changes. Members of the archive not contained in **E** are discarded.
- If **U** remains nonempty after the scan, the linker prints an error and terminates. Otherwise it merges all relocatable object files in **E** to build the executable.

This makes the order of the command line arguments important.

- Archives are put at the end of the command line.

- If archives are dependent, they need to be ordered so that for each symbol s externally referenced by a member of an archive, at least one definition of s follows a reference to s on the command line. For this purpose archives can be repeated on the command line.

7.5 Relocation

Relocation includes two steps:

- Relocate sections & symbol definitions
  - Merges all sections of the same type into an aggregate section (e.g. `.data`).
  - Assigns run-time memory address to each **aggregate section**, each **(original) section** defined by input modules and each **symbol** defined by input modules.
- Relocate symbol references within sections
  - Modifies symbol references in `.text` and `.data` so that they point to the correct run-time addresses according to the **relocation entries** in `.rel.text` and `.rel.data`.

When the assembler encounters an reference to an object whose ultimate address is unknown, a **relocation entry** is generated for this reference.

```
typedef struct {
  long offset; /*section offset of the reference to relocate*/
  long type:32, /*Relocation type. */
       symbol:32; /*Symbol table index*/
  long addend; /*Constant part of relocation expressions*/
} Elf64_Rela;
```

There are totally 32 relocation types. Basic types that support **small code model**(data + code $<$ 2GB):

- R_X86_64_PC32: for references using 32-bit PC-relative address.
- R_X86_64_32: for references using 32-bit absolute address.

The pseudo code for the relocation algorithm used by the linker:

```
for section s {
  for relocation entry r {
    refptr = s + r.offset; /*ptr to reference to be relocated*/

    if(r.type == R_X86_64_PC32) {
      refaddr = ADDR(s) + r.offset;
      *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr);
    } else if (r.type == R_X86_64_32)
      *refptr = (unsigned)(ADDR(r.symbol) + r.addend);
  }
}
```

8 Exception Control Flow

Understanding ECF can help us

- Understand important concepts: IO, process, virtual memory, etc.
- Understand how applications interact with the OS: writing data to disk, reading data from network, creating / terminating a process all involve a form of ECF known as **system call (trap)**.
- Write interesting new programs such as Unix shells and web servers, using ECF mechanisms provided by the OS.
- Understand concurrency.
- Understand how software exceptions work in C++, JAVA, etc.

8.1 Exception

8.1.1 Definition

Exception An abrupt change in the control flow in response to some change the processor's state.

Event Change in a processor's state (encoded in various bits and signals).

- Can be related to the execution of the current instruction: virtual memory page fault, arithmetic overflow, division by 0, etc.
- Or unrelated: system timer goes off, I/O request completion.

When the processor detects an event, it makes an indirect procedure call (the exception), through a jump table called the **exception table**, to an OS subroutine (the exception handler). When the exception handler finishes processing, it either returns control to the current instruction or the next instruction, or aborts the program.

Exception number An unsigned integer assigned to each type of exception.

- By processor designer: division by 0, page fault, illegal memory access, break point, arithmetic overflow.
- By OS kernel designer: system call, signals from external IO devices.

Exception table Allocated at system boot time. Entry k contains the address of the handler of exception k. Its starting address is stored in a special CPU register called the **exception table base register**.

Exception is akin to procedure call except for some important differences:

- As with procedure call, the return address is pushed on the stack. But the return address can be the current or the next instruction.
- The processor also pushes some additional processor state on the stack that will be necessary when the handler returns and the interrupted program continues, e.g. EFLAGS.

- When control is transferred to the kernel, these items are pushed onto the kernel stack instead of the user stack.
- Exception handlers run in kernel mode, thus have complete access to all system resources.

8.1.2 Classes

Async Not caused by any instruction.

Sync Caused by the current instruction (the faulting instruction).

Table 16: Classes of exceptions

Class	Cause	Async/Sync	Return address
Interrupt	Signal from IO devices	Async	The next instruction
Trap	Intentional exception	Sync	The next instruction
Fault	Potentially recoverable error	Sync	The current instruction
Abort	Non-recoverable error	Sync	Never returns

Table 17: Example of x86-64 exceptions

Number	Description	Class	Outcome
0	Division by 0 (Floating exception)	Fault	No attempt to recover
13	General protection fault (Segmentation fault)	Fault	No attempt to recover
14	Page fault	Fault	Maps disk page to memory page.
18	Machine check (Fatal hardware error)	Abort	
32-255	OS defined exceptions	Interrupt or trap	

syscall C function & x86-64 instruction to make system calls. For the instruction:

- System call number: `%rax`
- Arguments: `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value: `%rax`. -4095 $\sim$ -1 indicates error, corresponding to negative `errno`.
- Registers destroyed: `%rcx, %r11`

"Hello world" written using system-level function:

```
int main() {
  write(1, "Hello, world\n", 13); //1: stdout. 13: number of bytes to write.
  _exit(0);
}
```

Table 18: Example of x86-64 system calls

N.O.	Name	Description	N.O.	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

8.2 Processes

Process An instance of a program in execution. Each program runs in the context of a process. A process provides to applications 2 key abstractions:

- An independent **logical control flow** that provides the illusion that this program has exclusive use of the processor.
- A **private address space** that provides the illusion that this program has exclusive use of the memory system.

Context The state that the program needs to run correctly, including the program's code and data stored in the memory, its stack, general registers' contents, PC, environment variables, and the set of open file descriptors.

Concurrent flow A logical flow whose execution overlaps with another logical flow.

Concurrency The general phenomenon of multiple flows executing concurrently.

Multi-tasking(time slicing) The notion of a process taking turns with other processes. Processes get **preempted** by each other.

Time slice Each time period that a process executes a portion of its flow.

Parallel flows A proper subset of concurrent flows: two flows running concurrently on different processor cores or computers.

Private address space Private in the sense that the byte related to an address cannot be read-/written by other processes.

User/Kernel mode A mechanism provided by the processor to restrict the instructions that an application can execute and the portions of the address space that it can access.

- Controlled by the **mode bit** in a control register on the processor.
- A process in user mode cannot execute **privileged instructions**, e.g. halt the processor, change the mode bit, start an IO operation.
- User programs can only access kernel code and data via system call.

