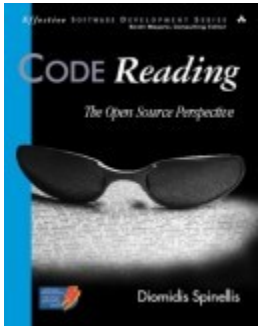
反璞歸真系列之

探究 UNIX v6 作業系統設計

Jim Huang (黃敬群) <jserv@0xlab.org>
Hideo Yamamoto a.k.a. ”magoroku15”

Aug 4, 2012 / JuluOSDev



”I wonder how many great novelists have never read someone else's work, ... painters ... another's brush strokes, ... skilled surgeons ... learned by looking over a colleague's shoulder, 767 captains ... in the copilot's seat ...”

(沒有研讀過其他作家作品的偉大作家，沒有研究過其他畫家筆法的偉大畫家，沒有盜取過並肩作戰的同事的技術的技巧高明的外科醫生，沒有在副駕駛的位置積累實際經驗的波音 767 的機長，在現實生活中真的會存在他們這樣的人嗎？)

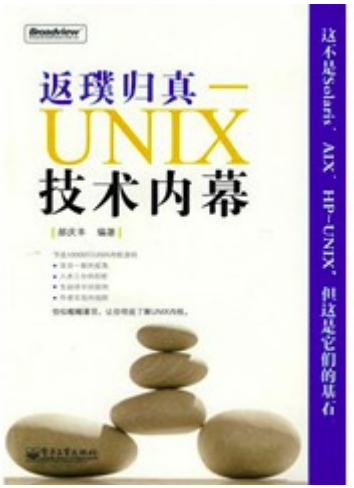
Dave Thomas 在《 Code Reading- The Open Source Perspective 》的序文
http://www.spinellis.gr/codereading/foreword.html

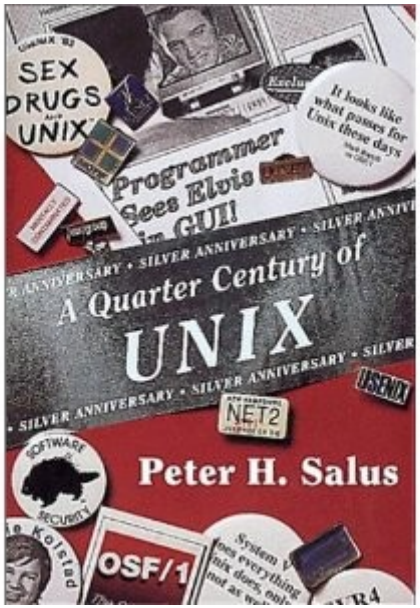
教材

- Unix version 6
  - Lions Commentary on UNIX
- 線上版本
  - http://v6.cuzuco.com/v6.pdf
  - http://www.lemis.com/grog/Documentation/Lions/book.pdf
- PDP-11/40
  - PDP-11 Processor Handbook
  - http://pdos.csail.mit.edu/6.097/readings/**pdp11-40**.pdf

參考書目

- 《返璞歸真 -UNIX 技術內幕》
- 《 A Quarter Century of UNIX 》
- 《黑客列傳：電腦革命俠客誌》





承先啟後的四個世代

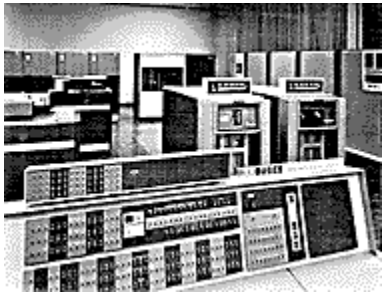
- 1st system (CTSS)
  - terrified of failure
  - simplified to bare bones
  - successful beyond its intended life-span
- 2nd system (Multics)
  - hugely ambitious
  - usually conceived by academics
  - many good ideas
  - a little ahead of its time
  - doomed to fail
- 3rd system (Unix)
  - pick and choose essence
  - usually made by good hackers
  - emphasize elegance and utility over performance and generality
  - become widely adopted
- 4th systems (BSD)
  - maturation

Source: CS560 – Advanced Operating Systems, Lecture 2: Multics and Unix

CTSS (Compatible Time-Sharing System; 1961-1973)

- MIT 推動 MAC 計畫 (Mathematics and Computation)，在 IBM 7090 主機上開發
- 可支援 32 使用者同時使用
- 實現了互動使令操作與除錯功能
- 硬體：
  - 0.35 MIPS; USD $3.5 million (IBM 7094)
  - 32 KB Memory made up of 36-bit words
  - Monitor uses 5KB words, leaving 27KB for users
  - User memory swapped between memory and fast drum
- Multi-level feedback queue scheduling

Source: http://www.multicians.org/thvv/7094.html



CTSS 的軟體特色

- 自批次系統移轉到多人分時
- 線上工作環境並提供線上儲存機制 (雲端概念？)
- 互動的 debugging, shell, editor
- 多個使用者間沒有保護機制

Multics (1964-1969-1985)

- 最初作為 CTSS 系統的延伸
- 由 MIT, GE (正在發展大型主機), Bell Labs (不能販售電腦卻人才濟濟的單位，負責承包軟體) 等單位合作開發
- 願景：

  “As larger systems become available, they will be connected by telephone wires to terminals in offices and homes throughout the city. The operating system would be a time-sharing system running continuously with a vast file system, just like electricity and water services”

1970 年 GE 售出電腦部門予 Honywell，後者在 1991 年售予 Groupe Bull

Multics

- 最初在 GE645 工作站開發
- 提出一套通用的 framework，採用 PL/1 高階程式語言開發
- 重新設計每項系統（《人月神話》的第二系統效應）
  - 客製化的硬體 + 新的程式語言 + 新的作業系統
- 影響
  - 雖然從未被廣泛採納，但探索了扣除電腦網路以外，幾乎所有作業系統設計的議題

PL/1 程式語言

- http://en.wikipedia.org/wiki/PL/I
- first specified in detail in the manual “PL/I Language Specifications. C28-6571” written in New York from 1965
- Sample program:

```
Hello2: proc options(main);
        put list ('Hello, world!');
end Hello2;
```

Multics 四大準則

- Naming and addressing
  - Combine virtual memory and file system
  - Filename + offset == Segment ID + offset
- Fine-grained sharing
  - Sharing procedures, not complete programs
- Dynamic Linking
  - Recompile without relinking
- Autonomy
  - Independent addresses for each application

UNIX

UNIX 公諸於世

“Perhaps the most important achievement of Unix is to demonstrate that a powerful operating system for interactive use need not be expensive…it can run on hardware costing as little as $40,000.” and less than two man-years were spent on the main system software.”

- **The UNIX Time-Sharing System,** D. M. Ritchie and K. Thompson (1974) ，對 Multics 的反思
  - http://cm.bell-labs.com/who/dmr/cacm.html

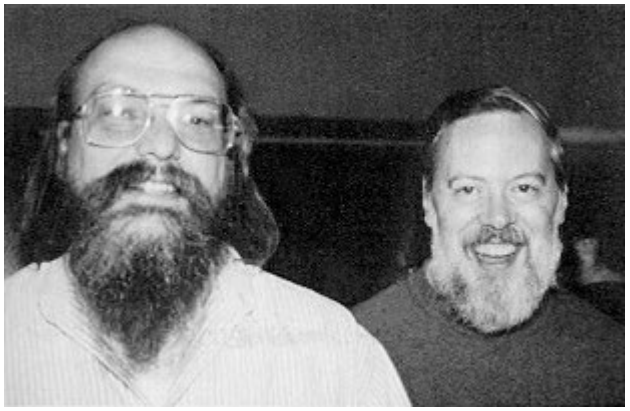
UNIX 兩大巨人

Kenneth Thompson

- Born in New Orleans in 1943
- Navy Brat
- Graduates with B.S. and M.S. (no Doctorate!) from UC Berkley in 1966
- Joins Bell Labs to work on Multics
  A mainframe timesharing operating system

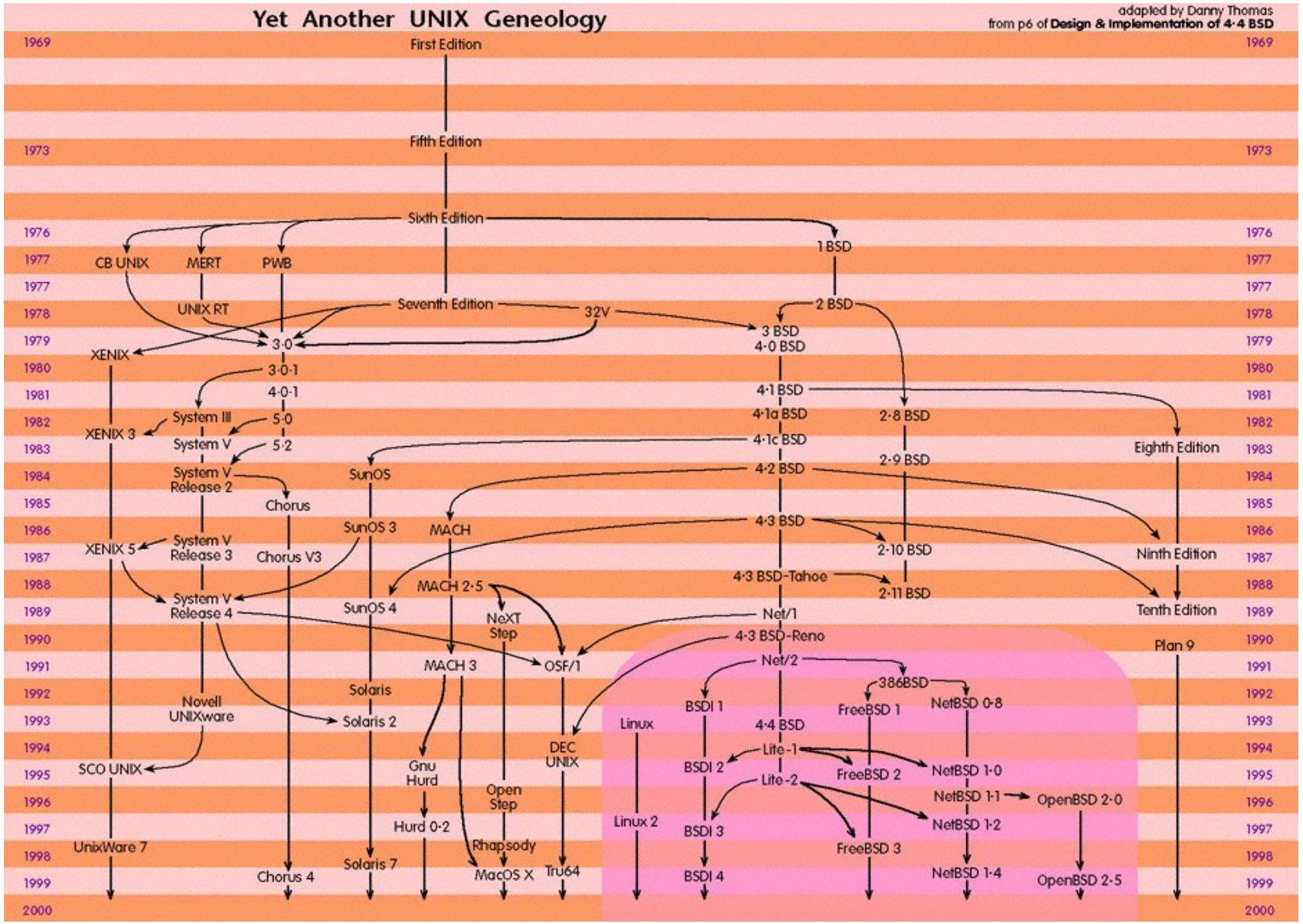
Dennis Ritchie

- Born in Bronxville N.Y. in 1941
- Graduates with B.S. in Physics, M.S. and Doctorate in Applied Math from Harvard in 1968
- Doctoral Thesis was on “Subrecursive Hierarchies of Functions.”
- Followed his father to work at Bell Labs



“My undergraduate experience convinced me that I was not smart enough to be a physicist, and that computers were quite neat”
~Dennis Ritchie

UNIX 家族


Yet Another UNIX Geneology
adapted by Danny Thomas
from p6 of Design & Implementation of 4·4 BSD
1969
First Edition
1973
Fifth Edition
1976
Sixth Edition
1977
CB UNIX
MERT
PWB
1977
UNIX RT
1978
Seventh Edition
32V
1979
XENIX
3·0
1980
3·0·1
1981
4·0·1
1982
System III
5·0
XENIX 3
1983
System V
5·2
1984
System V
Release 2
Chorus
1985
1986
System V
Release 3
XENIX 5
Chorus V3
1987
1988
System V
Release 4
1989
1990
1991
1992
Novell
UNIXware
Solaris
1993
Solaris 2
1994
1995
SCO UNIX
Gnu
Hurd
1996
Open
Step
1997
Hurd 0·2
UnixWare 7
1998
Solaris 7
Rhapsody
1999
Chorus 4
MacOS X
Tru64
2000
SunOS
SunOS 3
MACH
SunOS 4
MACH 2·5
NeXT
Step
MACH 3
OSF/1
DEC
UNIX
1 BSD
2 BSD
3 BSD
4·0 BSD
4·1 BSD
4·1a BSD
4·1c BSD
4·2 BSD
4·3 BSD
4·3 BSD-Tahoe
Net/1
4·3 BSD-Reno
Net/2
4·4 BSD
Lite·1
Lite·2
2·8 BSD
2·9 BSD
2·10 BSD
2·11 BSD
Eighth Edition
Ninth Edition
Tenth Edition
Plan 9
386BSD
BSDI 1
BSDI 2
BSDI 3
BSDI 4
FreeBSD 1
FreeBSD 2
FreeBSD 3
NetBSD 0·8
NetBSD 1·0
NetBSD 1·1
NetBSD 1·2
NetBSD 1·4
OpenBSD 2·0
OpenBSD 2·5
Linux
Linux 2

UNIX 背景

- 網路農夫的〈 UNIX 系統簡介〉：

  http://www.farmer.idv.tw/?viewDoc=479

- Micro implementations of Unix:

  http://www.robotwisdom.com/linux/nonnix.html

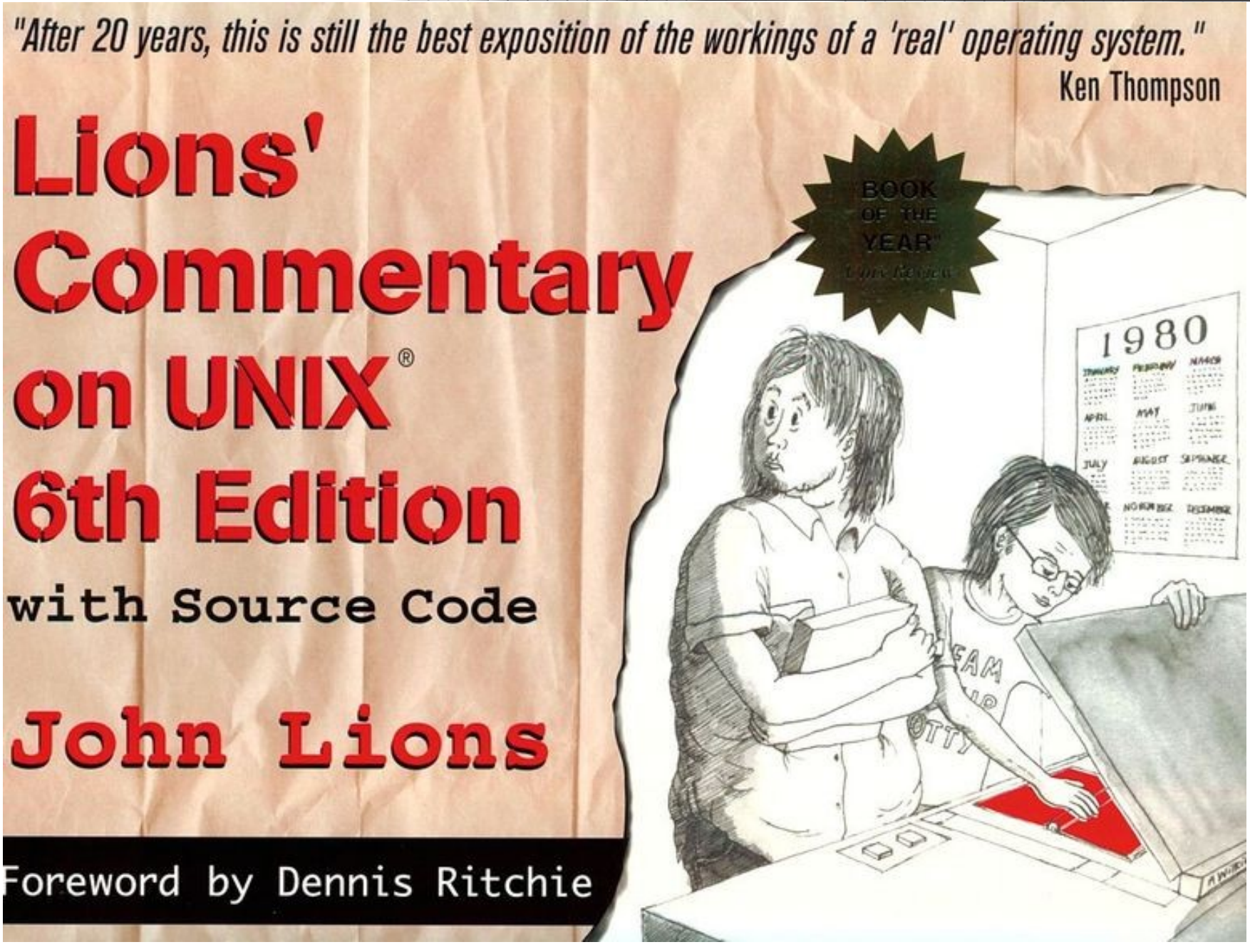
- The Strange Birth and Long Life of Unix

  http://spectrum.ieee.org/computing/software/the-strange-birth-and-long-life-of-unix/0

UNIX 初期自由發展的背景

- 依據 1958 年為了解決反托拉斯法案達成的和解協議， AT&T 不得進入電腦相關的商業領域，所以 UNIX 不能成為一種商品
- 並且依據和解協議， Bell Labs 必須將非電話業務的技術許可，提供給任何提出要求者
- AT&T 的主管階層對當時 UNIX 的發展未有太多支持與干預
- 為了應付 Bell Labs 內各部門日益增加的 UNIX 使用者與相關技術支援需求，成立 USG (UNIX System Group)

UNIX v6 廣泛的學術研究



UNIX v6 廣泛的學術研究

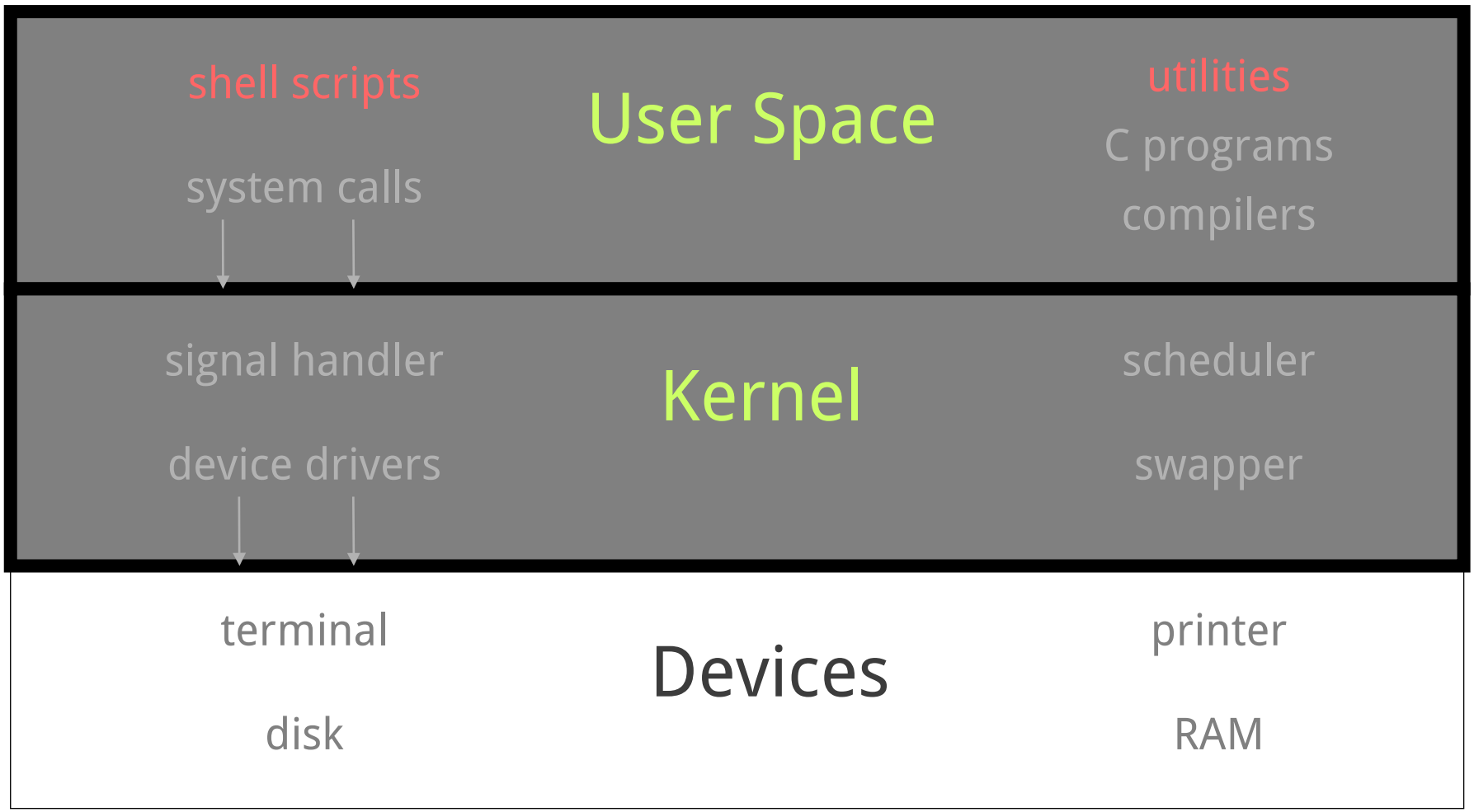
- 1976 年新南威爾斯大學 (也是催生 OKL4 與一系列重要作業系統研究的學術殿堂) John Lions 教授以註解的 UNIX v6 原始程式碼作為課堂教材，揭露 UNIX 簡潔又強大的設計
- AT&T 對 UNIX v7(含) 之後的版本採取嚴苛的授權條件，而 v6 則是允許課堂使用的最後一個版本
- 在 1996 年再次印刷，至今仍販售
- 2002 年一月份， Caldera International (目前為 SCO) 對以下 UNIX 版本重新授權，變更為 BSD License ，自此研究不再受限
  - UNIX/32V, UNIX Version 1-7
  - http://en.wikipedia.org/wiki/Ancient_UNIX

Mutlics 與 UNIX 簡短比較

- 研發資源投入量
  - ? Work years vs. 2 man-years
- 關鍵概念

Feature	Multix	Unix
Key abstraction	Unify file = memory	Unify I/O = file
Protection	rings (jump to memory)	suid (execute file)
Sharing	N segments; arbitrary sharing	3 segments (text, heap, stack) text is shared (RO); Communication between domains via files, pipes

UNIX 結構


shell scripts
system calls
User Space
utilities
C programs
compilers
signal handler
device drivers
Kernel
scheduler
swapper
terminal
disk
Devices
printer
RAM

UNIX 原始硬體組態

- 最早在 DEC PDP-7 主機上開發，稍候移到 PDP-11/20
- 在 PDP-11 之前的硬體並非以 byte (8 bits) 為基礎
  - PDP-7: 18-bit
  - PDP-6: 36-bit
  - PDP-5: 12-bit
- 為什麼當初會有 18-bit 與 36-bit 的系統呢？
  - “Many early computers aimed at the scientific market had a 36-bit word length. ...just long enough to represent positive and negative integers to an accuracy of ten decimal digits... allowed the storage of six alphanumeric characters encoded in a six-bit character encoding.”