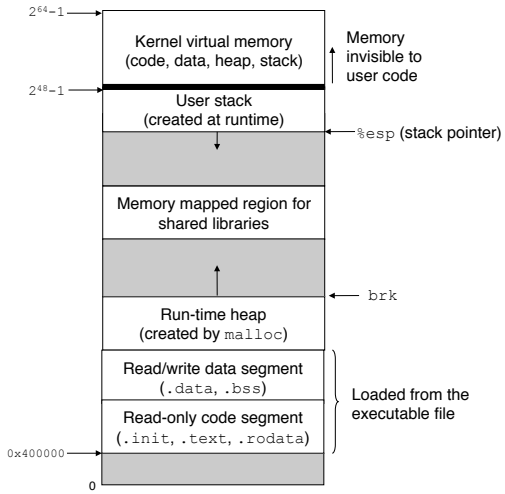


Figure 8: Structure of private address space

- Exception handler switches the processor to kernel mode, and switches back when it returns.
- `/proc` filesystem allows user mode processes to access the contents of kernel data structures.

Context switches A higher-level ECF used by the OS kernel to implement multi-tasking. It saves the context of the current process; restores the saved context of some previously preempted process; passes control to this newly restored process. Such decision is called **scheduling**, which is handled by some kernel code known as the **scheduler**.

8.3 System call error handling

Error-reporting function:

```
void unix_error(char *msg) {
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));//strerror in string.h
  exit(0);
}
```

Error-handling wrapper (using `fork`) as an example:

```
pid_t Fork(void) {
  pit_t pid;
  if((pid = fork()) < 0)
    unix_error("Fork error");
  return pid;
}
```

8.4 Process control in Unix

8.4.1 Obtain process ID

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void); //get pid of parent process.
```

8.4.2 Create/Terminate processes

From the perspective of a programmer, a process is always in one of the 3 states:

Running Either executing on CPU or waiting to be scheduled.

Stopped Suspended and won't be scheduled.

- Happens after receiving signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU.
- Remains stopped until receiving signal SIGCONT, after which the process becomes running again.

Terminated Permanently stopped. Happens after:

- Receiving signals whose default action is to terminate the process.
- Returning from the main routine.
- Calling the `exit` function. `status` is the **exit status**, which can also be set by returning an integer value from the main routine.

  ```
  #include <stdlib.h>
  void exit(int status);
  ```

A **parent process** creates a **child process** with the `fork` function:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- The child gets an identical yet separate copy of the parent's user-level virtual memory.
- The child also gets identical copies of the parent's open file descriptors.
- The parent and the child have different PIDs.
- In the parent, `fork` returns the pid of the child; in this child, it returns 0.
- The parent and the child are executed concurrently.

8.4.3 Reaping child processes

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statusp, int options);
```

1. Default behaviour (`options=0`):
   - Elements of **wait set**:
     - `pid>0`: the single process whose ID is `pid`.
     - `pid=-1`: all child processes of the parent process.
   - Suspend the calling process until one of the child processes in its wait set terminates.
   - If a process in the wait set already terminated, return immediately.
   - The pid of the child process is returned. It is reaped.
2. Other options:

   WNOHANG Return 0 immediately if no child process has terminated.

   WUNTRACED Terminated → terminated or stopped.

   WCONTINUED Suspend the calling process, until a running process in the wait set terminates or a stopped process in the wait set resumes after receiving signal SIGCONT.

   WNOHANG|WUNTRACED The options can be combined by |.
3. If `statusp` is not NULL, then the status of the reaped child process is saved in the value `status` pointed to by `statusp`. It can be explained with the following macros defined in `wait.h`:
   - WIFEXITED(`status`): true if the child process terminated normally via `exit` or `return`.
   - WEXITSTATUS(`status`): returns the exit status of a normally terminated child. Only defined if WIFEXITED() is true.
   - WIFSIGNALED(`status`): true if the child process terminated because of an uncaught signal.
   - WTERMSIG(`status`): returns the number of the signal that terminated the child process. Only defined if WIFSIGNALED() is true.
   - WIFSTOPPED(`status`): true if the child process is stopped.
   - WSTOPSIG(`status`): the number of the signal that caused the child to stop. Only defined if WIFSTOPPED() is true.
   - WIFCONTINUED(`status`): true if the child process restarted after receiving signal SIGCONT.
4. If the calling process has no children, `waitpid` returns -1 and sets `errno` to ECHILD. If `waitpid` is interrupted by a signal, it returns -1 and sets `errno` to EINTR.
5. Simplified version:

```
pid_t wait(int *statusp); //equivalent to waitpid(-1, statusp, 0)
```

8.4.4 Putting process to sleep

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
int pause(void); //always returns -1
```

`sleep` returns 0 or the number of seconds left to sleep. The latter is possible because it can be interrupted by a signal. `pause` puts the calling process to sleep until it receives a signal.

8.4.5 Load & run programs

```
#include <unistd.h>
int execve(const char* filename, const char *argv[], const char *envp[]);
//related main function:
int main(int argc, char **argv, char **envp);
```

- Does not return to the calling process, unless an error happens (e.g cannot file the file).
- `argv` points to a `null`-ending array of pointers. Each pointer points to an argument string. `argv[0]` is the name of the executable.
- `envp` points to a `null`-ending array of pointers. Each pointer points to an environment variable string, e.g. `name=value`.

To manipulate the environment array:

```
#include <stdlib.h>
char *getenv(const char* name); //NULL if name does not exist
int setenv(const char* name, const char* value, int overwrite);//0 for success, -1
    for failure.
void unsetenv(const char*);
```

8.5 Signals

Signal A small message that informs a process of an event of some type that has happened in the system.

Pending signal A sent but not received signal. There is at most one pending signal of a particular type. If there exists already a pending signal of a type, following signals of the same type are simply discarded. The kernel maintains the pending bit vector (signal mask) for each process.

Blocked signal Can be sent but won't be received. The kernel maintains the blocked bit vector for each process.

Process group Each process belongs to one process group. By default a child process belongs to the same process group as its parent.

```
#include <unistd.h>
//Return process group ID of the current process
pid_t getpgrp(void);
```

```
/* Change process group id of the process pid to pgid.
   If pid = 0, act on the current process.
   If pgid = 0, use the PID of the process specified by pid as the group ID.
   return 0 for success, -1 for error. */
int setpgid(pid_t pid, pid_t pgid);
```

8.5.1 Send signals

1. Use `/bin/kill` program. e.g. `/bin/kill -9 15123` sends 9 (SIGKILL) to process 15123. A negative PID causes the signal to be sent to all processes in the process group with the ID.
2. Send from keyboard.
   - A job represents processes created to evaluate a command line in Unix shell. Unix shell creates a separate process group for each job.
   - At any time, there can be at most 1 foreground job and 0 or more background jobs.
   - Ctrl + C sends SIGINT (default behaviour: terminate) to each process in the foreground process group.
   - Ctrl + Z sends SIGTSTP (default behaviour: suspend) to each process in the foreground process group.
3. Use `kill` function to send signals to other processes (including itself).