Source: http://en.wikipedia.org/wiki/36-bit

PDP-11 系列的硬體

- 高度正規化指令集的設計
  - 指令集中的定址模式可以視為一種「基底」
  - 指令集中的 opcode 則是另一個基底
- 16-bit 硬體，為 DEC 第一款使用 byte 操作的系統
- Unibus: CPU 對所有設備的存取皆經由單一 Bus
  - 所有設備都具有統一的定址規則，包含 memory, register, device register 等都具有統一的地址
- UNIX “First Edition” 運作於 PDP-11/20 型電腦
  - 512K bytes 硬碟
  - UNIX 提供 16K bytes 給系統、 8K bytes 給程式，檔案上線為 64K bytes

程式語言變遷

- http://en.wikipedia.org/wiki/BCPL
- BCPL 最早用於撰寫其他程式語言的 compiler，前身為 CPL(Combined Programming Language)， CPL 在 1960 年代前期提出，但首個編譯器在 1970 年實做 ）。 1967 年早期， Martin Richards 移除 CPL 之中不易實做的特性，在 IBM 7094 / CTSS 環境實做出第一個 BCPL 語言編譯器
-

```
GET "LIBHDR"
LET START() = VALOF $(
        FOR I = 1 TO 5 DO
                WRITEF("%N! = %I4*N", I, FACT(I))
        RESULTIS 0
)$
AND FACT(N) = N = 0 -> 1, N * FACT(N - 1)
```

BCPL 程式語言

- Typeless
  - Everything an n-bit integer (a machine word)
  - Pointers (addresses) and integers identical
- Memory is an undifferentiated array of words
- Natural model for word-addressed machines
- Local variables depend on frame-pointer-relative addressing: dynamically-sized automatic objects not permitted
- Strings awkward
  - Routines expand and pack bytes to/from word arrays

Source: The C Language, Prof. Stephen A. Edwards

從 B 語言演化

- (from Wikipedia) Like BCPL and FORTH, B had only one datatype: the computer word. Most operators (e.g., +, -, *, /) treated this as an integer, but others treated it as a memory address to be dereferenced.
- The typeless nature of B made sense on older computers, but was a problem on the PDP-11 because it was difficult to elegantly access the character data type that the PDP-11 and most modern computers fully support.
- During 1971 and 1972 B evolved into "New B" and then C, with the preprocessor being added in 1972. (B lacks of structures.)
-

```
printn(n,b) {
        extrn putchar;
        auto a;

        if(a=n/b) /* assignment, not test for equality */
                printn(a, b); /* recursive */
        putchar(n%b + '0');
}
```

PDP-11 was byte-addressed (now standard).
Meant BCPL's word-based model was insufficient

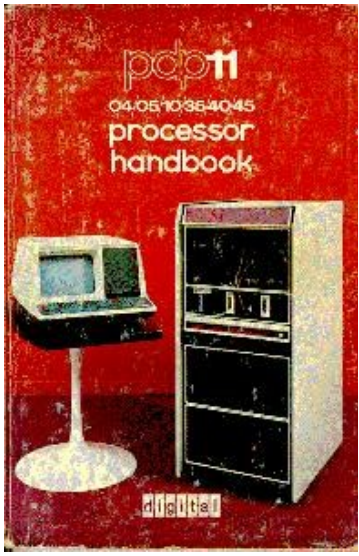
C 語言與 PDP-11

- PDP-11 was byte-addressed.
- Pointer arithmetic is natural: everything is an integer

  `int *p, *q;`

  `*(p+5)` equivalent to `p[5]`
- If p and q point into same array, `p - q` is number of elements between p and q.
- Accessing fields of a pointed-to structure has a shorthand:

  `p->field` means `(*p).field`
- Mapped to PDP-11 instructions transparently



C 語言與 PDP-11

- 透過 PDP-11 的暫存器定址模式的增值 / 減值語法，在 C 語言中，若 i 與 j 都是 register variable，那麼

```
*(--i) = *(j++)
```

  指令可編譯為單一機器碼指令
- 由於 PDP-11 對單精確與雙精確浮點數沒有不同的運算碼，造成 C 語言中缺乏單精確浮點數運算的運算模式
- 由 BCPL, B, 一直過渡到 C 語言的設計考量，可見 Dennis Ritchie 撰寫的〈 The Development of the C Language 〉
  - http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

V7: Portable UNIX

- 執行於 PDP-11 及 Interdata 8/32 型電腦
- Porting required:
  - write a C compiler for the new machine,
  - write device drivers for the new machine's I/O devices such as printers, terminals, disks
  - Small amount of machine-dependent code such as the interrupt handlers, memory management routines must be rewritten, usually in assembly code
- Problem: Ritchie's compiler – good, but produced PDP-11 object code.
  - Steve Johnson at Bell Labs implemented portable C compiler, called also Johnson's compiler

UNIX V32 與 DEC VAX

- 當時 DEC 公司推出了一款 32-bit supermini 主機 VAX， 搭配的 VAX 的作業系統為 VMS
- Bell Labs 的工程師們寧願使用 UNIX ，移植工作由 John Reiser 和 Tom London 共同完成。以 V7 為基礎移植 UNIX 到 VAX 電腦上使用，這個版本稱為 UNIX V32

System introduced in 1977

VAX – the “Virtual Address eXtension” of the PDP-11's 16-bit architecture to a 32 bit architecture

作業系統尺寸比較

Year	AT&T	BSD	MINIX	Linux	Solaris	Win NT
1976	V6 9K					
1979	V7 21K					
1980		4.1 38K				
1982	Sys III 58K					
1984		4.2 98K				
1986		4.3 179K				
1987	SVR3 92K		1.0 13K			
1989	SVR4 280K					
1991				0.01 10K		
1993		Free 1.0 235K			5.3 850K	3.1 6M
1994		4.4 Lite 743K		1.0 165K		3.5 10M
1996				2.0 470K		4.0 16M
1997			2.0 62K		5.6 1.4M	
1999				2.2 1M		
2000		Free 4.0 1.4M			5.8 2.0M	2000 29M

UNIX 標準化

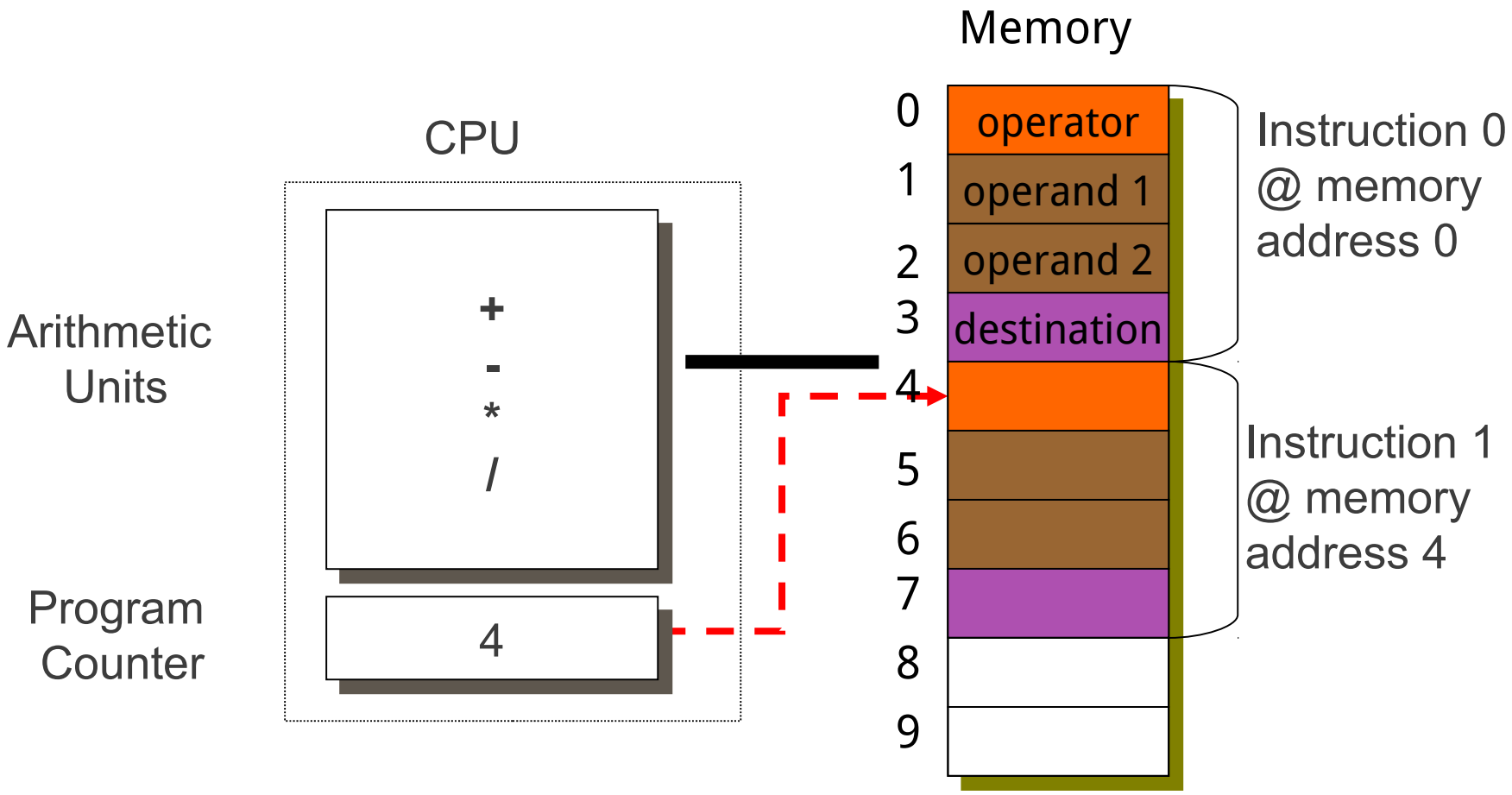
- POSIX: Portable Operating System Interface
- POSIX is a popular standard for Unix-like operating systems.
- POSIX is actually a *collection* of standards that cover system calls, libraries, applications and more…
- POSIX 1003.1 defines the C language interface to a Unix-like kernel.

基礎 PDP-11 指令集

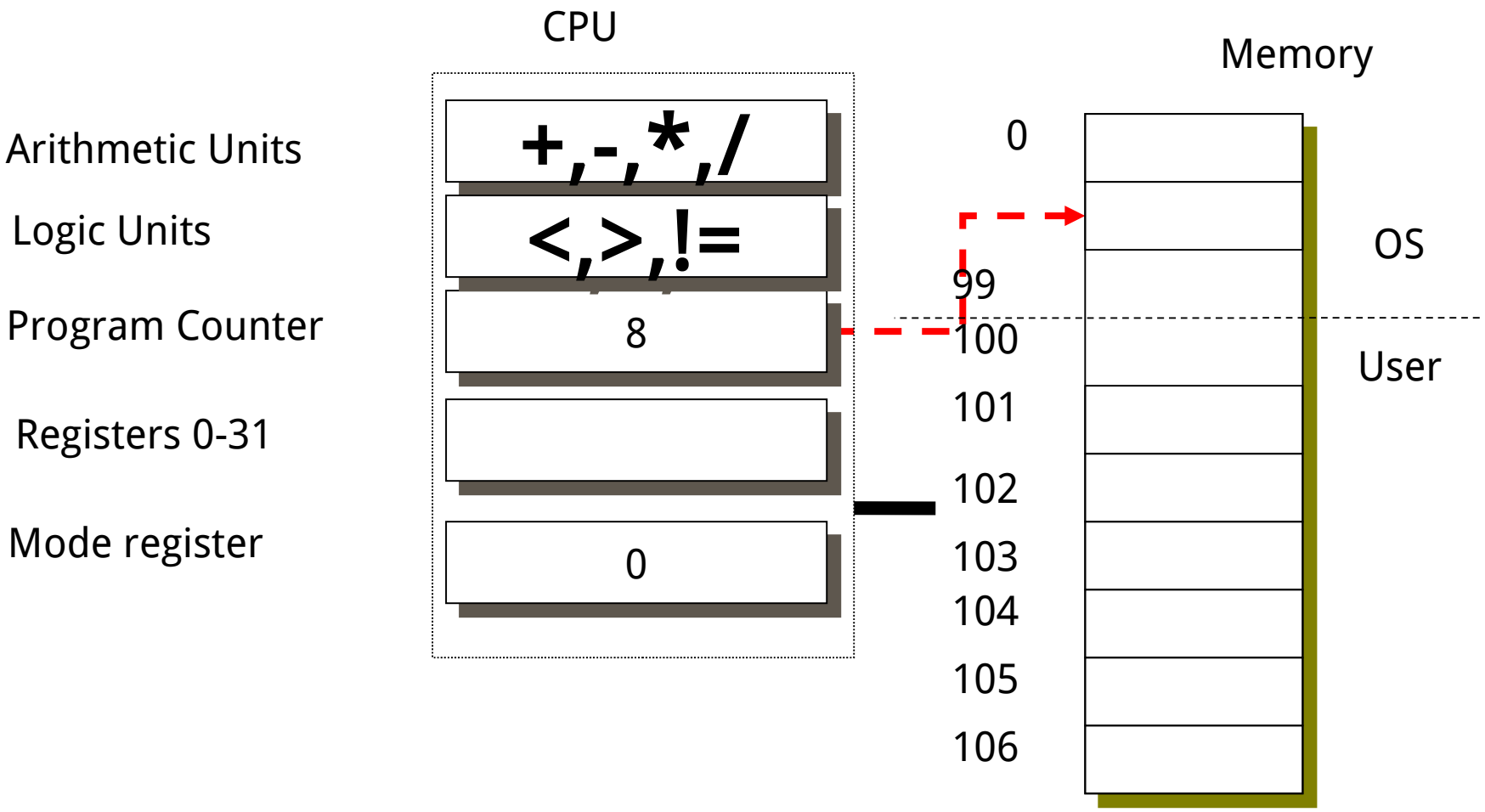
系統程式開發的觀點，看電腦組成

- CPU
  - Register
  - General-purpose register
  - Stack pointer: The top of the stack position
  - Program counter: Position of the instruction to be processed next
  - Flag register: Part of the operation result storage
  - Control / Status Register
- Memory
- 外部記憶

電腦模型概念


Memory
CPU
Arithmetic Units
+
-
*
/
Program Counter
4
0
1
2
3
4
5
6
7
8
9
operator
operand 1
operand 2
destination
Instruction 0 @ memory address 0
Instruction 1 @ memory address 4

簡化的 CPU 保護模式


CPU
Memory
Arithmetic Units
Logic Units
Program Counter
Registers 0-31
Mode register
+,-,*,/
<,>,!=
8
0
0
99
100
101
102
103
104
105
106
OS
User

Examples: Data Movement

Instruction	Meaning	Machine
MOV A, B	Move 16 bits from memory location A to Location B	VAX11
LDA A, Addr	Load accumulator A with the byte at memory M6800 location Addr	M6800
lwz R3, A	Move 32-bit data from memory location A to register R3	PPC601
li $3, 455	Load the 32-bit integer 455 into register $3	MIPS R3000
mov R4, dout	Move 16-bit data from R4 to output port dout	DEC PDP11
IN, AL, KBD	Load a byte from in port KBD to accumulator	Intel Pentium
LEA.L (A0), A2	Load the address pointed to by A0 into A2	M6800

Source: Computer Systems Design and Architecture by V. Heuring and H. Jordan

Examples: ALU

Instruction	Meaning	Machine
MULF A, B, C	multiply the 32-bit floating point values at mem loc'ns. A and B, store at C	VAX11
nabs r3, r1	Store abs value of r1 in r3	PPC601
ori $2, $1, 255	Store logical OR of reg $ 1 with 255 into reg $2	MIPS R3000
DEC R2	Decrement the 16-bit value stored in reg R2	DEC PDP11
SHL AX, 4	Shift the 16-bit value in reg AX left by 4 bit pos'ns	Intel 8086

Examples: Branch

Instruction	Meaning	Machine
BLSS A, Tgt	Branch to address Tgt if the least significant bit of mem loc'n. A is set (i.e. = 1)	VAX11
bun r2	Branch to location in R2 if result of previous floating point computation was Not a Number (NAN)	PPC601
beq $2, $1, 32	Branch to location (PC + 4 + 32) if contents of $1 and $2 are equal	MIPS R3000
SOB R4, Loop	Decrement R4 and branch to Loop if R4 ≠ 0	DEC PDP11
JCXZ Addr	Jump to Addr if contents of register CX ≠ 0	Intel 8086

PDP-11/40

- Register
  - 通用 16bit register 共 8 組 : r0-r7
  - r6 is the stack pointer (sp)
  - r7 is the program counter (pc)
- Processor Status Word
  - Flag register + status register
- Memory
  - virtual address width: 16bit 最大 64Kbyte
  - physical address width: 18bit 最大 256Kbyte

為有效使用 physical 記憶體空間，透過 Page 機制，將 virtual 記憶體對映到 physical 空間

Register r0-r5

- r0, r1

  used as temporary accumulators during expression evaluation, to return results from a procedure, and in some cases to communicate actual parameters during a procedure call

- r2, r3, r4

  used for local variables during procedure execution. Their values are almost always stored upon procedure entry, and re-

- stored upon procedure exit

- r5 (frame pointer)

  used as the head pointer to a “dynamic chain” of procedure activation records stored in the current stack. It is referred to as the “environment pointer”.

Register r6-r7

- r6 (stack pointer)

  used as the stack pointer. The PDP11/40 processor incorporates two separate registers which may be used as “sp”, depending on whether the processor is in kernel or user mode. No other one of the general registers is duplicated in this way

- r7 (pc)

  used as the program instruction address register.

PDP-11 記憶體定址

- 16-bit words, 8-bit bytes; byte-addressable to 64 K bytes; little-endian
- "effective address" (EA)
  - Given an ADD instruction, with two operands,
  - we get two effective addresses, and the meaning of the instruction will be EA1 <- [EA1]+[EA2].

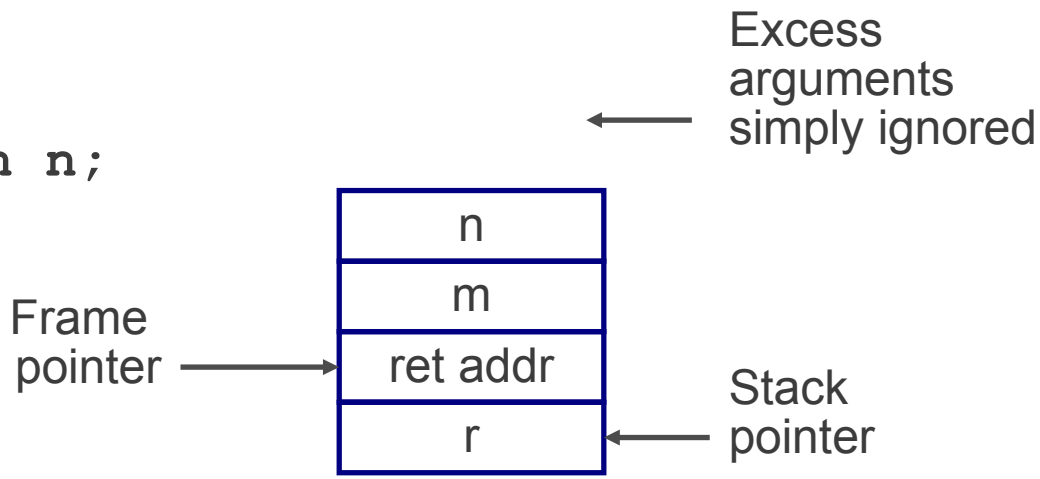
name	assembly syntax	EA and other semantics
register	Ri	Ri
autoincrement	(Ri)+	[Ri], then Ri<-[Ri]+2 or 1
autodecrement	-(Ri)	first Ri<-[Ri]-2 or 1, then EA=new [Ri]
index	n(Ri)	[[R7]]+[Ri], then inc PC by 2 (n follows in next memory word)

++ / -- operator 的由來

- 〈 The Development of the C Language 〉的解釋：
  - “Thompson went a step further by inventing the ++ and -- operators, which increment or decrement; their prefix or postfix position determines whether the alteration occurs before or after noting the value of the operand. They were not in the earliest versions of B, but appeared along the way. People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular.”
- ++ / -- operator 設計動機肇因於 PDP-11 的 auto-increment

C 語言版本之最大公因數（輾轉相除法）

```
int gcd(int m, int n)
{
  int r;
  while ( (r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```



Automatic variable

Storage allocated on stack when function entered, released when it returns.

All parameters, automatic variables accessed without frame pointer.

Extra storage needed while evaluating large expressions also placed on the stack

C 語言版本之 GCD

```
int gcd(int m, int n)
{
  int r;
  while ( (r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

High-level control-flow statement. Ultimately becomes a conditional branch.

Supports “structured programming”

Each function returns a single value, usually an integer. Returned through a specific register by convention.

GCD 在 PDP-11 架構的編譯輸出

```
.globl  _gcd
.text                   / PC is r7, SP is r6, FP is r5
_gcd:
    jsr   r5,rsave      / save sp in frame pointer r5
L2:mov   4(r5),r1       / r1 = n
    sxt   r0            / sign extend
    div   6(r5),r0      / m / n = r0,r1
    mov   r1,-10(r5)    / r = m % n
    jeq   L3
    mov   6(r5),4(r5)   / m = n
    mov   -10(r5),6(r5) / n = r
    jbr   L2
L3:mov   6(r5),r0       / return n in r0
    jbr   L1
L1:jmp   rretrn         / restore sp ptr, return
```

```
int gcd(int m, int n)
{
  int r;
  while ( (r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

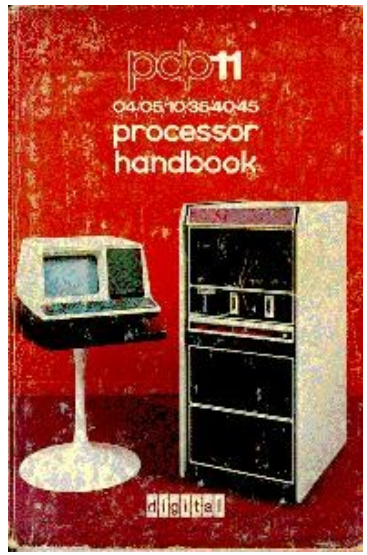
GCD 在 PDP-11 架構的編譯輸出

```
.globl  _gcd
.text
_gcd:
    jsr   r5,rsave
L2:mov   4(r5),r1
    sxt   r0
    div   6(r5),r0
    mov   r1,-10(r5)
    jeq   L3
    mov   6(r5),4(r5)
    mov   -10(r5),6(r5)
    jbr   L2
L3:mov   6(r5),r0
    jbr   L1
L1:jmp   rretrn
```

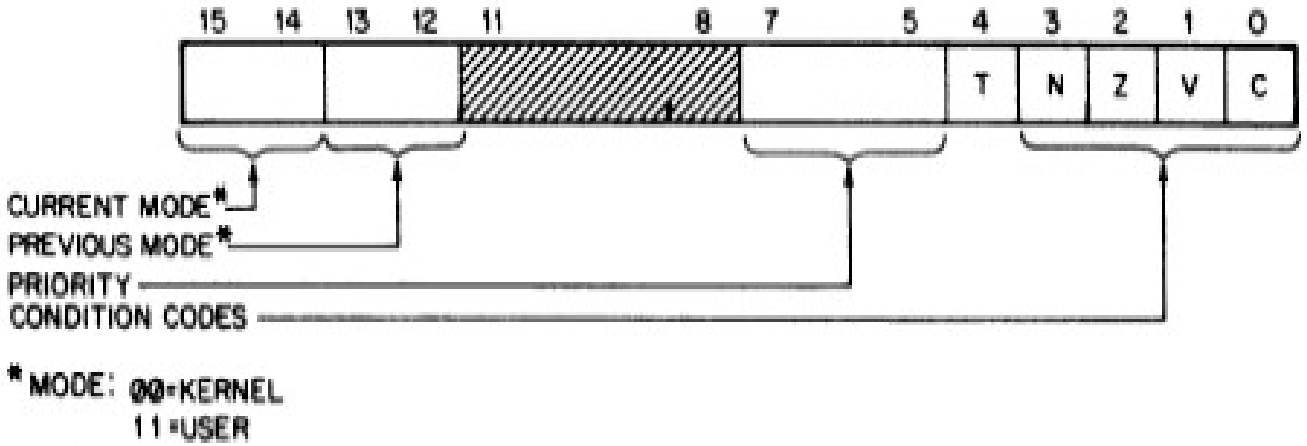
Very natural mapping from C into PDP-11 instructions.

Complex addressing modes make frame-pointer-relative accesses easy.

Another idiosyncrasy: registers were memory-mapped, so taking address of a variable in a register is straightforward.



Processor Status Word (PSW)

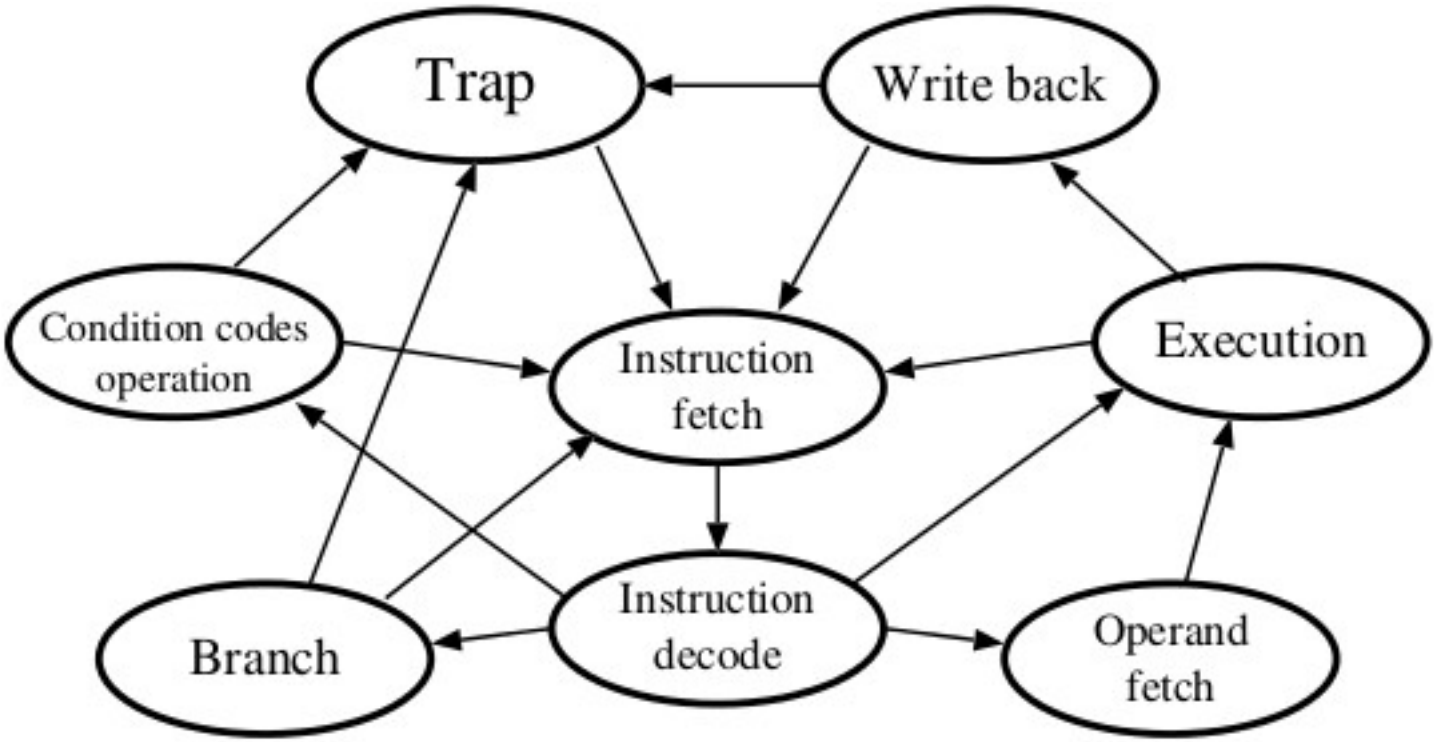


14-15	current mode (00 = kernel; 11 = user)
12-13	previous mode
5-7	processor priority (range 0..7)
4	trap bit
3 N	set if the previous result was negative
2 Z	set if the previous result was zero
1 V	set if the previous result gave an overflow
0 C	set if the previous operation gave a carry

CPU Processing

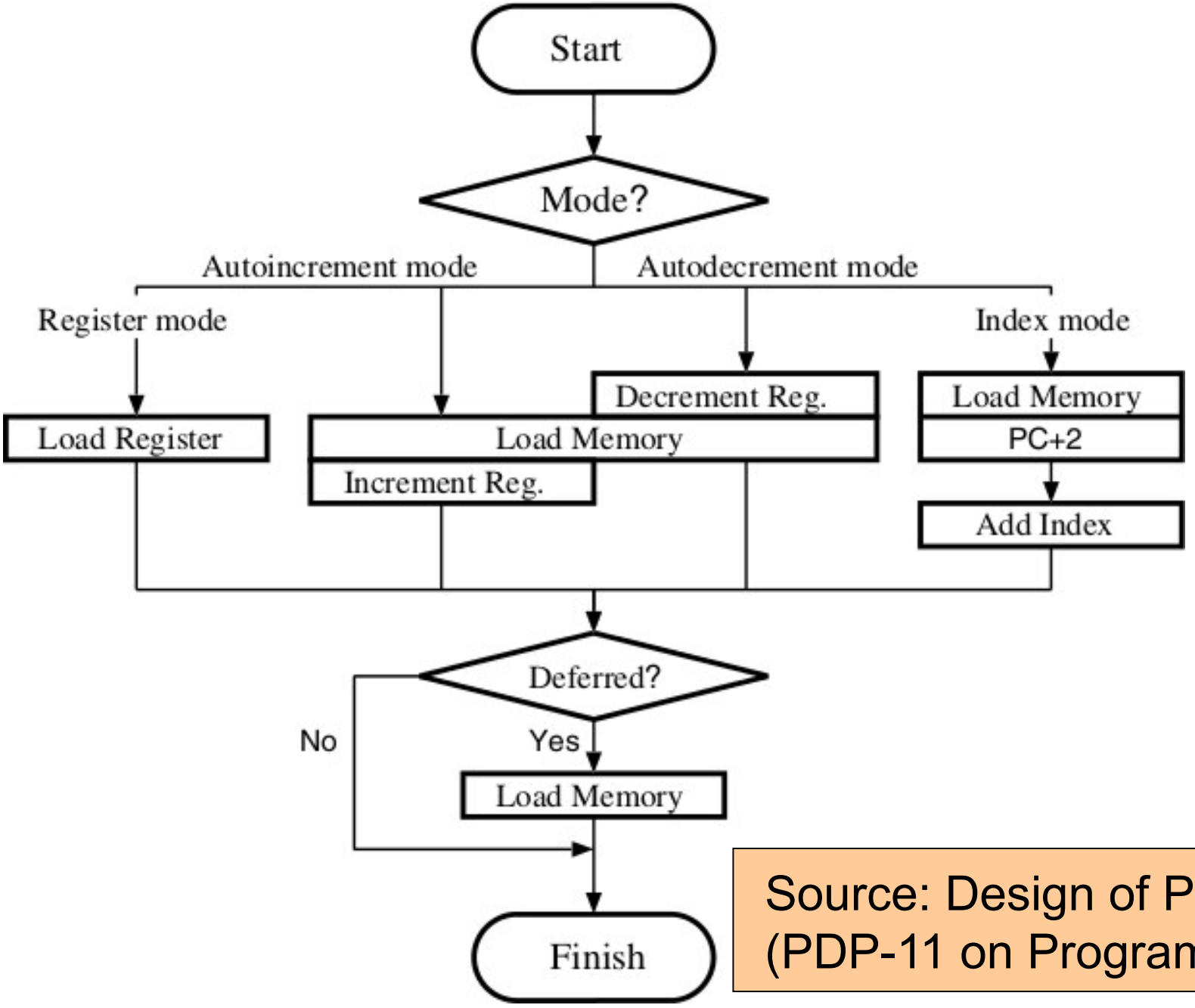
- Repeat the following
  - Fetch instructions from the location (address) indicated by the PC
  - Decode
  - Execute
- The following events occur as a side effect of execution
  - Rewrites PSW
  - Rewrite PC
  - Rewrite register
  - Or rewrite the contents of the memory

State transition of PDP-11/40



Source: Design of POP-11 (PDP-11 on Programmable chip)

Addressing model of PDP-11/40



Source: Design of POP-11
(PDP-11 on Programmable chip)

CPU Processing: Misleading point

- PC is also a kind of register
  - Jump to rewrite the PC and processing (r7)
- Affected by Comparisons and another branch instruction

原始 C 語言與 ANSI C 的落差

原始 C 語言：運算符號與型態長度

- +=, -=, /= 一類的運算符號寫法不同，記為 =+, =-, =/
- 當時沒有 long 型態，為描述長度大於 1 個 word (16-bits) 的資料時，往往藉由 2 個變數來存放。如 inode.h

```
5659: struct  inode {
…
5670:    char i_size0; /* most significant of size */
5671:    char *i_size1; /* least sig */
```

- 或者用兩個 word 來模擬一個 32-bit 整數

原始 C 語言：指標操作

- ANSI C 語言為強制型態的程式語言，但原始的 C 語言則略為鬆散。型態的落差，反映於指標 (pointer) 的操作
- 對指標型態鬆散處理的好處是，使用可以很靈活，如：

```
0164: #define PS 0177776   // 0177776 即 PS 暫存器的邏輯地址
      // 原始程式碼中有大量此類 device register 的定義，
      // 其特點為皆位於第 8 個邏輯 page( 會被映射到高位址空間 )
0175: struct { int integ; };
2070: s = PS->integ;
```

- PS 僅是個常數地址，無需定義即可透過指標方式進行存取，並且透過一個匿名 struct， PS 得到指標類型：指向 int 的指標

原始 C 語言：鬆散的指標型態

- 指標型態鬆散處理的另一個例子: (slp.c)

```
2156: setpri(up)
2157: {
2158:         register *pp, p;
2159:
2160:         pp = up;
2161:         p = (pp->p_cpu & 0377)/16;
2162:         p =+ PUSER + pp->p_nice;
```

- 指標 **register *pp** 在未作任何轉型的狀態，即可存取到 **p_** 開頭的結構成員，而這原本是定義在 **struct proc** (位於 proc.h)

原始 C 語言：鬆散的指標型態

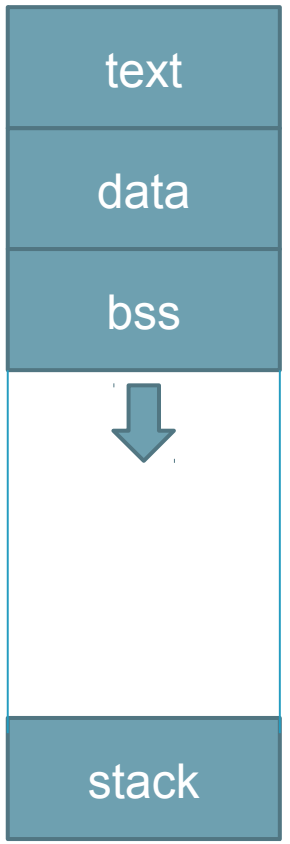
- 依據 UNIX Version 6 的 C Reference Manual，作以下解說：
  - 7.1.8 primary-expression -> member-of-structure

    The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the **primary-expression need not in fact be pointer**, it is sufficient that it be a pointer, character, or integer.
- 與 ANSI C 的規範不同，原始 C 語言的 -> 操作著重於「基於特定 offset 的資料存取」，而成員名稱只是用以表示 offset

Function call & Stack

C 語言的記憶體配置

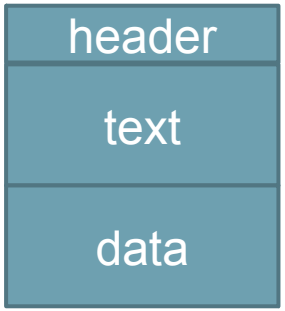
- text
  - 保存執行的 instructions
  - 禁止改寫
- data
  - data area is initialized with a non-zero
- bss
  - data area is initialized with 0
- Stack
  - 保存自動 (auto) 變數
  - Register save area at the time of the function call

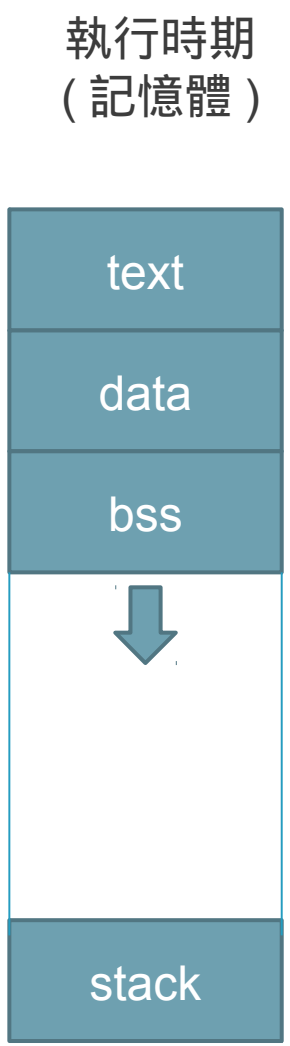


宣告變數時的記憶體配置

```
int i;          // Bss
int j = 1;      // Data


main()
{
    static int k = 1;    // Data
    int l = 1;           // Stack
    int m;               // Stack
    ...
}
```





Function calls and stack in C

```
main()
{
    int e, f, r;
    e = 1;
    f = 2;
    r = func(e, f);
}
```

```
int c, d;


func(a, b)
{
    c = a;
    d = b;
    return  c + d;
}
```

Function and stack in namelist

```
# nm -n a.out
000000a  0x00  crt0.o
000000t        start
000004a        a
000006a        b
000030T  0x18  _main
000030a        src.o
000030t        ~main
000102T  0x42  _func
000102t        ~func
000140T  0x60  _exit
000140a        exit.o
000152T  0x6a  csv
000152a        csv.o
000170T  cret 0x78
000206B  _c
000210B  _d
000212B  savr5
177764a  r
177766a  f
177770a  e
```

Function and stack in Assembly

```
.globl  _main
.text
_main:
~~main:
~e=177770
~f=177766
~r=177764
jsr     r5,csv
sub     $6,sp
mov     $1,-10(r5)
mov     $2,-12(r5)
mov     -12(r5),(sp)
mov     -10(r5),-(sp)
jsr     pc,*$_func
tst     (sp)+
mov     r0,-14(r5)
L1:     jmp     cret
```

```
.globl  _c
.comm   _c,2
.globl  _d
.comm   _d,2
.globl  _func
.text
_func:
~~func:
~a=4
~b=6
jsr     r5,csv
mov     4(r5),_c
mov     6(r5),_d
mov     _c,r0
add     _d,r0
jbr     L2
L2:     jmp  cret
.globl
.data
```

csv: Function entry

```
main:
        :
        jsr  r5,csv
        :
```

```
csv:
        mov     r5,r0
        mov     sp,r5
        mov     r4,-(sp)
        mov     r3,-(sp)
        mov     r2,-(sp)
        tst     -(sp)
        jmp     (r0)
```

```
      r0    r1    r2    r3    r4    r5   r6:sp r7:pc
---- 0484, 0000, 0000, 0000, 0000, 0000, 047e, 0018  jsr r5,006a
---- 0484, 0000, 0000, 0000, 0000, 001c, 047c, 006a  mov r5,r0
---- 001c, 0000, 0000, 0000, 0000, 001c, 047c, 006c  mov sp,r5
---- 001c, 0000, 0000, 0000, 0000, 047c, 047c, 006e  mov r4,-(r6)
-z-- 001c, 0000, 0000, 0000, 0000, 047c, 047a, 0070  mov r3,-(r6)
-z-- 001c, 0000, 0000, 0000, 0000, 047c, 0478, 0072  mov r2,-(r6)
-z-- 001c, 0000, 0000, 0000, 0000, 047c, 0476, 0074  tst -(r6)
-z-- 001c, 0000, 0000, 0000, 0000, 047c, 0474, 0076  jmp (r0)
```

csv: Function exit

```
main:
     :
     jmp      cret
     :
```

```
cret:
          mov      r5,r1
          mov      -(r1),r4
          mov      -(r1),r3
          mov      -(r1),r2
          mov      r5,sp
          mov      (sp)+,r5
          rts      pc
```

```
     r0    r1    r2    r3    r4    r5    r6:sp r7:pc
---- 0003, 0462, 0000, 0000, 0000, 047c, 046e, 003e jmp 0078
---- 0003, 0462, 0000, 0000, 0000, 047c, 046e, 0078 mov r5,r1
---- 0003, 047c, 0000, 0000, 0000, 047c, 046e, 007a mov -(r1),r4
-z-- 0003, 047a, 0000, 0000, 0000, 047c, 046e, 007c mov -(r1),r3
-z-- 0003, 0478, 0000, 0000, 0000, 047c, 046e, 007e mov -(r1),r2
-z-- 0003, 0476, 0000, 0000, 0000, 047c, 046e, 0080 mov r5,r6
---- 0003, 0476, 0000, 0000, 0000, 047c, 047c, 0082 mov (r6)+,r5
-z-- 0003, 0476, 0000, 0000, 0000, 0000, 047e, 0084 rts 0084
```

JSR: Jump to SubRoutine

- jsr src,dst
- 等價於以下指令
  - MOV src,-(R6)　src　　(push onto the stack)
  - MOV PC,src (Transfer to PC of the next instruction src)
  - JMP dst　　(jump to the dst)
- jsr r5, 0x006a 作以下處理
  - push r5, saves the return address (= PC, the address of the next instruction) in r5
  - jumps to the address of 0x006a
  - It follows that if r5 is the PC, jsr simply pushes the PC and jumps.

RTS – ReTurn from Subroutine

- rts src

  等價於以下指令

  - MOV src,PC
  - MOV (R6)+,src
- rts r7 作以下處理
  - restores the PC from r7 and pops r7.
  - It follows that if r7 is the PC, rts just pops it.
  - Jump by rewriting the pc

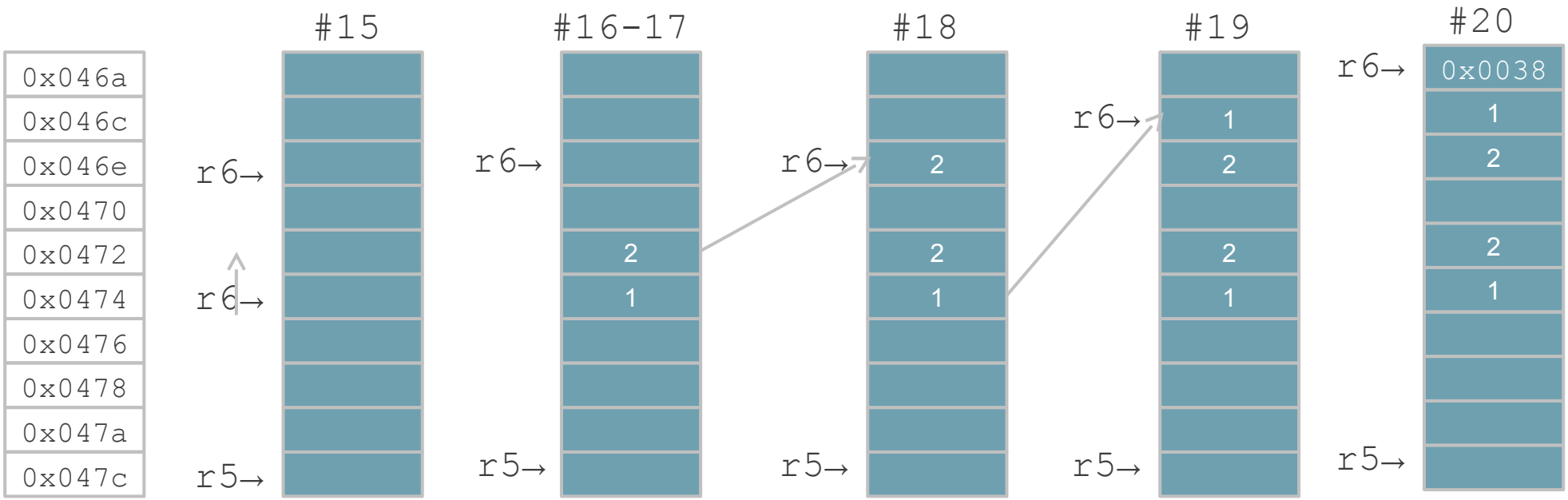
Subroutine Linkage and Jump

Instruction	Read + Machine code	Operations in C
jmp	jump 0001 dd	PC = &d (a−mode > 0)
rts	return from subroutine 00020 r	PC = R ; R = ∗SP++
jsr	jump subroutine 004 r dd	∗−−SP = R ; R = PC ; PC = &d (a−mode > 0)
mark	mark 0064 nn	SP = PC + 2∗ nn ; PC = FP ; FP = ∗SP++
sob	subtract one 077 r nn	R = R − 1 ; i f R ! = 0 : PC = PC − 2∗ nn

Source: PDP-11 - CPU, BA Stuttgart

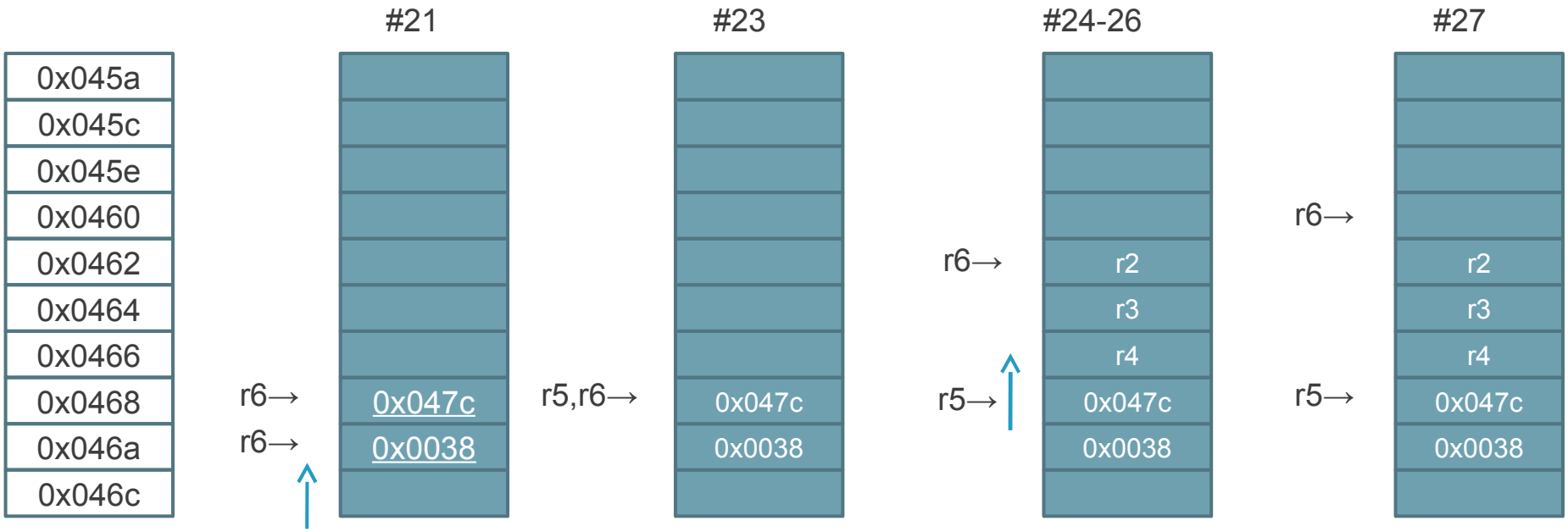
main 引數的設定