```
#include <sys/types.h>
#include <signal.h>
/* pid > 0: send sig to pid.
   pid = 0: send sig to all processes in the same process group as the
       calling process.
   pid < 0: send sig to all processes in the process group |pid|
   return 0 for success, -1 for error. */
int kill(pid_t pid, int sig);
```

4. Use `alarm` function to send SIGALRM signal to the process itself. Note that the default behaviour of the signal is to terminate.

```
#include <unistd.h>
/* Send SIGALRM to the calling process after secs seconds. secs = 0: no new
    alarm is scheduled.
Cancels any pending alarm.
Returns the number of remaining seconds of the pending alarm, or 0 if there
    isn't any pending alarm.*/
unsigned int alarm(unsigned int secs);
```

8.5.2 Receive signals

Predefined default behaviour after receiving a signal can be:

- The process terminates.
- The process terminates and dumps core.

- The process suspends until restarted by SIGCONT.
- The process ignores the signal.

The default behaviour of SIGSTOP and SIGKILL cannot be modified. The default behaviour of other signals can be modified with `signal` function.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
/*
handler = SIG_IGN: ignore signal signum.
handler = SIG_DFL: restore default behaviour of signal signum.
Otherwise: install new signal handler for signum.
Return previous handler for success. SIG_ERR for error.
*/
sighandler_t signal(int signum, sighandler_t handler);
```

8.5.3 Block/Unblock signals

Implicit blocking mechanism The kernel blocks any pending signals of the type currently being processed by a handler.

Explicit blocking mechanism Use `sigprocmask` and its helper functions.

```
#include <signal>
/*
how = SIG_BLOCK: blocked = blocked | set.
how = SIG_UNBLOCK: blocked = blocked & ~set.
how = SETMASK: blocked = set.
return 0 for success, -1 for error. Old value of blocked is kept in oldset (if
    oldset is not NULL).*/
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

// set manipulation functions. Return 0 for success, -1 for error.
int sigemptyset(sigset_t *set); //Initialize as empty set
int sigfillset(sigset_t *set); //Add all signals to the set
int sigaddset(sigset_t *set, int signum); // Add signum
int sigdelset(sigset_t *set, int signum); //Delete signum

//Check if signum is member of set. Return 1 for yes, 0 for no and -1 for error.
int sigismember(const sigset_t *set, int signum);
```

8.5.4 Writing signal handlers

1. Handlers run concurrently with the main program and share the same global variables, and thus can interfere with the main program and other handlers.
2. The rules for how / when signals are received are often counterintuitive.
3. Different systems may have different signal handling semantics.

Safe signal handling Some conservative guidelines:

1. Keep handlers as simple as possible, e.g. set a global flag and return immediately, and the main program checks the flag periodically for further processing associated with the receipt of the signal.
2. Call only async-signal-safe functions in handlers. Such functions are either reentrant or unable to be interrupted by a signal handler.
3. Save `errno` on entry to the handler and restore it when the handler returns to avoid interference with other parts of the program that rely on `errno`.
4. Block all signals when accessing shared global data structures. Accessing a data structure d from the main program typically requires a sequence of instructions. If the sequence is interrupted by a handler that also accesses d, d may end up in an inconsistent state.
5. Declare global variables as `volatile` to forbid the compiler to cache them.
6. Declare flags with `sig_atomic_t` to guarantee atomic read/write.

Correct signal handling Pending signals are not queued.

```
//SIGCHLD handlers

//Buggy!
void handler1(int sig) {
  int olderrno = errno;
  if(waitpid(-1, NULL, 0) < 0) //1 reaped. 1 pended. Other possibly discarded!
    sio_error("waitpid error");
  Sleep(1);
  errno = olderrno;
}

//Correct
void handler2(int sig) {
  int olderrno = errno;
  while(waitpid(-1, NULL, 0) > 0); //Reaps as many children as possible.
  if(errno != ECHILD)
    Sio_error("waitpid error");
  Sleep(1);
  errno = olderrno;
}
```

Portable signal handling `sigaction` function and `Signal` wrapper to resolve historical incompatibility. Not something interesting.

8.5.5 Synchronize flows to avoid concurrency bugs

How to program concurrent flows that read and write the same storage locations? Synchronize concurrent flows to allow the largest set of instruction interleavings that produce the correct result.

The following program describes the typical structure of a Unix shell.

```
void handler(int sig) {
  int olderrno = errno; /* (*@\hyperref[safesignalhandling]{\color{gray}Safe signal
      handling 3}@*) */
  pid_t pid;
  sigset_t mask_all, prev_all;
  Sigfillset(&mask_all);
  while((pid = waitpid(-1, NULL, 0)) > 0) { /*(*@\hyperref[safesignalhandling]{\
      color{gray}Safe signal handling 4}@*)*/
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    deletejob(pid);
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
  }
  if(errno != ECHILD)
    Sio_error("waitpid error");
  errno = olderrno;
}
int main(int argc, char** argv) {
  pid_t pid;
  sigset_t mask_all, prev_all;

  Sigfillset(&mask_all);
  Signal(SIGCHLD, handler);
  initjobs(); /*Initialize job list*/

  while(1) {
    if((pid = Fork()) == 0) { /*child*/
      Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /*parent. (*@\hyperref[
        safesignalhandling]{\color{gray}Safe signal handling 4}@*)*/
    addjob(pid);
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
  }
  exit(0);
}
```

Unfortunately it will potentially cause race condition. If the child is scheduled to run after the parent calls `Fork`, and completes before the parent is scheduled to run again, a SIGCHLD will be sent to the parent, so that `deletejob` will be called before `addjob`.

The solution is to block SIGCHLD before calling `Fork`. Note that the child inherits the block set from it parent, so SIGCHLD has to be unblocked before calling `Execve`.

```
int main(int argc, char** argv) {
  pid_t pid;
  sigset_t mask_all, mask_one, prev;

  Sigfillset(&mask_all);
  Sigemptyset(&mask_one);
  Sigaddset(&mask_one, SIGCHLD);
  Signal(SIGCHLD, handler);
  initjobs();

  while(1) {
    Sigprocmask(SIG_BLOCK, &mask_one, &prev);
    if((pid = Fork()) == 0) {
      Sigprocmask(SIG_SETMASK, &prev, NULL); //Unblock SIGCHLD
      Execve("/bin/date", argv, NULL);
```

```
    }
    Sigprocmask(SIG_BLOCK, &mask_all, NULL);
    addjob(pid);
    Sigprocmask(SIG_SETMASK, &prev, NULL);
  }
}
```