```
 #    r0    r1    r2    r3    r4    r5    r6    r7
15: 001c,0000,0000,0000,0000,047c,0474,001c sub $6,r6          main
16: 001c,0000,0000,0000,0000,047c,046e,0020 mov $1,-8(r5)      main
17: 001c,0000,0000,0000,0000,047c,046e,0026 mov $2,-a(r5)      main
18: 001c,0000,0000,0000,0000,047c,046e,002c mov -a(r5),(r6)    main
19: 001c,0000,0000,0000,0000,047c,046e,0030 mov -8(r5),-(r6)   main
20: 001c,0000,0000,0000,0000,047c,046c,0034 jsr r7,*$0x0040    main
21: 001c,0000,0000,0000,0000,047c,046a,0042 jsr r5,0x006a      func
```



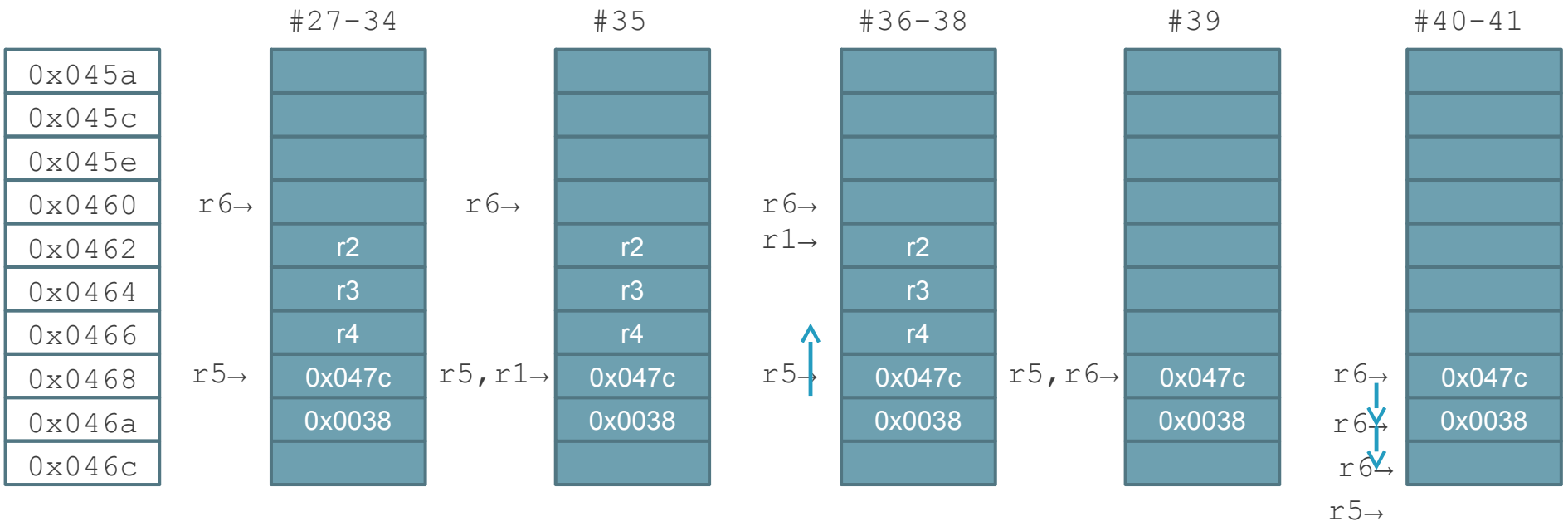
func 進入點 (func-csv)

```
 #    r0    r1    r2    r3    r4    r5    r6    r7
21: 001c,0000,0000,0000,0000,047c,046a,0042   jsr r5,0x006a    func
22: 001c,0000,0000,0000,0000,0046,0468,006a   mov r5,r0       func-csv
23: 0046,0000,0000,0000,0000,0046,0468,006c   mov r6,r5       func-csv
24: 0046,0000,0000,0000,0000,0468,0468,006e   mov r4,-(r6)    func-csv
25: 0046,0000,0000,0000,0000,0468,0466,0070   mov r3,-(r6)    func-csv
26: 0046,0000,0000,0000,0000,0468,0464,0072   mov r2,-(r6)    func-csv
27: 0046,0000,0000,0000,0000,0468,0462,0074   tst -(r6)       func-csv
28: 0046,0000,0000,0000,0000,0468,0460,0076   jmp (r0)        func
```



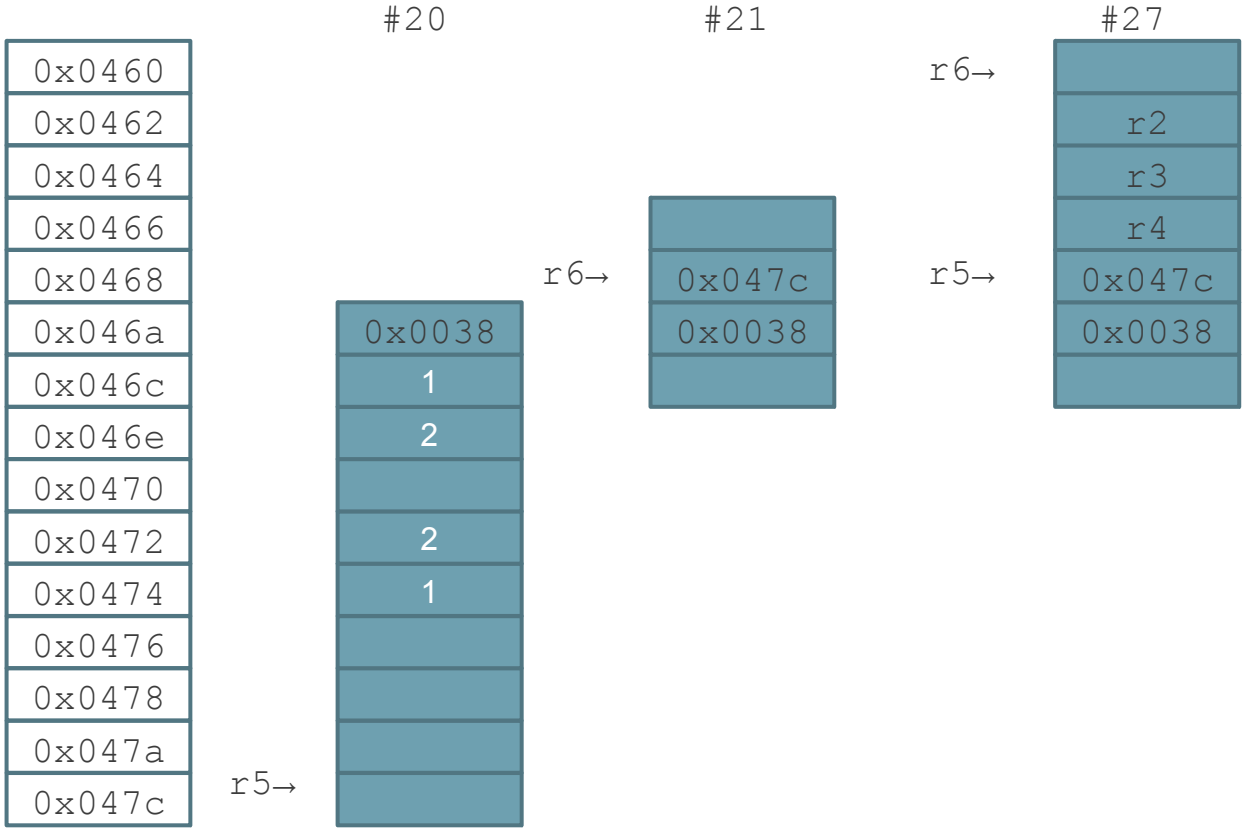
func 離開點 (func-cret)

```
 #   r0      r1     r2      r3      r4      r5      r6     r7
34:0003,0000,0000,0000,0000,0468,0460,005c jmp 0x00078      func
35:0003,0000,0000,0000,0000,0468,0460,0078 mov r5,r1        func-cret
36:0003,0468,0000,0000,0000,0468,0460,007a mov -(r1),r4     func-cret
37:0003,0466,0000,0000,0000,0468,0460,007c mov -(r1),r3     func-cret
38:0003,0464,0000,0000,0000,0468,0460,007e mov -(r1),r2     func-cret
39:0003,0462,0000,0000,0000,0468,0460,0080 mov r5,r6        func-cret
40:0003,0462,0000,0000,0000,0468,0468,0082 mov (r6)+,r5     func-cret
41:0003,0462,0000,0000,0000,047c,046a,0084 rts r7           func-cret
42:0003,0462,0000,0000,0000,047c,046c,0038 tst (r6)+        main
```



Function & Stack

- SP: Stack now r6
- R5 previous stack
- chain with a pair of r6: r5



Assembly language & C execution environment

- asm instruction sequence generated from the stack

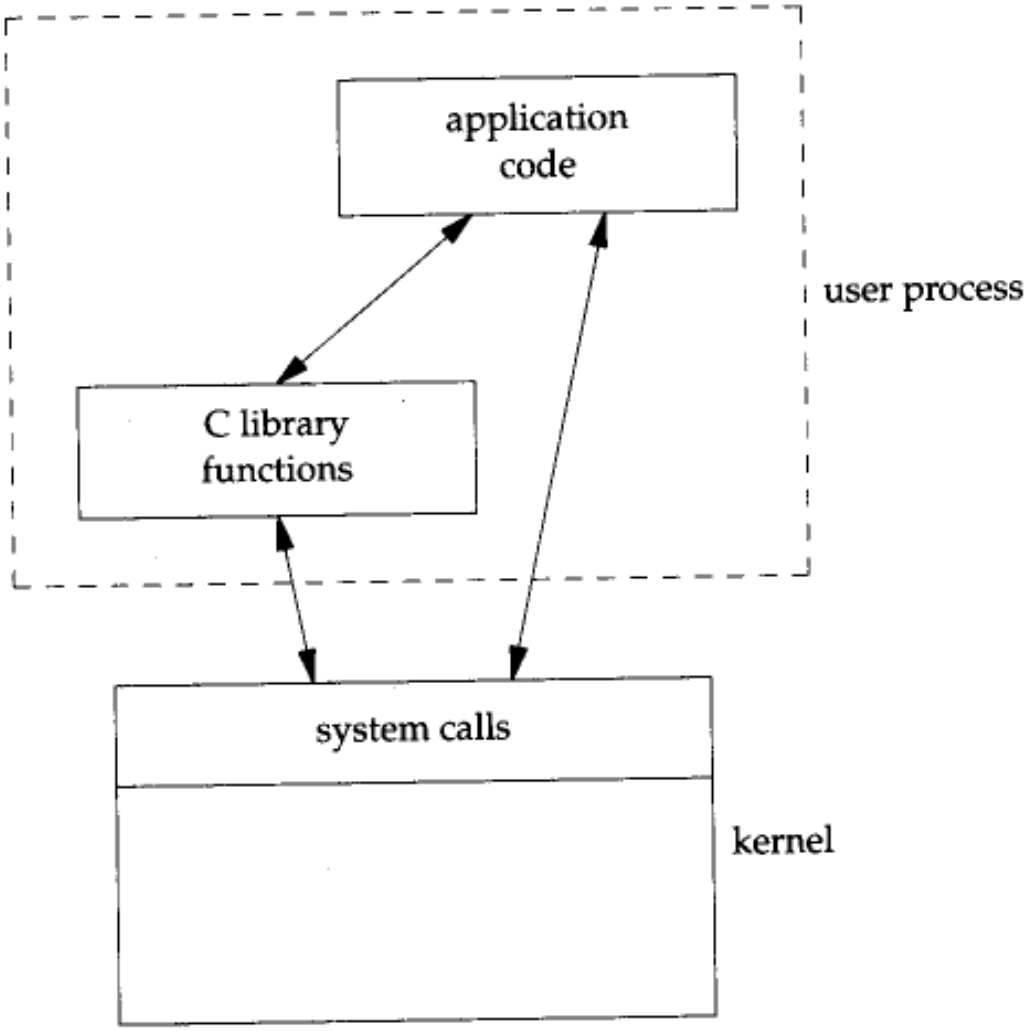
  does not require extra stack
- Instruction sequence generated from the C language,

  you need a stack at run time
  - Passing Arguments use area
  - Area of “auto” variable
  - Save area (r2, r3, r4) register the caller
  - Save area of the return address from the callee

System Call

System Call

- The kernel implements a set of special routines
- A user program invokes a routine in the kernel by issuing a hardware TRAP
- The trap switches the CPU into a privileged mode and the kerne executes the system call
- The CPU goes back to user mode
- A C language API exists for all system calls



C function call: getpid(2)

```
main()
{
    int i;
    i = getpid();
}
```

getpid(2) 對應的組合語言

```
.globl  _main
.text
    _main:
    ~~main:
    ~i=177770
    jsr     r5,csv
    tst     -(sp)
    jsr     pc,_getpid
    mov     r0,-10(r5)
L1:   jmp  cret
.globl
.data
```

getpid.s [/usr/source/s4/getpid.s]

```
getpid  = 20.

.globl  _getpid

_getpid:
        mov     r5,-(sp)
        mov     sp,r5
        sys      getpid
        mov     (sp)+,r5
        rts       pc
```

getpid.s 的執行追蹤

```
3:  001c, 0000, 0000, 0000, 0000, 0446, 043c, 001e  jsr r7, 0x00034
4:  001c, 0000, 0000, 0000, 0000, 0446, 043a, 0034  mov r5, -(r6)
5:  001c, 0000, 0000, 0000, 0000, 0446, 0438, 0036  mov r6,r5
6:  001c, 0000, 0000, 0000, 0000, 0438, 0438, 0038  sys getpid
7:  001c, 0000, 0000, 0000, 0000, 0438, 0438, 003a  mov (r6)+, r5
8:  001c, 0000, 0000, 0000, 0000, 0446, 043a, 003c  rts r7
```

C function call: open(2)

```
main()
{
    int f;
    f = open("hoge", 2);
}
```

open(2) 對應的組合語言

```
.globl  _main
.text
    _main:
    ~~main:
    ~f=177770
    jsr         r5,csv
    tst         -(sp)
    mov         $2,(sp)
    mov         $L2,-(sp)
    jsr         pc,*$_open
    tst         (sp)+
    mov         r0,-10(r5)
L1: jmp     cret
.globl
.data
L2:.byte 150,157,147,145,0
```

open.s [/usr/source/s5/open.s]

```
globl _open, cerror
_open:
    mov r5,-(sp)
    mov sp,r5
    mov 4(r5),0f
    mov 6(r5),0f+2
    sys 0; 9f
    bec 1f
    jmp cerror
1:
    mov (sp)+,r5
    rts pc
.data
9:
sys open;
0:  ..;
 ..
```

open.s 的執行追蹤

```
#   r0    r1    r2    r3    r4    r5    r6    r7
1: 001c, 0000, 0000, 0000, 0000, 0482, 0474, 0034   mov r5,-(r6)
2: 001c, 0000, 0000, 0000, 0000, 0482, 0472, 0036   mov r6,r5
3: 001c, 0000, 0000, 0000, 0000, 0472, 0472  0038   mov 4(r5), 0x008e
4: 001c, 0000, 0000, 0000, 0000, 0472, 0472, 003e   mov 6(r5), 0x0090
5: 001c, 0000, 0000, 0000, 0000, 0472, 0472, 0044   sys indir 0x0008c
6: 0003, 0000, 0000, 0000, 0000, 0472, 0472, 0048   bcc 0x0004e
7: 0003, 0000, 0000, 0000, 0000, 0472, 0472, 004e    mov (r6)+, r5
8: 0003, 0000, 0000, 0000, 0000, 0482, 0474, 0050   rts
```

System call & library call

- Library
  - Manual section 3
  - 例 fopen(3)
  - Behavior in the space of the user program
  - System calls as necessary
- System Call
  - Manual section 2
  - 例 : open(2)
  - 呼叫 OS 內部處理
  - How to call and Call / Return different from normal

Exception & Interrupt

Exceptions & Interrupts

- Trap
  - Result of unexpected internal CPU events like hardware or power failures.
  - A user mode program can explicitly use trap as part of a system call.
  - Ranked top priority.
- Interrupt
  - Controllers of peripheral devices **interrupt** CPU for some operating system service.
  - This is caused by an event external to CPU.
  - Handled by a priority based scheme.
  - Teletype, paper tape, line printer, magnetic disk, clock

Trap/Exception

Flow of exception

- Interrupt / exception
- Save the context
  - Temporarily stored on the PC and PSW internal CPU
  - Identify the factors of development, factors: search (vector) of the address table
  - Sets the PC to the vector
  - Saved in the kernel stack (push) PC had been stored temporarily, the PSW
- Execution of handler
- Restore the context
  - Instruction is executed in the state rtt PC, PSW is stored in the kernel stack
  - instructions from the kernel stack rtt restore (pop) PC, the PSW
- Execute the next instruction of an interrupt occurs

Trap

- Also called software interrupts
  - Bus errors
  - Illegal instructions
  - Segmentation exceptions
  - Floating exceptions
  - System calls
- The operating system
  - Captures the trap
  - Identifies the trap
  - If system calls, performs the requested tasks
  - Possibly sends a signal back to the user program.

Exception vectors

Vector Location	Trap type	Priority
004	Bus timeout	7
010	Illegal instruction	7
014	bpt-trace	7
020	iot	7
024	Power failure	7
030	Emulator trap	7
034	Trap instruction/ system entry	7
114	11/70 parity	7
240	Programmed interrupt	7
244	Floating point error	7
250	Segmentation violation	7

Hint: obtain Lions' source listing: `http://www.tom-yam.or.jp/2238/src/`

Line #500 [low.s]

```
0500 / low core
0501
0505   br7 = 340
0506
0507   . = 0^.
0508   br 1f
0509   4
0510
0511   / trap vectors
0512   trap; br7+0. / bus error
0513   trap; br7+1. / illegal instruction
0514   trap; br7+2. / bpt-trace trap
0515   trap; br7+3. / iot trap
0516   trap; br7+4. / power fail
0517   trap; br7+5. / emulator trap
0518   trap; br7+6. / system entry
```

Line #752 [m40.s]

```
0752    .globl trap, call
0753    /* -------------- */
0754    .globl _trap
0755    trap:
0756    mov PS,-4(sp)
0757    tst nofault
0758    bne 1f
0759    mov SSR0,ssr
0760    mov SSR2,ssr+4
0761    mov $1,SSR0
0762    jsr r0,call1; _trap
0763    / no return
```

Line #2693 [trap.c]

```
2693 trap(dev, sp, r1, nps, r0, pc, ps)
2694 {
2695    register i, a;
2696    register struct sysent *callp;
2697
2698    savfp();
2699    if ((ps&UMODE) == UMODE)
2700        dev =| USER; 2750
2702    switch(dev) {
   :
2715    default:
2716        printf("ka6 = %o¥n", *ka6);
2717        printf("aps = %o¥n", &ps);
2718        printf("trap type %o¥n", dev);
2719        panic("trap");
2721    case 0+USER: /* bus error *
2722        i = SIGBUS;
2723        break;
```

Line #2751 [trap.c]

```
2751   case 6+USER: /* sys call */
2752       u.u_error = 0;
2753       ps =& ~EBIT;
2754       callp = &sysent[fuiword(pc-2)&077];
```

- PC is the address following the instruction fell into the "trap"
- Pull the two 2-byte instruction
- Reading (2 bytes) word from user space
- Retrieve the lower 6 bits
- Looking at the index sysent then this value

Line #2906 [sysent.c]