8.5.6 Explicitly waiting for signals

Sometimes the main program has to explicitly wait for a handler to run. For example, after a foreground job is created in Linux shell, another command can be processed only after the job completes with a SIGCHLD signal.

```
volatile sig_atomic_t pid;
void sigchld_handler(int s) {
  int olderrno = errno;
  pid = waitpid(-1, NULL, 0);
  errno = olderrno;
}
void sigint_handler(int s) {}

int main(int argc, char** argv) {
  sigset_t mask, prev;
  Signal(SIGCHLD, sigchld_handler);
  Signal(SIGINT, sigint_handler);
  Sigemptyset(&mask);
  Sigaddset(&mask, SIGCHLD);

  while(1) {
    Sigprocmask(SIG_BLOCK, &mask, &prev); //to avoid race condition. SIGCHLD being
        received before setting pid to 0 results in infinite loop.
    if(Fork() == 0) /* child */
      exit(0);

    /* parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL);

    while(!pid);

    //work after receiving SIGCHLD
    printf(".");
  }
  exit(0);
}
```

The code is correct but the while loop is wasteful. Changing it to

```
while(!pid) //we still need a loop because of SIGINT
  pause();
```

causes race condition: SIGCHLD may arrive after the test in `while` and before `pause`, which causes infinite loop. Changing `pause()` to `sleep(1)` is correct but too slow. The appropriate solution is to use `sigsuspend`.

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

It's equivalent to an atomic version of:

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

`while(!pid);` should be changed to

```
while(!pid)
  sigsuspend(&prev);
//SIGCHLD is still blocked here. Unblock it.
Sigprocmask(SIG_SETMASK, &prev, NULL);
```

8.6 Nonlocal jump

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int retval);

//version of setjmp and longjmp that can be used by handlers
int sigsetjmp(sigjmp_buf env, int savesigs);
int siglongjmp(sigjmp_buf env, int retval);
```

`setjmp` saves the current calling environment in the `env` buffer, including PC, stack pointer and general-purpose registers. It returns 0 here, but its return value should not be assigned to a variable. Yet it can safely be used in a `switch` or conditional statement.

`longjmp` restores the calling environment from `env` and triggers a return from the most recent call of `setjmp` that initialized `env`. The `setjmp` returns with non-zero value `retval`.

The exception mechanisms in JAVA and C++ are higher-level, more structured versions of `setjmp` and `longjmp`. A `catch` clause inside a `try` statement is akin to a `setjmp` function, and a `throw` statement is akin to a `longjmp` function.

9 Virtual Memory

Virtual memory provides 3 important abilities:

1. It uses the main memory efficiently by treating a cache for a namespace stored on the disk.
2. It simplifies memory management by providing each process with a uniform address space.
3. It protects the address space of each process from being corrupted by other processes.

9.1 Introduction

9.1.1 Virtual address space

The main memory is organized as a M-byte consecutive array. Each byte has a unique **physical address** starting from 0. CPU can access memory with its physical address, namely **physical addressing**. Modern processors use **virtual addressing** to access memory. CPU generates a **virtual address** to access memory. The VA is translated into a physical address by the **memory management unit(MMU)** on the CPU chip before being sent to the main memory. CPU generates VA inside a **virtual address space** $\{0, 1, 2, \ldots, N-1\}$ with $N = 2^n$. Similarly there is a **physical address space** $\{0, 1, 2, \ldots, M-1\}$. We assume $M = 2^m$.

9.1.2 VM as cache

Conceptionally, VM is organized as a N-byte consecutive array on the disk. Each byte has a unique VA. Data on the disk is cached inside the main memory. Like other caches in the memory hierarchy, data on the disk is divided into blocks to serve as basic units for transportation between the disk and the main memory. The virtual memory is divided into **virtual pages(VP)** of size $P = 2^p$, and the physical memory is divided into **physical pages(PP)**, or **page frames** of the same size. At any time, the set of VPs is divided into 3 subsets:

- Unallocated: pages not yet created by the VM system. Not related to any data, thus occupies no disk space.
- Cached: Allocated and cached in the physical memory.
- Uncached: Allocated but not cached in the memory.

Cost of DRAM cache miss is enormous:

- DRAM 10 times slower than SRAM; hard disk 100000 times slower than DRAM.
- Cost of reading the first byte from a disk sector is 100000 times higher than that of reading further bytes consecutively.

Properties of DRAM cache: large pages (4KB-2MB); full associative (any VP can be put in any PP); write-back instead of write-through.

Page table is used to tell if a VP is cached inside a PP; if so, which PP; if not, where is it on disk. It's an array of PTE(page table entry)s stored in memory and managed by the OS. Each VP is related to a PTE in the page table. A PTE has a valid bit and an n-bit address segment, which is used to indicate the starting address of the PP in DRAM if the valid bit is set (1), or the

starting address of the PP on disk if the valid bit is unset (0). If the VP is unallocated, the valid bit is 0 and the address is set to null.

A DRAM cache miss is called a page fault. An exception handler in the kernel will be called. The needed VP will be copied into a PP at the cost of a sacrificed page. The instruction that caused the page fault will be executed again.

The efficiency of VM is guaranteed by locality. Programs tend to work mainly on a small set of pages called the working set. After some initial cost, page fault is supposed to be rare.

9.1.3 VM as tool for memory management

- VM simplifies linking. A separate virtual address space allows each process to use the same basic format for its memory image, e.g. `.data` always starts at 0x400000. The implementation of the linker is greatly simplified.
- VM simplifies loading. Loader does not have to copy any data from disk to memory. All it does is to allocate VPs for `.text` and `.data` segments of the executable and mark them as uncached. They will be paged in by the VM automatically on demand.
- VM simplifies data & code sharing. Different processes can share data & code, e.g. C standard libraries and kernel codes, by mapping VPs in their virtual address spaces to the same PPs.
- VM simplifies memory allocation. Consecutive VPs do not have to be mapped to consecutive PPs.

9.1.4 VM as tool for memory protection

- Separate virtual address space makes it easy to isolate private memories of different processes.
- Additional control bits can be added to PTEs for memory protection. For example, a SUP bit can be used to restrict access of the VP to super user; READ/WRITE bits can be used to allow and forbid processes from reading and writing the VP. If such restrictions are violated, the CPU triggers a protection fault, which is reported as "segmentation fault" in Linux.

9.2 Memory Translation

Memory translation is a map from the virtual address space to the physical address space:

$$MAP : VAS \to PAS \cup \emptyset$$

$$MAP(A) = \begin{cases} A' & \text{If data at VA A is at PA A'} \\ \emptyset & \text{If data at VA A is not in physical memory} \end{cases}$$

PTBR Page table base register: a control register in the CPU. Points to the current page table.

VPN Virtual page number.

VPO Virtual page offset in bytes. Last p bits of VA.

PPN Physical page number.

PPO Physical page offset in bytes. Last p bits of PA. Equal to VPO.

Page hit:

1. CPU generates a VA and sends it to the MMU.
2. MMU generates the address of the appropriate PTE (PTBR + VPN), and asks SRAM/DRAM for the PTE.
3. SRAM/DRAM returns the PTE to the MMU.
4. MMU obtains the PPN from the PTE, and constructs the PP(PPN + VPN). The PP is sent to SRAM/DRAM.
5. SRAM/DRAM returns the asked word to the CPU.

Page fault:

1. 1-3: the same as page hit.
2. The valid bit of the PTE is 0. MMU triggers an exception. Control is transferred to the page fault exception handler in the kernel.
3. The handler identifies a victim page in the physical memory. If it has been modified, it's paged out to the disk.
4. The handler pages in the new page (whose address on disk is in the PTE) and updates the PTE.
5. The handler returns control to the original process and the instruction is executed again.

TLB Translation lookaside buffer. A small buffer in the MMU dedicated to PTEs. It's usually highly associative. If TLB has 2^t sets, then the least significant t bits of the VPN is used as the TLBI(TLB index) and the rest is used as TLBT(TLB tag).

Multi-level Page Tables One single page table calls for too much memory footprint. With a 32-bit address space, 4KB pages and 4B PTEs, a 4MB page table has to stay in memory. For a 64-bit address space it's even large. The page table can be compacted with a hierarchy of page tables.

As shown in the figure, each entry in the 1st level page table is responsible for a 4MB chunk in the VAP, which is 1024 VPs. If none of the pages in the chunk is allocated, the entry in the 1st level page table is empty. If at least one of them is allocated, the entry points to the address of a 2nd level page table.

9.3 Case study: Intel Core i7/Linux VM system

9.3.1 Intel Core i7 memory translation

- Core i7 uses 48-bit VAS (256TB) and 52-bit PAS (4PB).
- L1 d-cache: 32KB, 64B block, 8-way associative, thus 64 sets. PA: 40-bit CT, 6-bit CI, 6-bit CO.

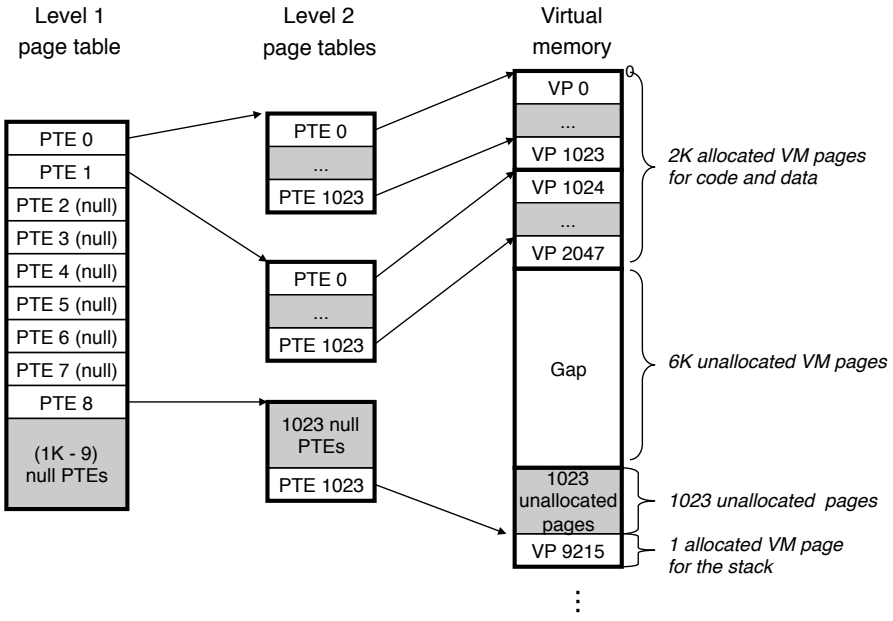


Figure 9: A 2-level page table hierarchy

- 4-level page table. VA: 4× 9-bit VPNs + 12-bit VPO. Level 1: 512GB each entry. Level 2: 1GB. Level 3: 2MB. Level 4: 4KB (page size).
- Parallel address translation: 12-bit VPO = 12-bit PPO = 6-bit CI + 6-bit CO, thus PPN = CT. During address translation, VPN is transferred to MMU while VPO is transferred to L1 d-cache. When MMU asks TLB for a PTE, L1-cache finds the appropriate set according to CI and reads the 8 tags and data words (according to CO). After the MMU obtains PPN from TLB, it simply compares it against the 8 tags and choose the word whose tag matches (in case of a cache hit).

9.3.2 Linux VM system

Linux maintains a separate VM space for each process. VM is organized as a collection of areas(segments). An area is a contiguous chunk of existing (allocated) VM whose pages are related in some way. Each existing VP is contained in an area. Any VP that is not part of some area does not exist, cannot be referenced by the process, and does not consume any additional resources.

Kernel data structures used to keep track of VM areas of a process:

- A separate task struct for each process (`task_struct`). Its elements either contain of point to information needed by the kernel to run the process: PID, pointer to the user stack, name of the executable file, PC, etc.
- One of the entries points to an `mm_struct` that characterizes the current state of the VM. It contains:

  - `pgd`: base of the level 1 page table.
  - `mmap`: a list of `vm_area_structs`, each of which characterizes an area.

- `vm_area_structs` contains:
  - `vm_start`: points to the beginning of the area.
  - `vm_end`: points to the end of the area.
  - `vm_prot`: read/write permissions for all pages in the area.
  - `vm_flags`: whether pages in the area are shared with other processes, etc.
  - `vm_next`: points to the next area struct in the list.

Linux page fault exception handling:

1. Check if the VA is legal, i.e. if the VA is contained in an area, by comparing it against `vm_start` and `vm_end` in each area struct. If illegal: segmentation fault.
2. Check if the access is legal, i.e. if the process has enough permissions to read-/write/execute the pages in the area. If illegal: protection exception.
3. Page out victim page and page in the new page. Re-execute the instruction.

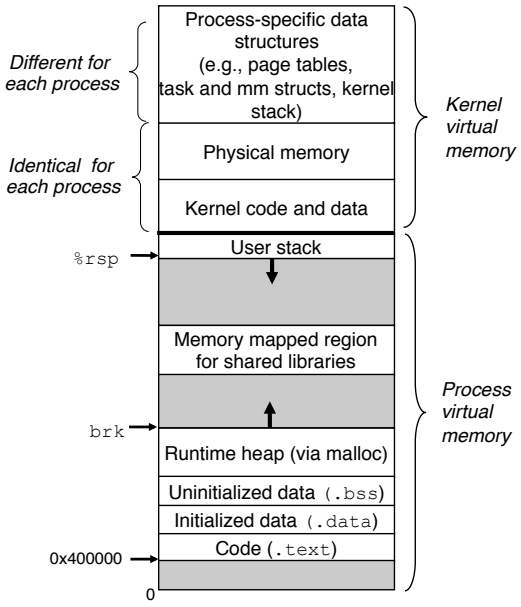


Figure 10: VM of a Linux process

9.4 Memory mapping

Linux **initializes**[2] the contents of a VM area by associating it with an object on disk. The process is called memory mapping. Areas can be mapped to two kinds of objects:

- Regular. A contiguous section of a regular disk file. The section is divided into page-size pieces, each containing the initial contents of a VP. Because of demand paging, no VP is actually swapped into physical memory until the page is touched by the CPU.
- Anonymous. An anonymous file created by the kernel containing only binary zeros. When the page is first touched, a victim page in the physical memory is overwritten with 0. No data is transferred between disk and memory.

[2]Further changes made to VM areas are not necessarily reflected back to the original objects on disk. See below (private areas).

Once a VP gets initialized, it is swapped back and forth between the physical memory and the swap file on disk[3]. The size of the swap file bounds the number of VPs that can be allocated by the running processes.

Let's revisit some concepts introduced before.

Shared Object An object can be mapped to an area of VM as either a shared object or a private object. Any writes made by an process to the area in its VM mapped to a shared object are visible to other processes that also map the shared object into their VMs, and are reflected in the original object on disk. On the contrary, changes made to an area mapped to an private object are not visible to other processes. Neither are they reflected back to the object on disk.

Private objects are mapped into VM using copy-on-write technique. Multiple processes can map the same private object on disk into their VMs. Related PTEs are flagged as read-only, and related area structs are flagged as **private copy-on-write**. Only a single copy of the private object is kept in the physical memory until either process attempts to write to some page in the private area, when a protection fault is triggered. A new copy of the page will be created in the memory, with the related PTE updated. The process will have write permission to the new page.

fork When the current process calls the `fork` function, the kernel creates various data structures for the new process and assigns it a unique PID. In order to create VM for the new process, it create exact copies of the current process's `mm_struct`, area structs and page table. All pages in both processes are flagged as read-only, and all area structs are flagged as private copy-on-write. When either process performs writes, new pages will be created by the copy-on-write mechanism. The abstraction of a private address space for each process is preserved.

execve `execve("a.out", NULL, NULL);` loads and runs the program contained in the executable object file `a.out` within the current process, effectively replacing the current program with the `a.out` program. This is accomplished in the following steps:

- Delete existing area structs in the user portion of the current process's VM.[4]
- Map private areas. New area structs are created for the code, data, bss, stack and heap areas of the new programs, all of whom are private copy-on-write. Code and data areas are mapped to `.text` and `.data` sections of `a.out`. Bss is mapped to an anonymous file whose length is contained in `"a.out"`. Stack and heap are mapped to an anonymous file of initial length 0.
- Map shared areas. Shared objects such as `libc.so` are mapped into the shared region of the VM.
- Set the PC of the current process's context to the entry point in the code area.

Linux processes can use the `mmap` function to create new VM areas and map objects to these areas.

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void* start, size_t length);
```