```
2906 * to the appropriate routine for processing a system call.
2907 * Each row contains the number of arguments
2908 * and a pointer to the routine.
2909 */
2910 int sysent[]
2911 {
2912         0, &nullsys, /* 0 = indir */
2913         0, &rexit,       /* 1 = exit */
2914         0, &fork,        /* 2 = fork */
2915         2, &read,        /* 3 = read */
2916         2, &write,   /* 4 = write */
2917         2, &open,    /* 5 = open */
```

System call 號碼

- `indir = 0.`
- `exit = 1.`
- `fork = 2.`
- `read = 3.`
- `write = 4.`
- `open = 5.`
- `close = 6.`
- `wait = 7.`
- `creat = 8.`
- `...`

System call 處理方式

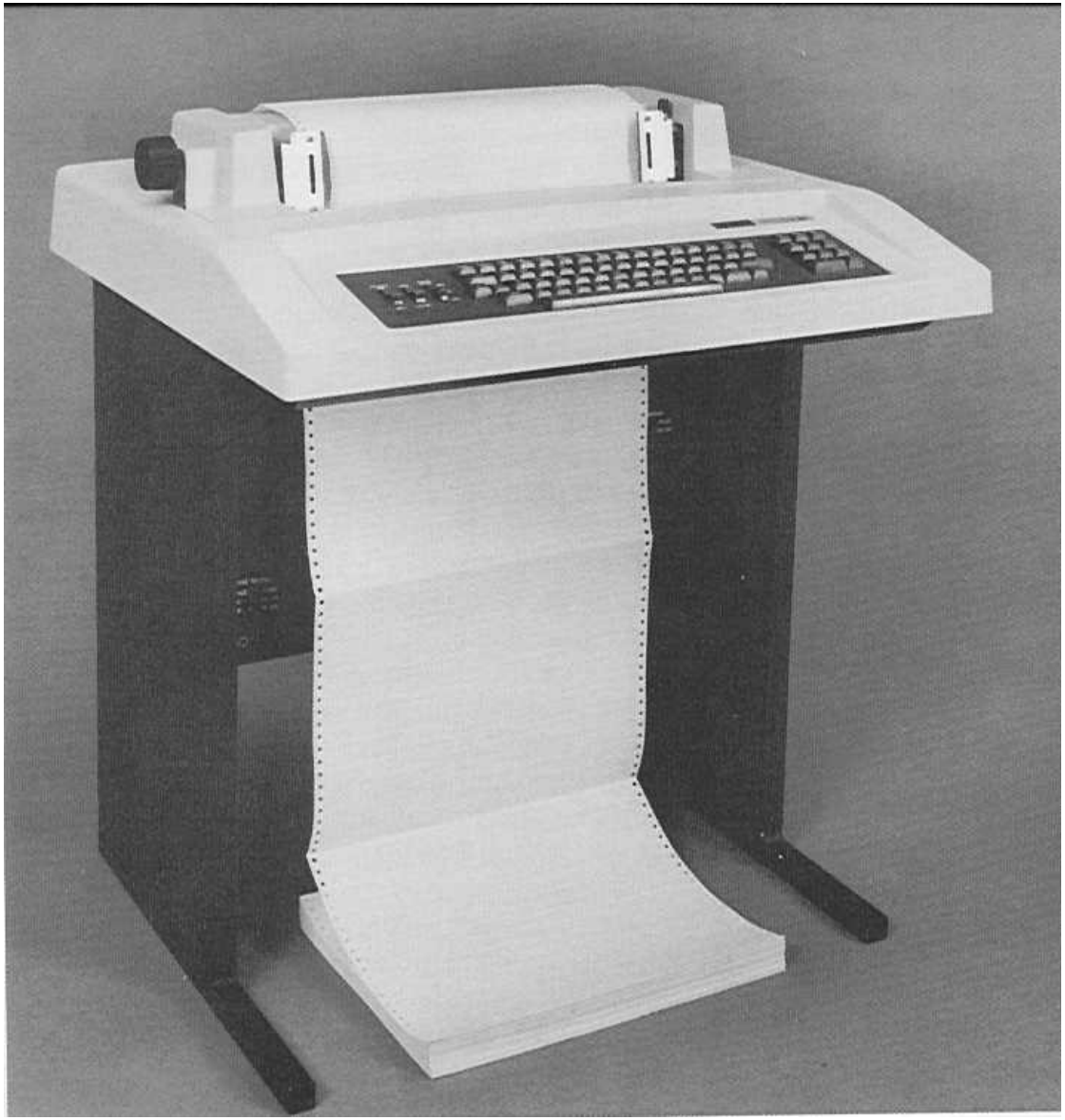
- C 語言敘述 open(“aa”, 2)
- 透過 libc 的 sys 5 來執行
  - sys 為 emulator trap instruction
- 切換到 kernel space ，並取得系統呼叫號碼 5
- Locate and open process in the kernel sysent in [5]
- 呼叫核心的 open 操作

Interrupts

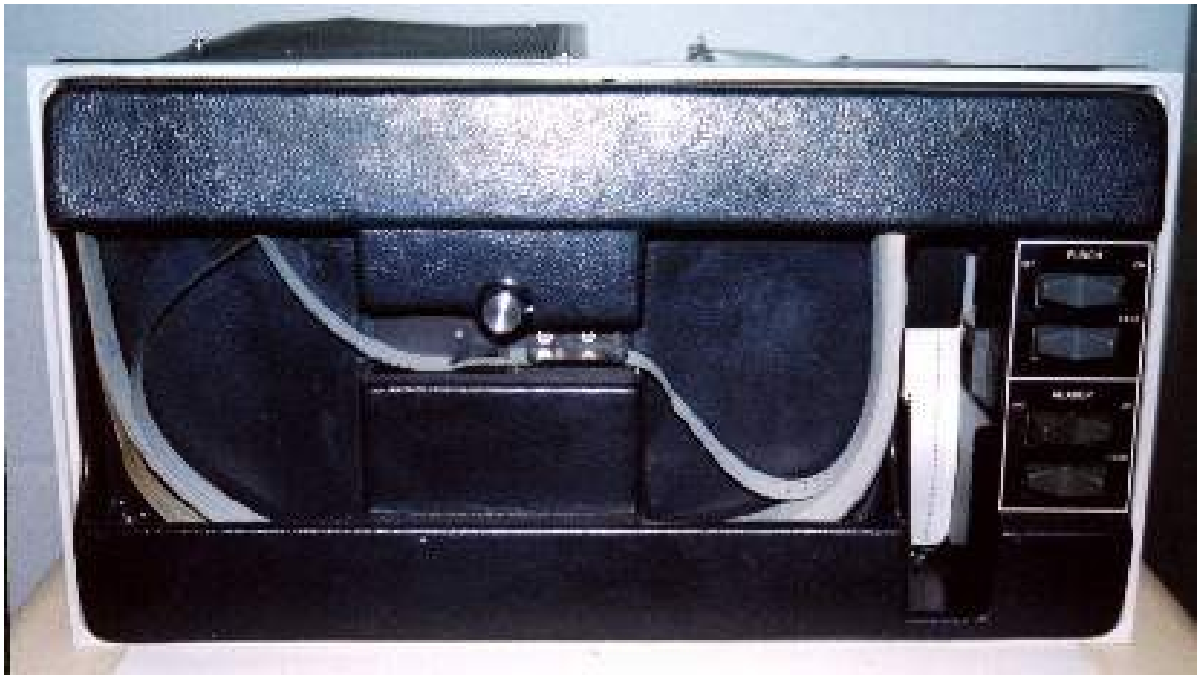
Interrupt vector

Vector Location	device	priority
060	Teletype input	4
064	Teletype output	4
070	Paper tape input	4
074	Paper tape output	4
100	Line clock	6
104	Programmable clock	6
200	Line printer	4
220	RK disk driver	5

Teletype?



Paper type?




DIGITAL EQUIPMENT CORPORATION - PROGRAMMED DATA PROCESSOR
PDP

Clock: line and programmable

- Line Clock
  - 電源藉由週期震盪取得
  - AC > 降壓變壓 > 整流 > 電容
  - Allowed to retrieve the pulse frequency of power
  - 20ms interval is 50HZ
  - old digital clock generates a second electrical pulse in a HZ
- Programmable clock
  - Pulses at a specified interval
- PDP-11 需要上述其一方可運作

Line #525 [low.s]

```
0525: . = 60^.
0526:           klin; br4
0527:           klou; br4
0528:
0529: . = 70^.
0530:           pcin; br4
0531:           pcou; br4
0532:
0533: . = 100^.
0534:           kwlp; br6
0535:           kwlp; br6
0539:
0540: . = 200^.
0541:           lpou; br4
0542:
0543: . = 220^.
0544:           rkio; br5
```

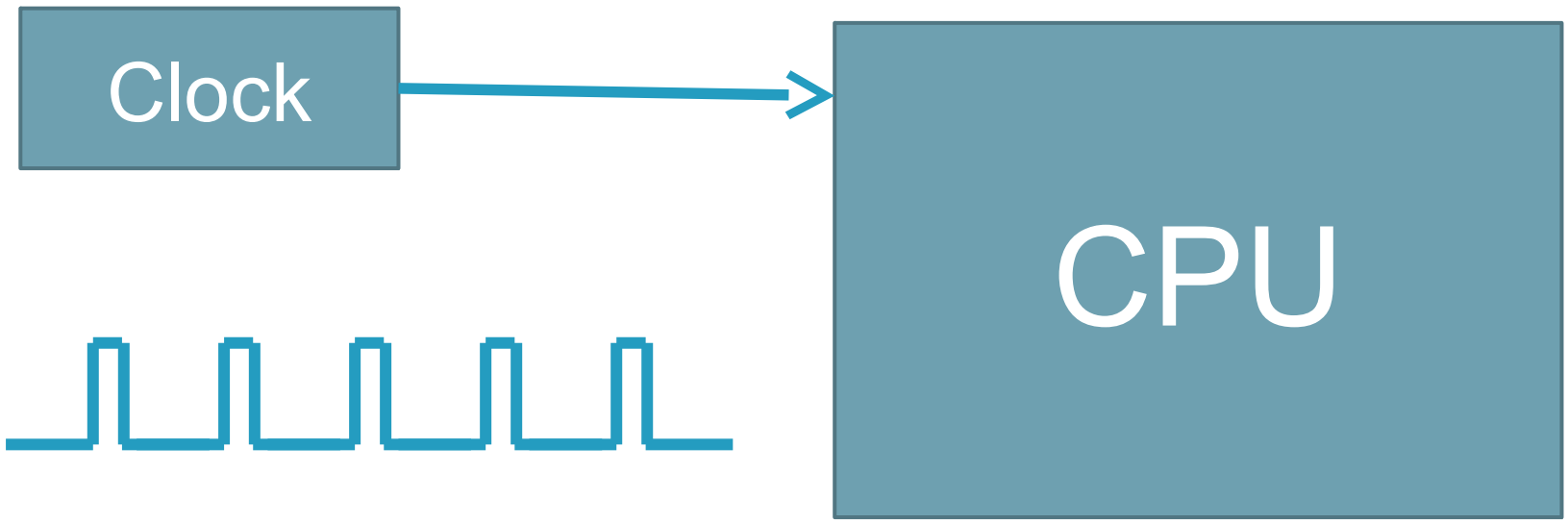
Vector	device	entry
060	Teletype input	klin
064	Teletype output	klou
070	Paper tape input	pcin
074	Paper tape output	pcou
100	Line clock	kwlp
104	Programmable clock	kwlp
200	Line printer	lpou
220	RK disk driver	rkio

RK Disk

- RK disk storage consists of
  - disk controller RK11-D
  - a number of RK disk drives, up to eight for each disk controller
  - Removable disk cartridges
- Most used in PDP11 systems

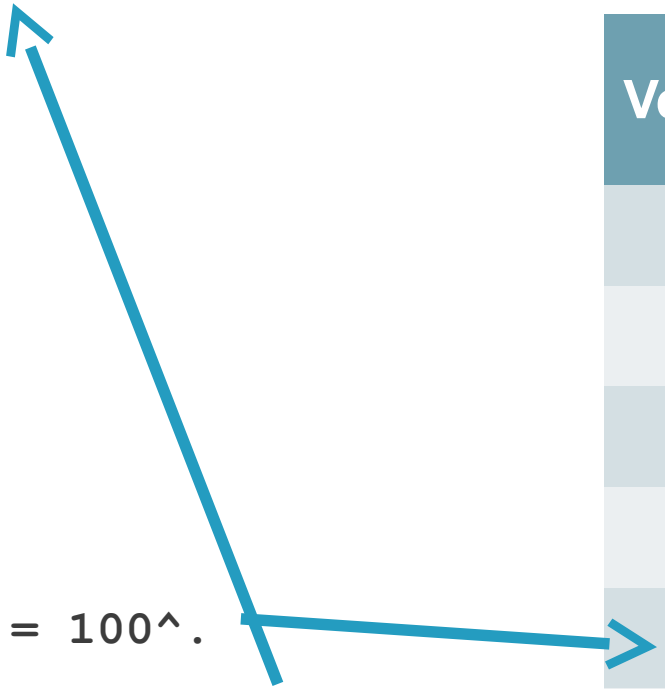
Line Clock

- 50 Pulses per second enters the CPU from the external clock
- Perform the processing of the interrupt vector to a pulse each



Line #568 [low.s]

```
0568:
0569: .globl _clock
0570: kwlp: jsr r0,call; _clock
```



```
0533: . = 100^.
0534:          kwlp; br6
0535:          kwlp; br6
```

Vector	device	entry
060	Teletype input	klin
064	Teletype output	klou
070	Paper tape input	pcin
074	Paper tape output	pcou
100	Line clock	kwlp
104	Programmable clock	kwlp
200	Line printer	lpou
220	RK disk driver	rkio

Line #3725 [clock.c]

```
3725: clock(dev, sp, r1, nps, r0, pc, ps)
3726: {
3727:         register struct callo *p1, *p2;
3728:         register struct proc *pp;
3729:
3730:         /*
3731:          * restart clock
3732:          */
3733:
3734:         *lks = 0115;
3735:
3736:         /*
3737:          * display register
3738:          */
3739:
3740:         display();
```

Line #3743 [clock.c]

Callout

```
3743:          * callouts
3744:          * if none, just return
3745:          * else update first non-zero time
3746:          */
3747:
3748:         if(callout[0].c_func == 0)
3749:                 goto out;
3750:         p2 = &callout[0];
3751:         while(p2->c_time<=0 && p2->c_func!=0)
3752:                 p2++;
3753:         p2->c_time--;
3754:
3755:         /*
3756:          * if ps is high, just return
3757:          */
3758:
3759:         if((ps&0340) != 0)
3760:                 goto out;
```

Line #3763 [clock.c]

```
3763:           * callout
3764:           */
3765:
3766:          spl5();
3767:          if(callout[0].c_time <= 0) {
3768:                  p1 = &callout[0];
3769:                  while(p1->c_func != 0 && p1->c_time <= 0) {
3770:                          (*p1->c_func)(p1->c_arg);
3771:                          p1++;
3772:                  }
3773:                  p2 = &callout[0];
3774:                  while(p2->c_func = p1->c_func) {
3775:                          p2->c_time = p1->c_time;
3776:                          p2->c_arg = p1->c_arg;
3777:                          p1++;
3778:                          p2++;
3779:                  }
3780:          }
```

Callout

Line #3782 [clock.c]

```
3782:          /*
3783:           * lightning bolt time-out
3784:           * and time of day
3785:           */
3786:
3787: out:
3788:          if((ps&UMODE) == UMODE) {
3789:                  u.u_utime++;
3790:                  if(u.u_prof[3])
3791:                          incupc(pc, u.u_prof);
3792:          } else
3793:                  u.u_stime++;
3794:          pp = u.u_procp;
3795:          if(++pp->p_cpu == 0)
3796:                  pp->p_cpu--;
```

Acct

Line #3797 [clock.c]

```
3797:           if(++lbolt >= HZ) {
3798:                   if((ps&0340) != 0)
3799:                           return;
3800:                   lbolt =- HZ;
3801:                   if(++time[1] == 0)
3802:                           ++time[0];
3803:                   spl1();
3804:                   if(time[1]==tout[1] && time[0]==tout[0])
3805:                           wakeup(tout);
3806:                   if((time[1]&03) == 0) {
3807:                           runrun++;
3808:                           wakeup(&lbolt);
3809:                   }
3810:                   for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
3811:                   if (pp->p_stat) {
3812:                           if(pp->p_time != 127)
3813:                                   pp->p_time++;
```

sched 的進入點

Line #3814 [clock.c]

```
3814:                           if((pp->p_cpu & 0377) > SCHMAG)
3815:                                   pp->p_cpu =- SCHMAG; else
3816:                                   pp->p_cpu = 0;
3817:                           if(pp->p_pri > PUSER)
3818:                                   setpri(pp);
3819:                   }
3820:                   if(runin!=0) {
3821:                           runin = 0;
3822:                           wakeup(&runin);
3823:                   }
3824:                   if((ps&UMODE) == UMODE) {
3825:                           u.u_ar0 = &r0;
3826:                           if(issig())
3827:                                   psig();
3828:                           setpri(u.u_procp);
3829:                   }
3830:           }
3831: }
```

謎樣的變數

- `u.u_time`
- `u.u_stime`
- `pp->p_time`
- `lbolt`
- `time[2]`
- `tout[2]`

callout 結構

```
struct callo
{
    int c_time;         /* ticks between events */
    int c_arg;          /* function argument */
    int (*c_func)();    /* function pointer */
} callout[NCALL];
```

The “callout” array maintains a list of functions to be executed after “c_time” passes.

clock callout (1/2)

```
3748:           if(callout[0].c_func == 0)
3749:                   goto out;
3750:           p2 = &callout[0];
```

If the first callout registered, unregistered and also to determine the subsequent callout, Callout skipped

```
3751:           while(p2->c_time<=0 && p2->c_func!=0)
3752:                   p2++;
3753:           p2->c_time--;
3758:
3759:           if((ps&0340) != 0)
3760:                   goto out;
3761:
```

Locate the first c_time Callout is greater than 0, c_time--
Previous callout will be processed after this from here

clock callout (2/2)

```
3766:           spl5();
3767:           if(callout[0].c_time <= 0) {
3768:                   p1 = &callout[0];
3769:                   while(p1->c_func != 0 && p1->c_time <= 0) {
3770:                           (*p1->c_func)(p1->c_arg);
3771:                           p1++;
3772:                   }
3773:                   p2 = &callout[0];
3774:                   while(p2->c_func = p1->c_func) {
3775:                           p2->c_time = p1->c_time;
3776:                           p2->c_arg = p1->c_arg;
3777:                           p1++;
3778:                           p2++;
3779:                   }
3780:           }
```

Callout registration time has passed to the top

一旦有效則個別處理

Pack array Callout

clock acct

```
3787: out:
3788:           if((ps&UMODE) == UMODE) {
3789:                   u.u_utime++;
3790:                   if(u.u_prof[3])
3791:                           incupc(pc, u.u_prof);
3792:           } else
3793:                   u.u_stime++;


3794:           pp = u.u_procp;
3795:           if(++pp->p_cpu == 0)
3796:                   pp->p_cpu--;
```

If the previous mode was USER mode, increment the user time.

If the previous mode was KERNEL mode, increment the system time.

Add the p_cpu of Proc. Adjusted to avoid overflow

pp is a pointer structure proc, u is the structure user.
One by one to a process management structure of both process

clock sched 的進入點 (1/2)

```
3797:           if(++lbolt >= HZ) {
3798:                   if((ps&0340) != 0)
3799:                           return;
3800:                   lbolt =- HZ;
3801:                   if(++time[1] == 0)
3802:                           ++time[0];
3803:                   spl1();
3804:                   if(time[1]==tout[1] && time[0]==tout[0])
3805:                           wakeup(tout);
3806:                   if((time[1]&03) == 0) {
3807:                           runrun++;
3808:                           wakeup(&lbolt);
3809:                   }
```

Continue if lbolt has reached 60. (也就是經過 1 秒週期)

重設 lbolt

Process causes sleeping

Cause the processes that are waiting in lbolt every 4 seconds

Flow without hesitation this time

clock sched 的進入點 (2/2)