[3]Looks like an error in the book. "Swapped between a swap file"?

[4]Maybe also the page table?

`mmap` asks the kernel to create a new VM area mapped to a contiguous chunk of the object specified by file descriptor `fd`. The size of the chunk is `length` and it starts from `offset` bytes from the beginning of the file. `start` is the suggested beginning address of the new area, usually specified as NULL. `prot` contains access permissions of the new area (i.e. `vm_prot` in its area struct). It can be

- PROT_EXEC: pages within the area are composed of instructions executable by the CPU.
- PROT_READ: pages within the area are readable.
- PROT_WRITE: pages within the area are writeable.
- PROT_NONE: pages within the area cannot be accessed.

`flags` describes type of the object. They can be ORed with |.

- MAP_ANON: anonymous object.
- MAP_PRIVATE: private copy-on-write.
- MAP_SHARED: shared object.

`munmap` deletes an area created by `mmap`.

9.5 Dynamic memory allocation

Dynamic memory allocation is necessary because we often do not know the size of some data structures until runtime.

9.5.1 Requirements

- Handle arbitrary sequence of requests. No assumption should be made about the sequence of allocate / free requests except that each free request should correspond to a previous successful allocate request.
- Reply to requests immediately. The allocator is not allowed to reorder or buffer requests to improve efficiency.
- Use only the heap.
- Align the blocks so that any data object can be stored.
- Leave allocated blocks unmodified. The allocator can only manipulate or change free blocks.

9.5.2 Performance goals

- Maximize throughput, i.e. requests handled per minute. Reasonable performance: the worst running time of an allocate request is linear in the number of free blocks, while the free request can be finished in constant time.
- Maximize memory utilization. Peak utilization:

$$U_k = \frac{\max_{i \le k} P_i}{\max_{i \le k} H_k}.$$

  P_k is the aggregate payload after request R_k and H_k is the size of the heap (the heap can grow or shrink).

9.5.3 Fragmentation

There is enough memory left but it cannot be used to fulfil an allocate request. Internal fragmentation: an allocated block is larger than the payload, usually to fulfil aligning requirement. It can be quantified. External fragmentation: no single block is large enough for an allocate request, though there is enough aggregate free memory. It's hard to quantify because it depends on future allocate requests.

9.5.4 Implicit free lists

Any allocator needs some data structure to tell the boundaries of blocks and to tell allocated blocks from free ones. Most allocators embed this information in the block itself.

- Implicit free list uses a word (4B) at the beginning of each block as its header.
- The lowest bit is used to indicated if it's allocated (1) or free(0).
- The highest 29 bits are used to indicated its size (8B alignment is required so that the lowest 3 bits are not needed).
- A special ending block is needed, e.g. one with size 0 and flagged as allocated.
- Simple. Searching within free blocks takes time linear in the number of all blocks.

9.5.5 Placement policies

- First fit: Search from the beginning and use the first suitable free block. Leaves large free blocks at the end of the list, but tends to leave small free blocks near the beginning of the list, which increases the time to search for large free blocks.
- Next fit: Search from where the last query ends and use the first suitable free block. Faster than first fit but results in lower memory utilization.
- Best fit: Check each free block and choose the smallest appropriate one. Requires thorough search of the heap.

9.5.6 Splitting free blocks

If the chosen free block is larger than the requested size, the allocator can split it into 2 parts: an allocated block and a new free block.

9.5.7 Obtaining additional heap memory

If no appropriate free block is available, the allocator has to call the `sbrk` function to ask the kernel for more heap memory.

```
/* Increase pointer brk by incr bytes. Returns the old brk. When incr is negative,
    it shrinks the heap. */
#include <unistd.h>
void *sbrk(intptr_t incr);
```

The obtained memory is turned into a large free block at the end of the list.

9.5.8 Coalescing free blocks

Fault fragmentation: a series of contiguous small free blocks. They should be coalesced into a large free block.

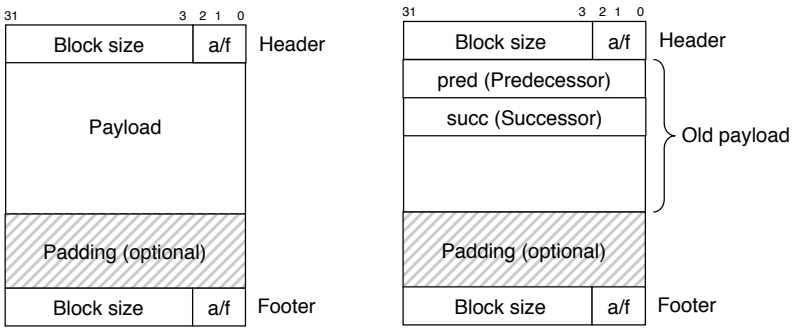
- Immediate coalescing: each time a block is freed, coalesce neighbouring free blocks. Simple but might cause thrashing.
- Deferred coalescing: coalesce sometime later, e.g. after an allocate request fails.

Coalescing the next free block is simple. To coalesce the previous one, **boundary tag** is needed. A copy of the block header is put at the end of the block as a footer, so that the allocator can decide to coalesce the previous block or not by checking its footer. Boundary is elegant but causes memory overhead, especially when blocks are small. The method can be optimized by eliminating footers of allocated blocks: store the allocated/free bit of the previous block inside an excess low bit of the current block, so that we no longer have to check its footer when it's allocated.

9.5.9 Explicit free lists

Implicit free list: allocation time linear to total number of blocks.

Explicit free list: allocation time linear to total number of free blocks. Downside: free blocks have to be large enough to hold pointers, causing larger minimum size and better chance of internal fragmentation. Free time: depends on policy to order the blocks in the free list.



(a) Allocated block **(b) Free block**

Figure 11: Format of heap blocks using doubly linked free lists

- LIFO: inserting a newly freed block at the beginning of the list. Guarantees constant free of blocks.
- Maintains blocks in address order: linear free time, yet better memory utilization.

9.5.10 Segregated free lists

Segregated storage: maintain multiple free lists to reduce allocation time. Partition the set of all possible block sizes into equivalence classes called size classes, e.g. by power of 2:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, 7, 8\}, \ldots, \{1025 - 2048\}, \{2049 - 4096\}, \{4097 - \infty\}$$

Or maybe assign small blocks to their own size classes, e.g. a size class for each of 1-1024. Each size class has its own free list.

9.5.11 Garbage collection

Reachability graph: each node is an allocated memory block. An edge $p \rightarrow q$ means some position in p holds a pointer to some position in q. Root node of the graph: out-of-heap position holding pointers pointing to positions in the heap, e.g. registers, stack variables, global variables, etc. If there exists a path from any root node to a node p, then p is reachable. At anytime, unreachable nodes correspond to garbage that can no longer be used by the application, and thus should be reclaimed. The garbage collector maintains some representation of the reachability graph, frees unreachable nodes and returns them to the free list. Since C does not maintain any type information for memory locations, there is no obvious way to determine if a variable is a pointer, neither is there an obvious way to tell if it points to a position inside an allocated block. A garbage collector implemented for C has to be conservative: there are false positives when detecting reachable nodes.

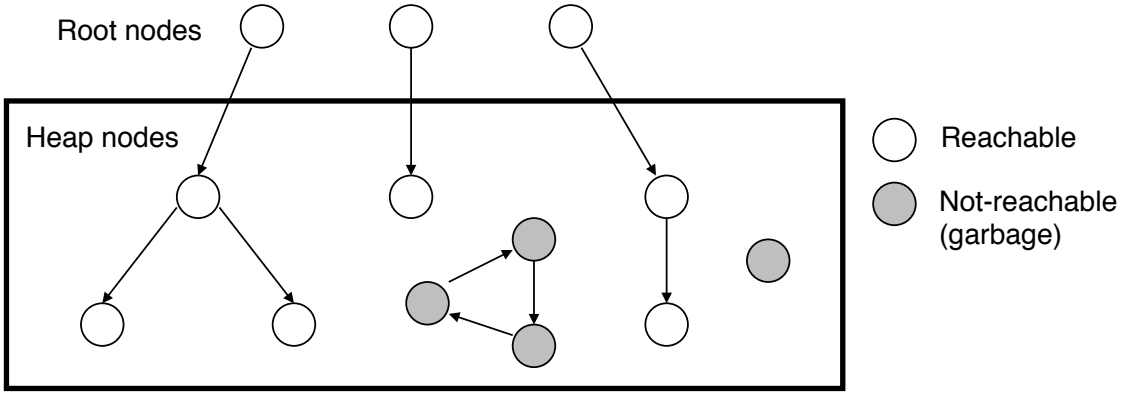


Figure 12: Reachability graph

10 System-level I/O

I/O is the process of copying data between the main memory and external devices, e.g. disk drivers, terminal and network. All runtime systems offer high-level I/O tools, e.g. `stdio.h` of C and `iostream` of C++. Unix system offer system-level I/O functions to implement these high-level I/O functions. Learning Unix I/O can help us understand other system concepts, and sometimes we have no other option but to use Unix I/O.

10.1 Unix I/O interface

A Linux file is a sequence of bytes. All I/O devices are modeled as files, and all input and output is performed by reading and writing the appropriate files. This elegant mapping allows the Linux kernel to export a simple low-level I/O application interface.

- Opening files. An application asks the kernel to open a file in order to access the related IO device. The kernel returns a small non-negative integer called the **descriptor** that identifies the file in subsequent operations. The kernel keeps all information of the opened file, while the application keeps nothing but the descriptor. Each process created by the Linux shell has 3 open files as it begins: standard input (0), standard output (1) and standard error (2). `unistd.h` defines the constants `STDIN_FILENO, STDOUT_FINENO` and `STDERR_FILENO`.
- Change the current file position. Linux keeps a file position k for each opened file, initialized as 0. It describes the byte offset from the beginning of the file. An application can explicitly set the current file position k with the `seek` operation.
- Read from / Write to a file. Read: copy $n > 0$ bytes from the file to the memory, starting from the current file position k, and increasing k to $k+n$. Given a file of size m, a read when $k \geq m$ will trigger the EOF condition, which can be detected by the application. There exists no explicit "EOF" symbol at the end of a file. Write: copy $n > 0$ files from the memory to a file, starting from the file position k and update k.
- Close a file. When an application has finished accessing a file, it informs the kernel to close the file. The data structures created when opening the file are freed, and the file descriptor is restored back to the pool of available descriptors. When a process terminates for some reason, the kernel closes all open files and frees their memory resources.

10.2 Files

Each Linux file has a type to indicate its role in the system.