```
3810:                   for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
3811:                   if (pp->p_stat) {
3812:                           if(pp->p_time != 127)
3813:                                   pp->p_time++;
3814:                           if((pp->p_cpu & 0377) > SCHMAG)
3815:                                   pp->p_cpu =- SCHMAG; else
3816:                                   pp->p_cpu = 0;
3817:                           if(pp->p_pri > PUSER)
3818:                                   setpri(pp);
3819:                   }
3820:                   if(runin!=0) {
3821:                           runin = 0;
3822:                           wakeup(&runin);
3823:                   }
3824:                   if((ps&UMODE) == UMODE) {
3825:                           u.u_ar0 = &r0;
3826:                           if(issig())
3827:                                   psig();
3828:                           setpri(u.u_procp);
3829:                   }
3830:           }
```

Scan the array of management structure of the process

When runin is true, wakeup a process that is waiting in runin

謎樣的變數之解釋

- `u.u_time:` USER 時間
- `u.u_stime:` SYSTEM 時間
- `pp->p_time :` 啟動 Process 之後經歷的時間
- `lbolt:` 每秒遞增 60 次 (Milliseconds)
- `time[2]:` 自 1970 年以來的秒數
- `tout[2]:` time of next sleep(2)

UNIX 的 PPDA (PerProcDataArea)

- User 結構體： user.h

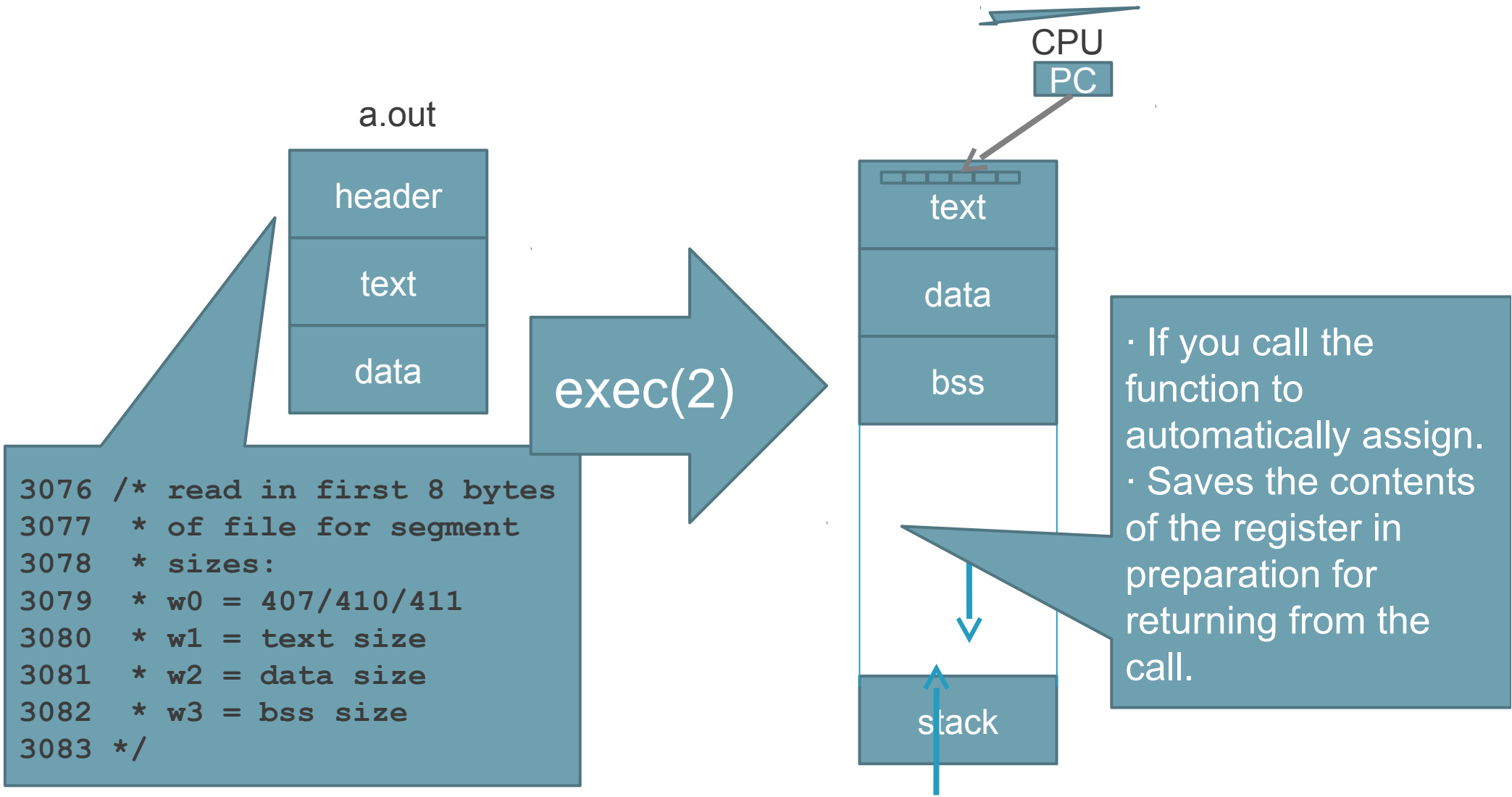
  The user structure. One allocated per process. Contains all per process data that doesn't need to be referenced while the process is swapped.

- Proc 結構體： proc.h

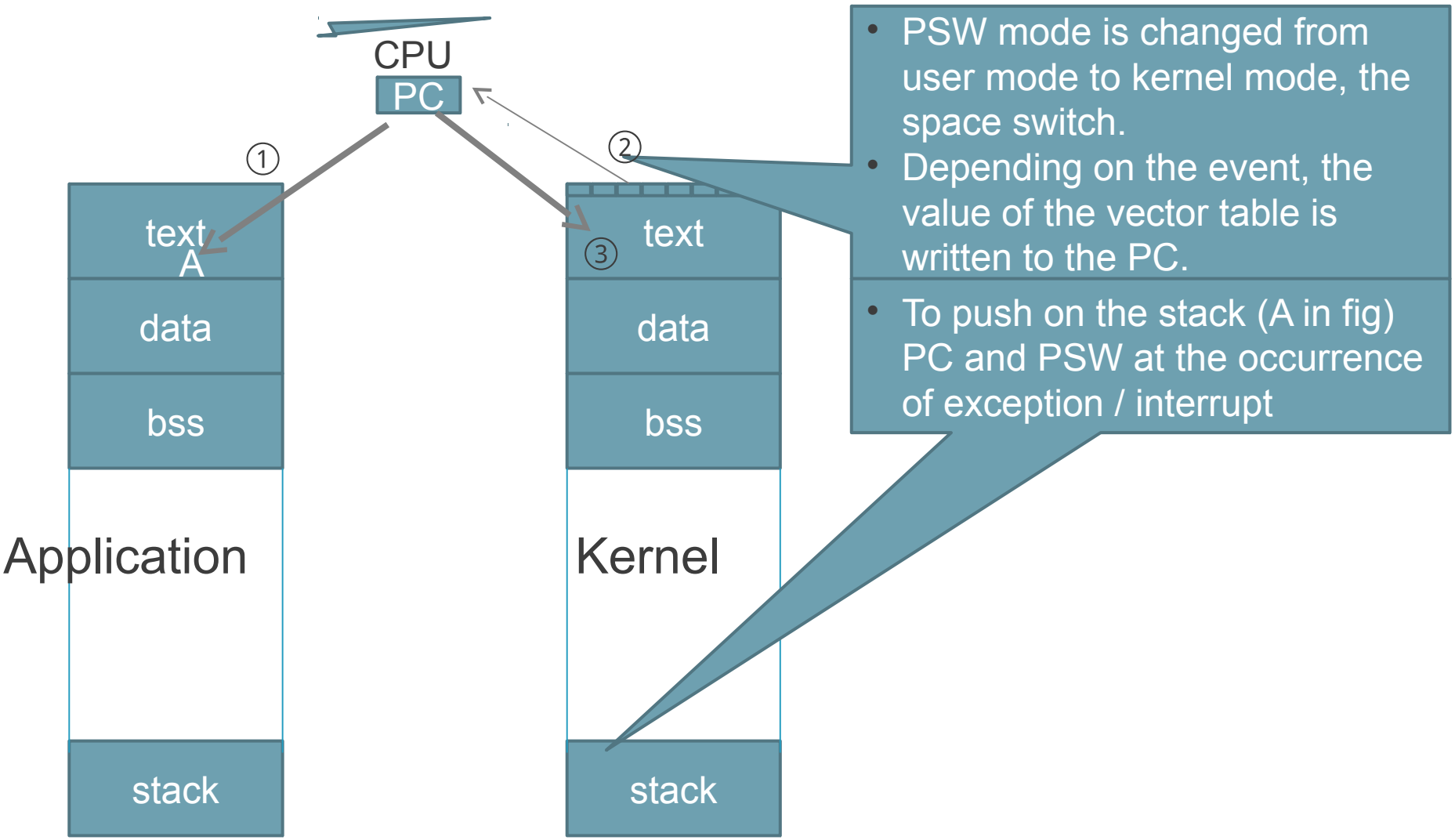
  One structure allocated per active process. It contains all data needed about the process while the process may be swapped out. Other per process data (user.h) is swapped with the process.

定址模式

a.out 的定址模式


a.out
header
text
data
exec(2)
CPU
PC
text
data
bss
stack
· If you call the function to automatically assign.
· Saves the contents of the register in preparation for returning from the call.
3076 /* read in first 8 bytes
3077 * of file for segment
3078 * sizes:
3079 * w0 = 407/410/411
3080 * w1 = text size
3081 * w2 = data size
3082 * w3 = bss size
3083 */

一旦 Exception/Interrupt 觸發



Applications and Kernel.
Two spaces?

Virtual address & physical address

- Virtual memory of the current general
  - Physical address > Virtual address
  - Amount of physical memory > amount of virtual memory
  - Restored when out of memory when the real need is saved to secondary storage in units (pages) fixed length

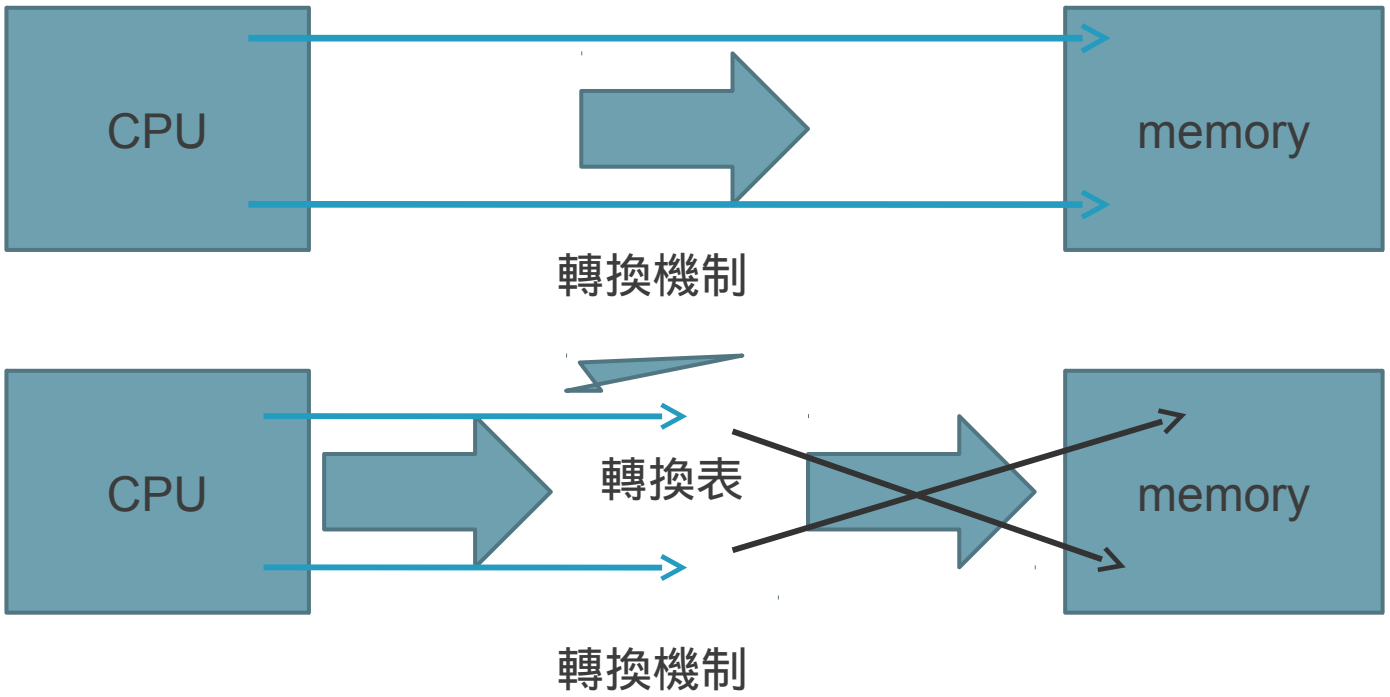
→Paging

- PDP-11 virtual memory
  - Virtual address < physical address
  - amount of virtual memory < amount of physical memory
  - Out of memory when the real deterrent is saved to secondary storage to run the entire process, to be run to restore the whole after a period of time

→Swapping

virtual memory 機制

- Hardware support is required
- Translation mechanism of the virtual address ↔ Physical address



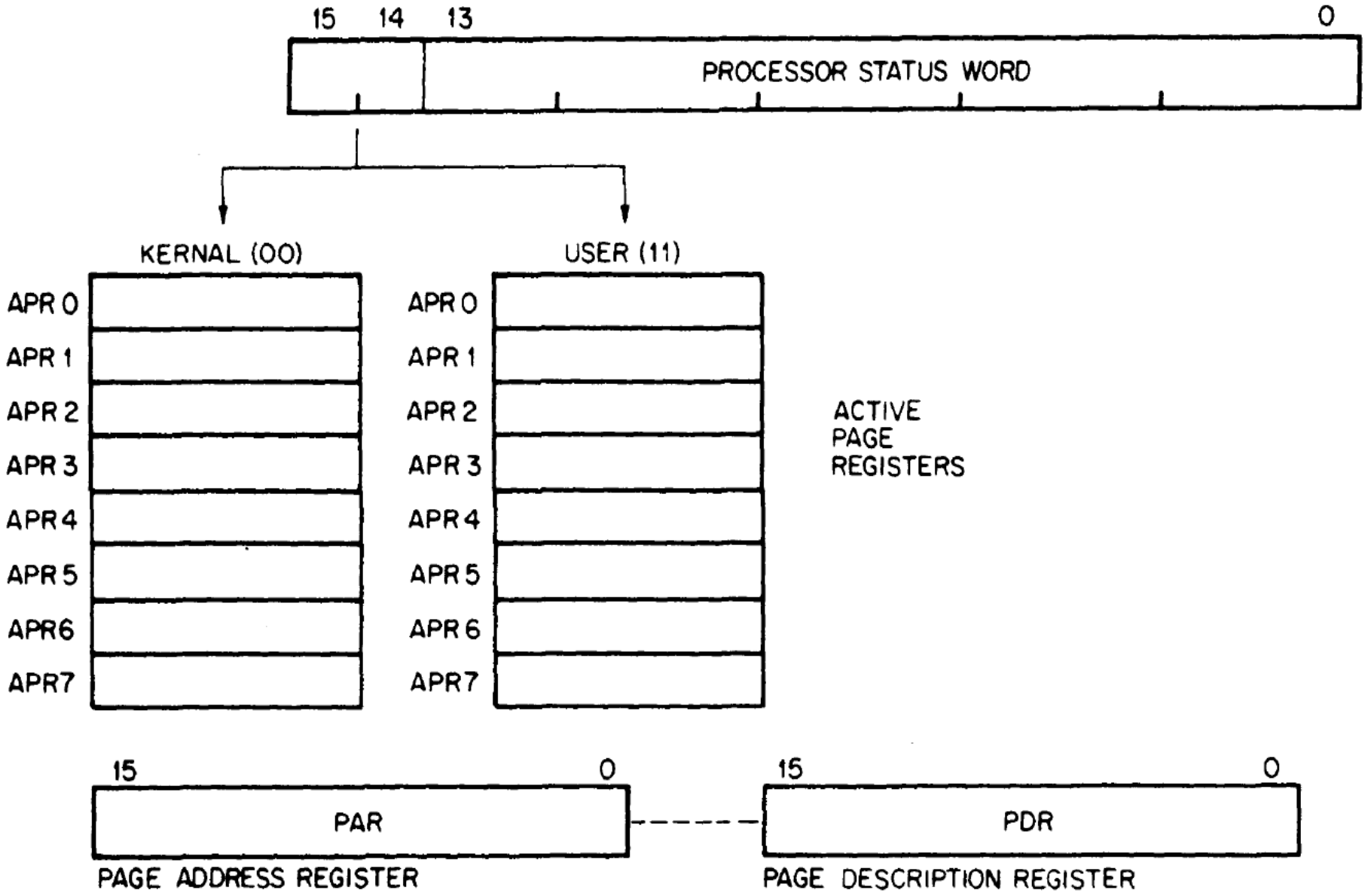
Control of address translation

- After reset, the translation mechanism operates in OFF
- In the program during the initialization of the OS
  - Initialize the translation table
  - Enable the translation mechanism
- At the time of acquisition and memory formation of a new process
  - OS modifies translation table
- When the switching process
  - Swap the translation table

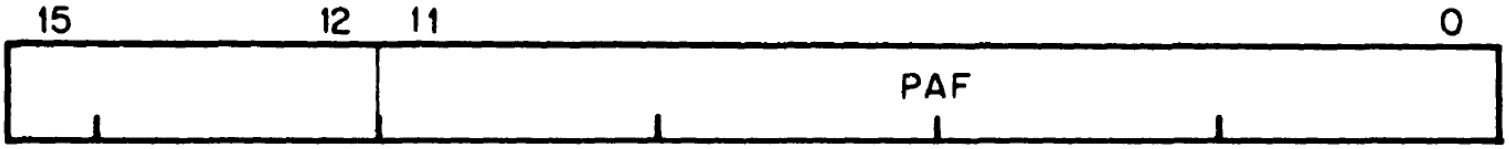
Memory management of PDP-11/40

- 8 個 Kernel-mode page
- 8 個 user-mode page
- Variable length of up to one page 8Kbyte
- Switching mode is coupled to the mode of PSW

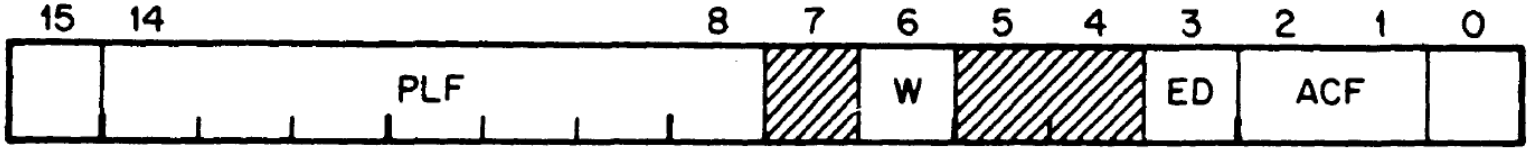
Active Page Registers (APR)



Pair of APR-PAR and PDR

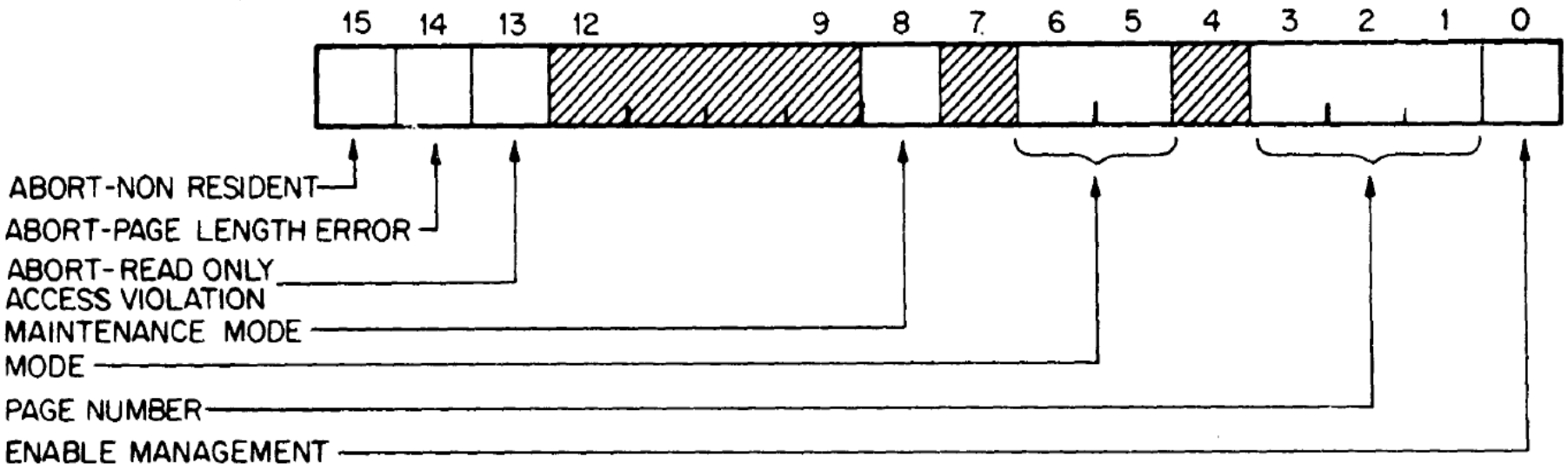


PAR specify the base address



PDR attribute specifies the memory

Enable virtual memory



valid for 1 bit0: enable management

重要位址定義 [m40.s]

- USIZE = 16 Size of User Block (*64 = 1024 B)
- PS = 177776 Program Status Word
- SSR0 = 177572 Status Register
- KISA0 = 172340 Kernel Segment Address Register #0
- KISA6 = 172354 Kernel Segment Address Register #6
- KISD0 = 172300 Kernel Segment Descriptor Register #0
- UISA0 = 177640 User Segment Address Register #0
- UISA1 = 177642 User Segment Address Register #1
- UISD0 = 177600 User Segment Descriptor Register #0
- UISD1 = 177602 User Segment Descriptor Register #1
- IO = 7600 I/O Segment Register

start: configure virtual space [m40.s]

```
0612: start:
0613:         bit     $1,SSR0
0614:         bne     start                   / loop if restart
0615:         reset
0616:
0617: / initialize systems segments
0618:
0619:         mov     $KISA0,r0
0620:         mov     $KISD0,r1
0621:         mov     $200,r4
0622:         clr     r2
0623:         mov     $6,r3
0624: 1:
0625:         mov     r2,(r0)+
0626:         mov     $77406,(r1)+            / 4k rw
0627:         add     r4,r2
0628:         sob     r3,1b
```

start: initialize userspace [m40.s]

```
0630: / initialize user segment
0631:
0632:           mov     $_end+63.,r2
0633:           ash     $-6,r2
0634:           bic     $!1777,r2
0635:           mov     r2,(r0)+                        / ksr6 = sysu
0636:           mov     $usize-1\<8|6,(r1)+
0637:
0638: / initialize io segment
0639: / set up counts on supervisor segments
0640:
0641:           mov     $IO,(r0)+
0642:           mov     $77406,(r1)+                    / rw 4k
0643:
0644: / get a sp and start segmentation
0645:
0646:           mov     $_u+[usize*64.],sp
0647:           inc     SSR0
```

start: clear bss and u [m40.s]

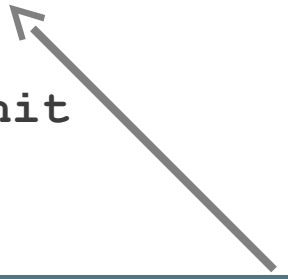
```
0646:           mov     $_u+[usize*64.],sp
0647:           inc     SSR0
0648:
0649: / clear bss
0650:
0651:           mov     $_edata,r0
0652: 1:
0653:           clr     (r0)+
0654:           cmp     r0,$_end
0655:           blo     1b
0656:
0657: / clear user block
0658:
0659:           mov     $_u,r0
0660: 1:
0661:           clr     (r0)+
0662:           cmp     r0,$_u+[usize*64.]
0663:           blo     1b
```

start: 呼叫 main 函式 [m40.s]

```
0665: / set up previous mode and call main
0666: / on return, enter user mode at 0R
0667:
0668:           mov     $30000,PS
0669:           jsr     pc,_main
0670:           mov     $170000,-(sp)
0671:           clr     -(sp)
0672:           rtt
```

建立首個 Process [main.c]

```
1627:           if(newproc()) {
1628:                   expand(USIZE+1);
1629:                   estabur(0, 1, 0, 0);
1630:                   copyout(icode, 0, sizeof icode);
1631:                   /*
1632:                    * Return goes to loc. 0 of user init
1633:                    * code just copied out.
1634:                    */
1635:                   return;
1636:           }
1637:           sched();
```



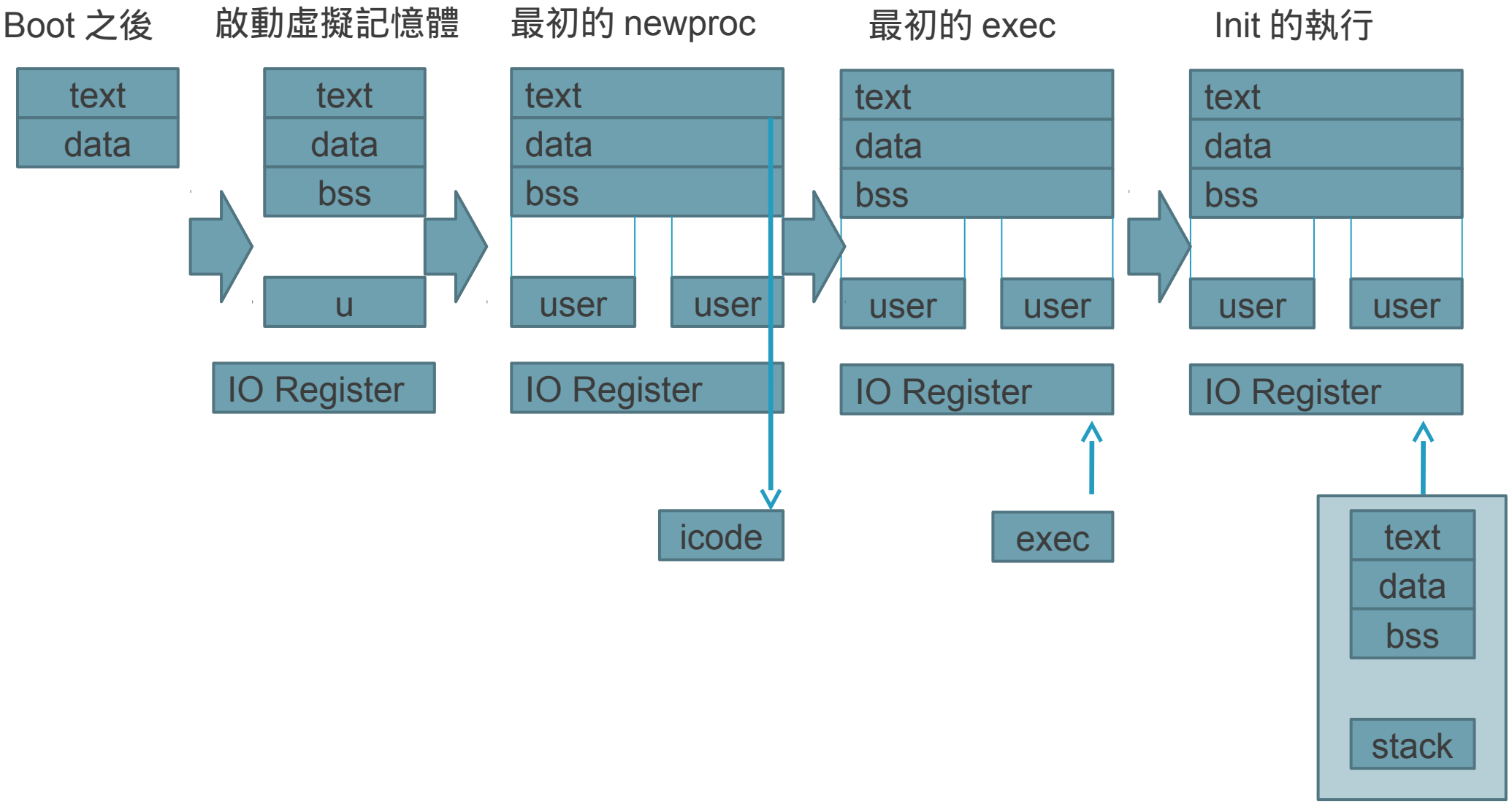
icode is th bootstrap program executed in user mode to bring up the system.

icode [main.c]

```
1511: /*
1512:  * Icode is the octal bootstrap
1513:  * program executed in user mode
1514:  * to bring up the system.
1515:  */
1516: int     icode[]
1517: {
1518:         0104413,        /* sys exec; init; initp */
1519:         0000014,
1520:         0000010,
1521:         0000777,        /* br . */
1522:         0000014,        /* initp: init; 0 */
1523:         0000000,
1524:         0062457,        /* init: </etc/init\0> */
1525:         0061564,
1526:         0064457,
1527:         0064556,
1528:         0000164,
1529: };
```

```
char* init = "/etc/init";
main( ) {
    execl( init, init, 0 );
    while( 1 ) ;
}
```

Boot space initialization



UNIX 的定址模式

- Address area of application and kernel separation
  - Little change in full virtual memory support later
  - Separate space for each application
- The kernel space
  - Text, Data, Bss share, except for some
  - Stack is a separate application for each
  - Stack the second half of the page struct user

Saving and restoring of context

PPDA - Per Processor Data Area

- Focus on the first structure of OS
- Space allocated to each unit of execution of processes, etc.
- UNIX user and proc

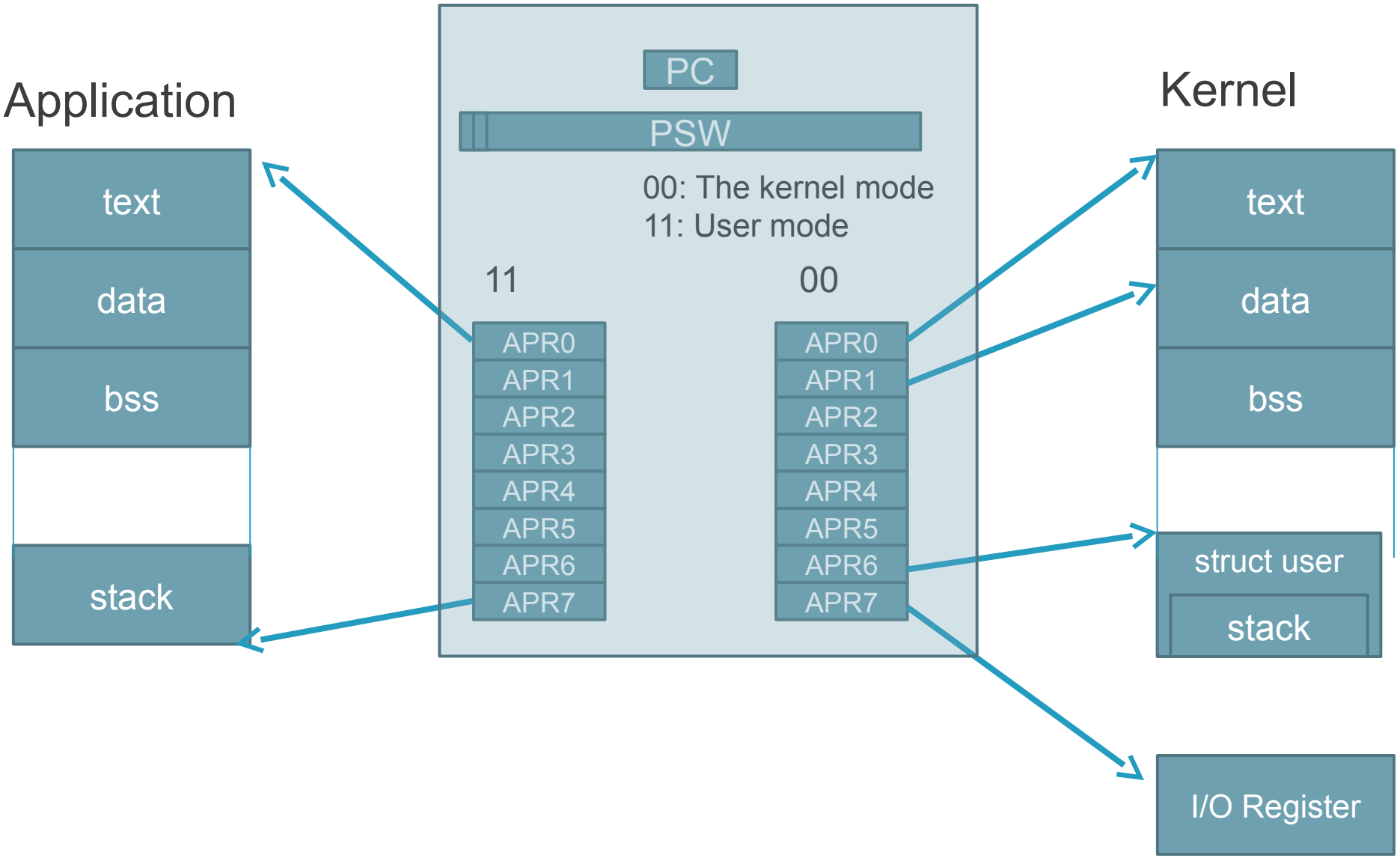
struct user [user.h]

```
0413 struct user
0414 {
0415     int u_rsav[2];      /* save r5,r6 when exchanging stacks */
0416     int u_fsav[25];     /* save fp registers */
0417     /* rsav and fsav must be first in structure */
0418     char u_segflg;      /* flag for IO; user or kernel space */
0419     char u_error;       /* return error code */
0420     char u_uid;         /* effective user id */
0421     char u_gid;         /* effective group id */
0422     char u_ruid;        /* real user id */
0423     char u_rgid;        /* real group id */
0424     int u_procp;        /* pointer to proc structure */
0436     int u_uisa[16]; /* prototype of segmentation addresses */
0437     int u_uisd[16]; /* prototype of segmentation descriptors */
```

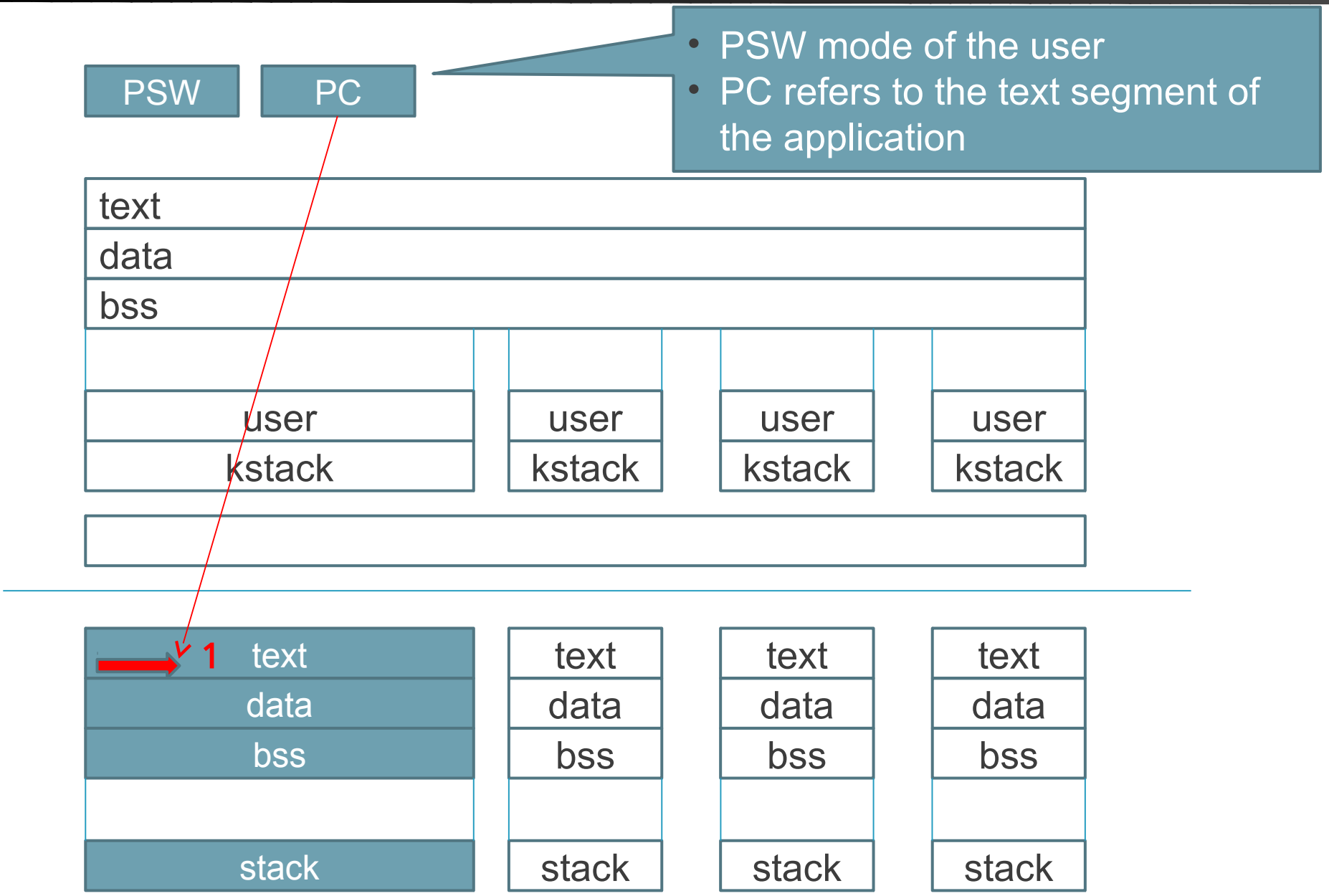
struct proc [proc.h]