- A **regular** file contains any data: text file or binary file. There is no difference between the two for the kernel.
- A **directory** contains a series of links, each maps a file name to a file. Each directory contains at least two items: . to itself and .. to its parent directory in the directory hierarchy.
- A **socket** is used to communicate with another process across a network.
- Other file types include named pipe, symbolic link, character and block device, etc.

All files are organized in a single directory hierarchy. Each process has a current working directory to indicate its current position in the hierarchy. It can be changed by the `cd` command and specified with a pathname, which can be absolute or relative.

10.2.1 Open & close a file

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(char *filename, int flags, mode_t mode);
```

```
#include <unistd.h>
int close(int fd);
```

`flags`: indicate how the process wants to access the file:

O_RDONLY Read only

O_WRONLT Write only

O_RDWR Read/Write

O_CREAT Create a truncated (empty) version if the file does not exist.

O_TRUNC Truncate the file if it exists.

O_APPEND Set the file position to the end of the file before each write.

`mode`: Access permission bits. P = R(read), W(write), X(execute).

S_IPUSR User (Owner)

S_IPGRP The group to which the owner belongs

S_IPOTH Others (Anyone)

Each process has a `umask` as part of its context. The access permission bits are set as `mode & ~umask`, i.e. it is used to forbid some permissions, such as execute permission of anyone (S_IXOTH).

10.2.2 Read & write a file

```
#include <unistd.h>
ssize_t read(int fd, void* buf, size_t n);
ssizee_t write(int fd, const void* buf, size_t n);
```

In x86-64, `ssize_t` is a typedef of `long` while `size_t` is a typedef of `unsigned long`, so that the former can return -1 to indicate error.

Short count `read` or `write` transfers fewer bytes than the application specifies, for the following reasons:

- Encountering EOF on reads.
- Reading lines from a terminal: `read` transfers a text line each time.
- Reading / Writing network sockets.

10.3 Robust reading / writing with RIO package

Unbuffered IO functions Transfer data directly between memory and a file, with no application-level buffering. Useful for binary r/w to and from networks.

Buffered input functions Read text lines and binary data from a file whose contents are cached in an application-level buffer, similar to `printf`.

10.3.1 Unbuffered IO

```
/* Unbuffered read. Returns short count when encountering EOF.*/
ssize_t rio_readn(int fd, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nread;
  char *bufp = userbuf;

  while (nleft > 0) {
    if ((nread = read(fd, bufp, nleft)) < 0) { //error
      if (errno == EINTR) //Interrupted by sig handler. Should call read again
        nread = 0;
      else
        return -1;
    } else if (nread == 0) //EOF
      break;
    nleft -= nread;
    bufp += nread;
  }

  return n - nleft;
}

/* Unbuffered write. Never returns short count. */
ssize_t rio_writen(int fd, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nwritten;
  char *bufp = userbuf;

  while(nleft > 0) {
    if((nwritten = write(fd, bufp, nleft)) < 0) {
      if(error == EINTR)
        nwritten = 0;
      else
        return -1;
    }
    nleft -= nwritten;
    bufp += nwritten;
  }
  return n;
}
```

10.3.2 Buffered input

```
#define RIO_BUFSIZE 8192
typedef struct {
```

```
  int rio_fd;                /* Descriptor */
  int rio_cnt;               /* Num of unread bytes in internal buffer.  */
  char *rio_bufptr;          /* Next unread byte in internal buffer */
  char rio_buf[RIO_BUFSIZE]; /* Internal buffer */
} rio_t;

/* Called once per open descriptor */
void rio_readinitb(rio_t *rp, int fd) {
  rp->rio_fd = fd;
  rp->rio_cnt = 0;
  rp->rio_bufptr = rp->rio_buf;
}

/* Internal buffered read (buffered version of Linux read)*/
static ssize_t rio_read(rio_t *rp, char *userbuf, size_t n) {
  int cnt;
  while (rp->rio_cnt <= 0) {
    rp->rio_cnt = read(rp->rio_fd, rp->rio_buf, sizeof(rp->rio_buf));
    if (rp->rio_cnt < 0) {
      if (errno != EINTR)
        return -1;
    } else if (rp->rio_cnt == 0) {
      return 0;
    } else {
      rp->rio_bufptr = rp->rio_buf;
    }
    cnt = n < rp->rio_cnt ? n : rp->rio_cnt;
    memcpy(userbuf, rp->rio_bufptr, cnt);
    rp->rio_bufptr += cnt;
    rp->rio_cnt -= cnt;
    return cnt;
  }
}

/* Buffered readn */
ssize_t rio_readnb(rio_t *rp, void *userbuf, size_t n) {
  size_t nleft = n;
  ssize_t nread;
  char *bufp = userbuf;

  while (nleft > 0) {
    if ((nread = rio_read(rp, bufp, nleft)) < 0)//error
      return -1; //no need to check EINTR because it is guaranteed by rio_read
    else if (nread == 0) //EOF
      break;
    nleft -= nread;
    bufp += nread;
  }

  return n - nleft;
}

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen) {
  int n, rc;
  char c, *bufp = usrbuf;
  for(n = 1; n < maxlen; ++n) {
    rc = rio_read(rp, &c, 1);
    if(1 == rc) {
```

```
      *bufp++ = c;
      if('\n' == c) {
        ++n;
        break;
      }
    } else if (0 == rc) {
      if (1 == n)
        return 0;
      else
        break;
    } else
      return -1;
  }
  *bufp = '\0';
  return n - 1;
}
```

10.4 Read metadata of files

```
#include <unistd.h>
#include <sys/stat.h>
int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

`struct stat` contains `mode_t st_mode` that encodes the visit permission bits of the file, and `off_t st_size` which is the bytes of the file.

10.5 Read directory content

```
#include <sys/types.h>
#include <dirent.h>
struct direct {
  ino_t d_ino;       /* inode number */
  char  d_name[256]; /* file name */
};
DIR *opendir(const char *name);
/* DIR: directory stream. Stream is the abstraction of an ordered sequence of
    entries.*/
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

10.6 Sharing files

Descriptor table Each process has its own descriptor table indexed by the process's open file descriptors. Each fd points to an entry in the file table.

File table Shared by all processes. Each entry represents an open file, containing the current file position, reference count and a pointer to the v-node table. Closing an fd decrements the reference count of the file table entry it points to. The kernel won't delete the entry until its reference count is reduced to 0.

v-node table Shared by all processes. Each entry contains most information in the `stat` struct.

Each fd has its own current position, so different descriptors related to the same file point to separate entries in the file table, while the same fd in a parent process and its children processes point to the same entry.

10.7 IO redirection

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
/*Copies fd table entry of oldfd to that of newfd.*/
```

11 Network programming