```
0358 struct proc
0359 {
0360     char p_stat;
0361     char p_flag;
0362     char p_pri;         /* priority, negative is high */
0363     char p_sig;         /* signal number sent to this process */
0364     char p_uid;         /* user id, used to direct tty signals */
0365     char p_time;        /* resident time for scheduling */
0366     char p_cpu;         /* cpu usage for scheduling */
0367     char p_nice;        /* nice for scheduling */
0368     int p_ttyp;         /* controlling tty */
0369     int p_pid;          /* unique process id */
0370     int p_ppid;         /* process id of parent */
0371     int p_addr;         /* address of swappable image */
0372     int p_size;         /* size of swappable image (*64 bytes) */
0373     int p_wchan;        /* event process is awaiting */
0374     int *p_textp;       /* pointer to text structure */
0376 } proc[NPROC];
```

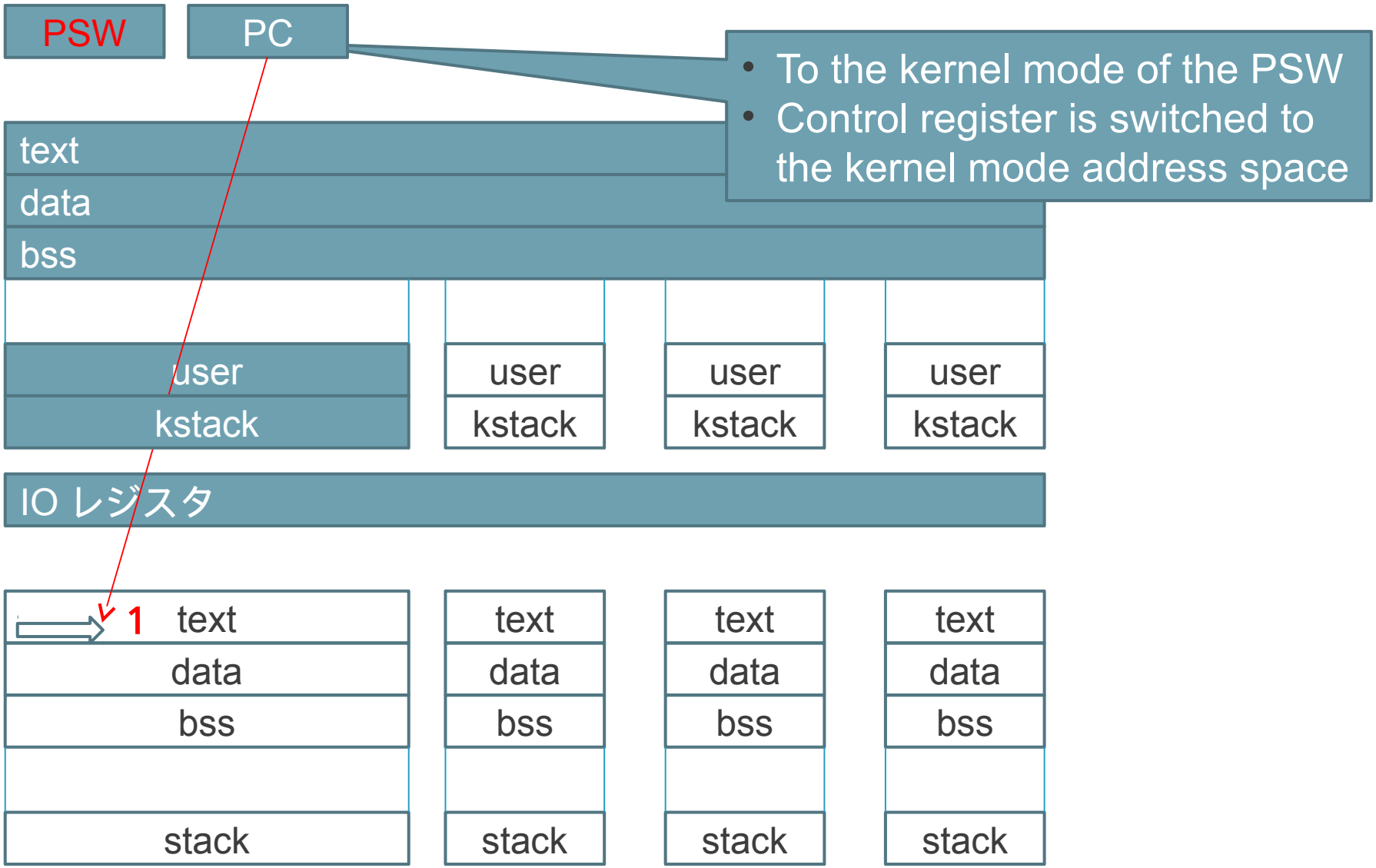
Switching mechanism



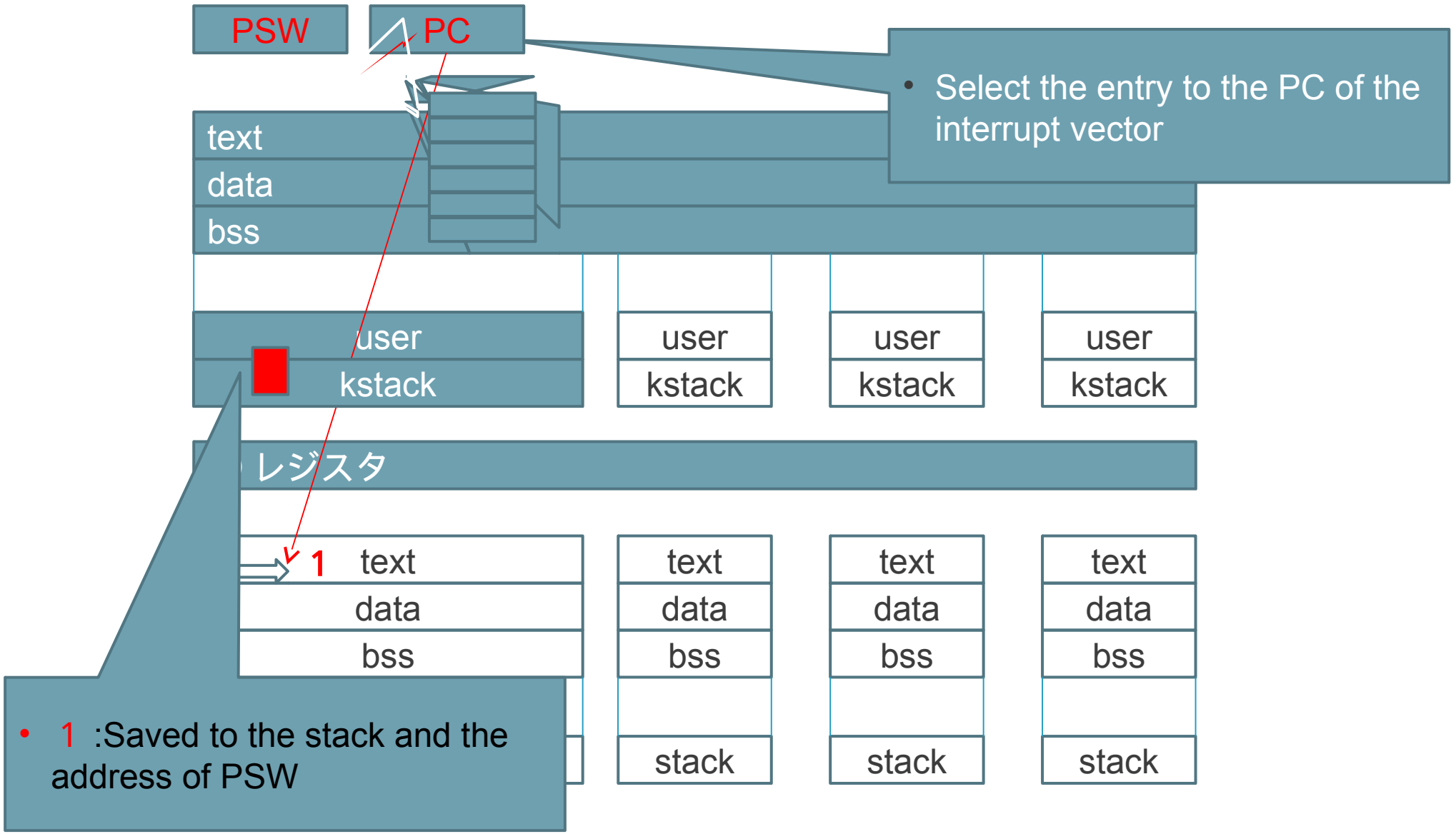
Application 運作時



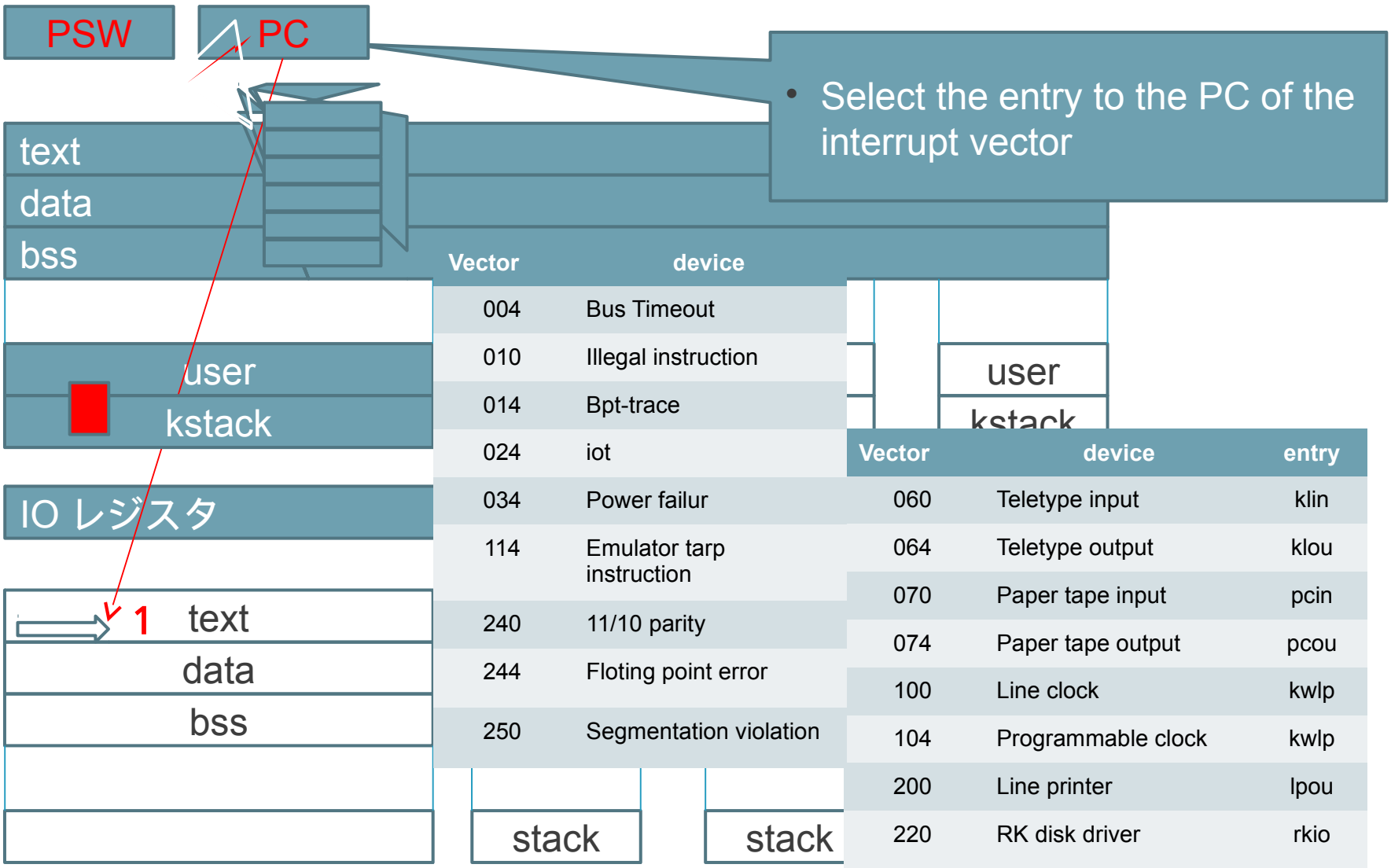
switch: interrupt 觸發時



interrupt vector



interrupt vector

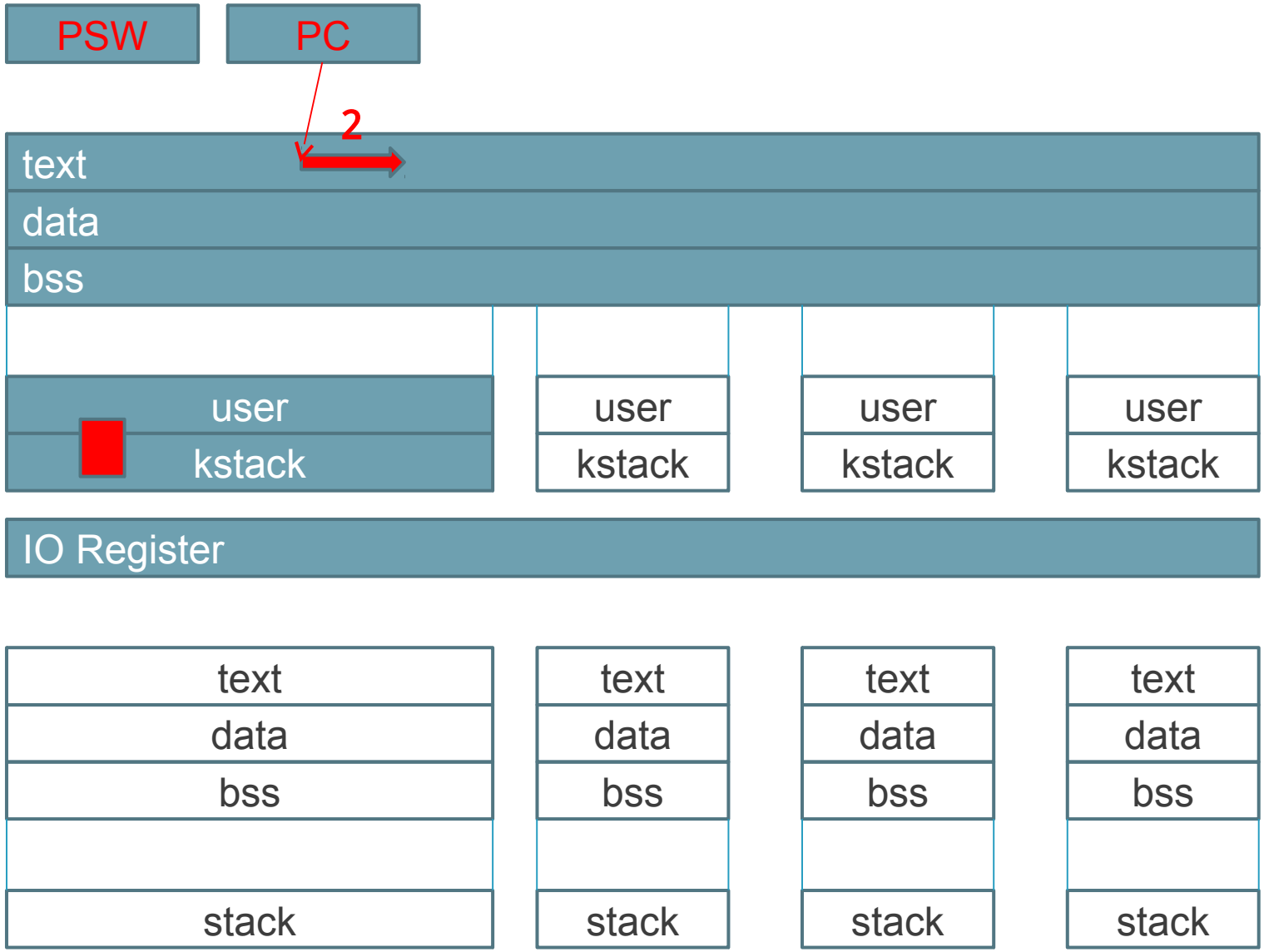


- Select the entry to the PC of the interrupt vector

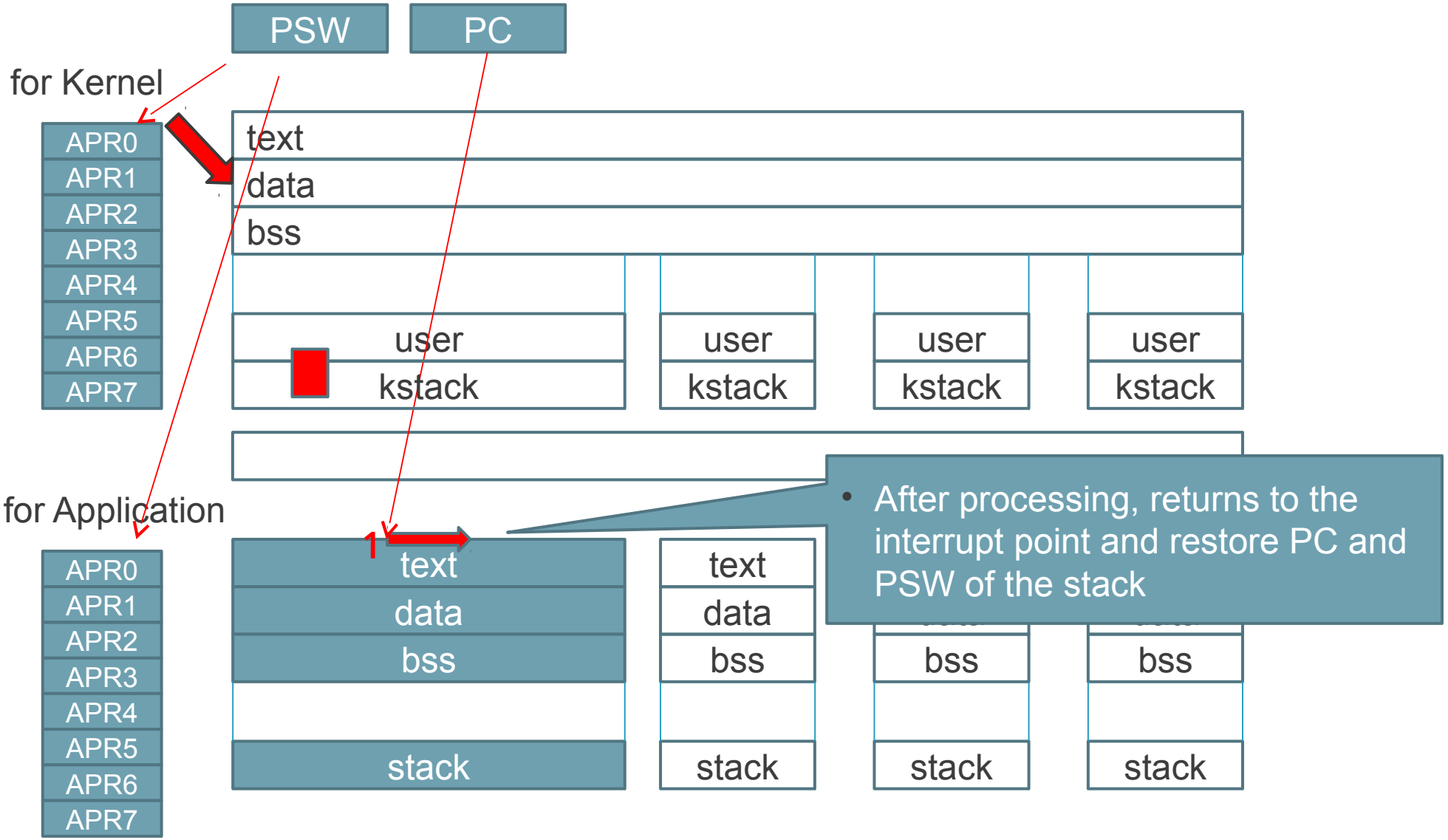
Vector	device
004	Bus Timeout
010	Illegal instruction
014	Bpt-trace
024	iot
034	Power failur
114	Emulator tarp instruction
240	11/10 parity
244	Floting point error
250	Segmentation violation

Vector	device	entry
060	Teletype input	klin
064	Teletype output	klou
070	Paper tape input	pcin
074	Paper tape output	pcou
100	Line clock	kwlp
104	Programmable clock	kwlp
200	Line printer	lpou
220	RK disk driver	rkio

interrupt handler



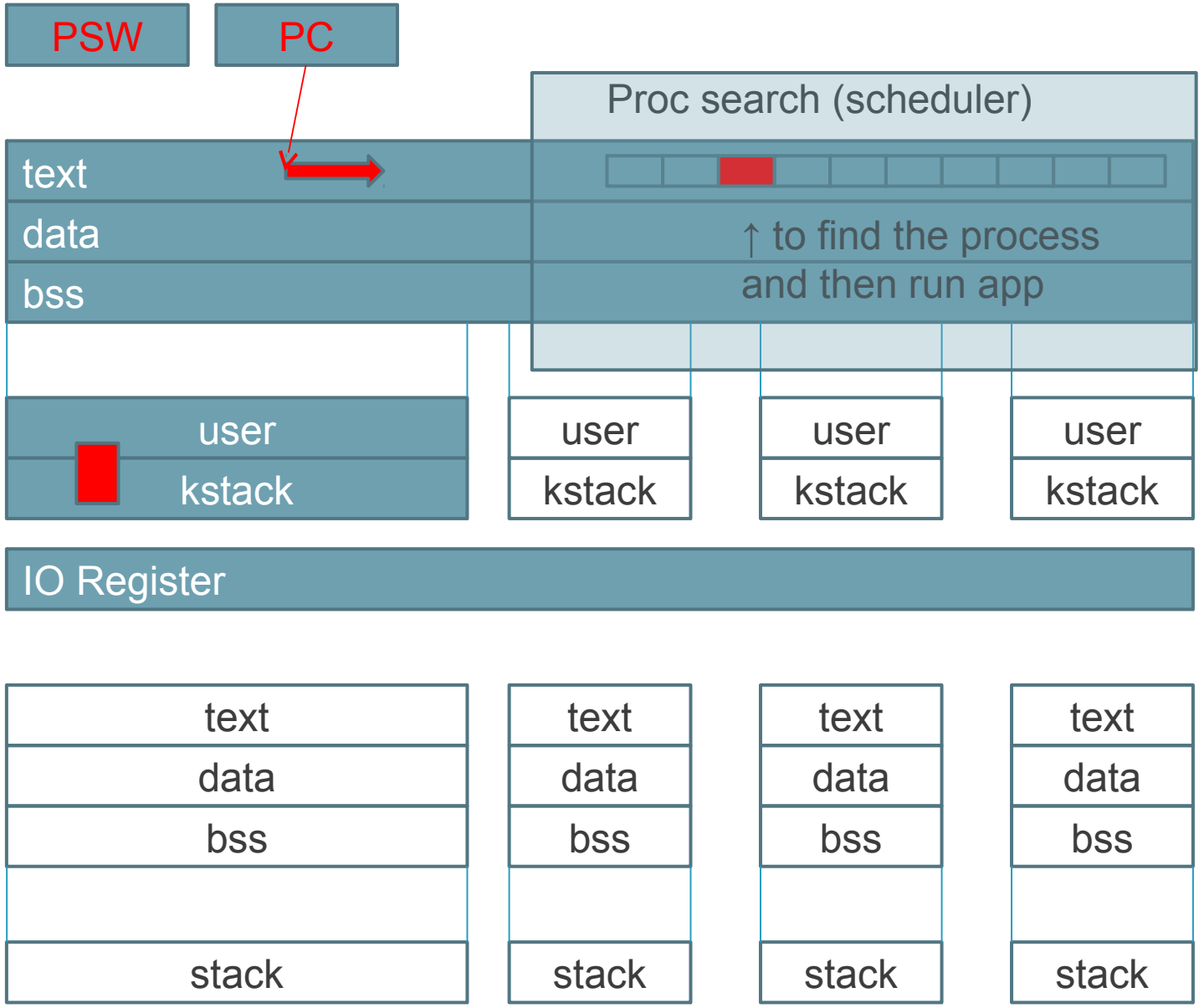
Return from interrupt



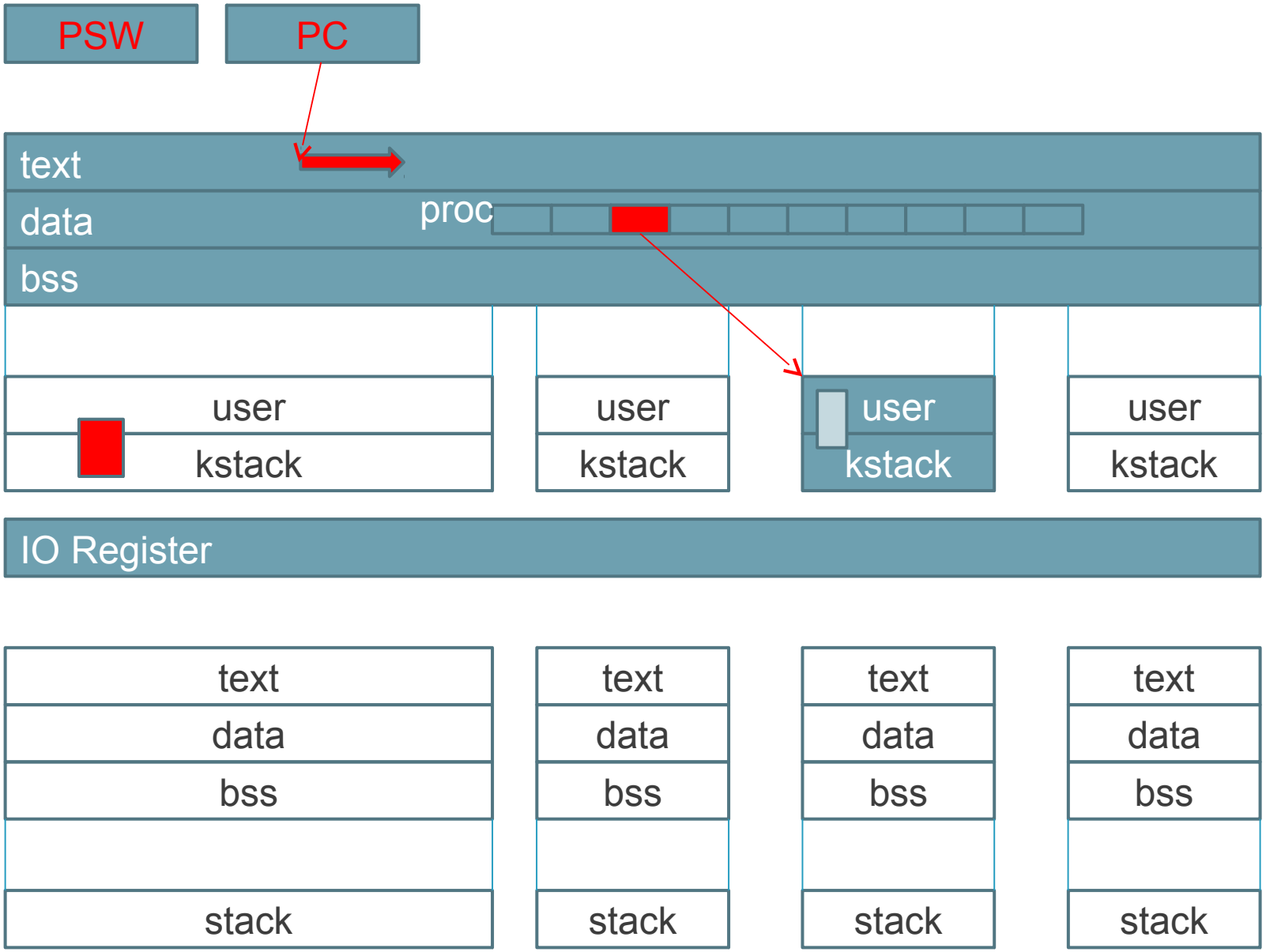
Switch

- The above description, from the time when an interrupt occurs, switching of the processes do not occur before the return
- Depending on the condition of the process switch occurs in the following processing
  - Search and selection of proc
  - switching of the user structure
  - Restore the saved state structure to + kstack User
  - Return to the application

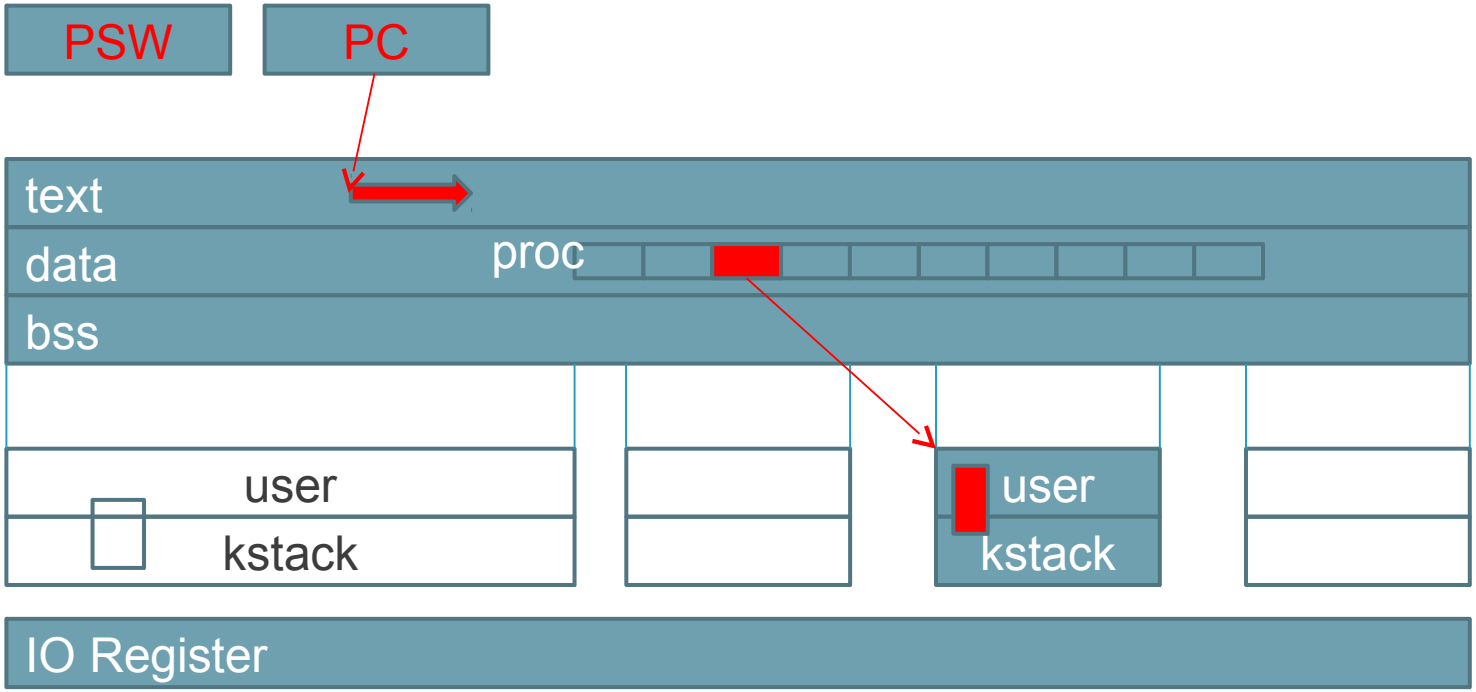
Process Switch: selection



Process Switch: user

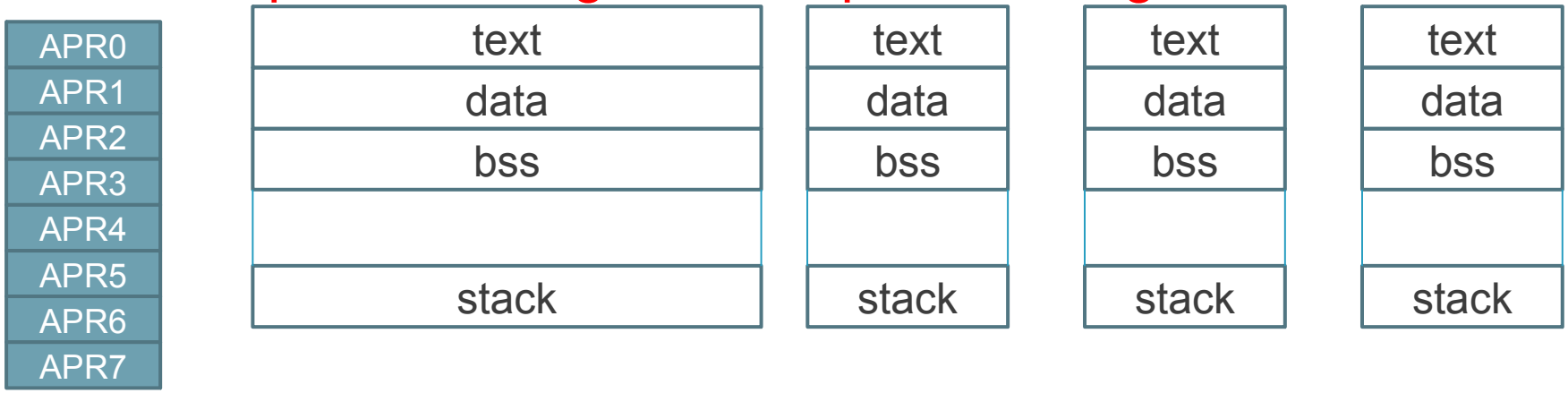


Process Switch: user

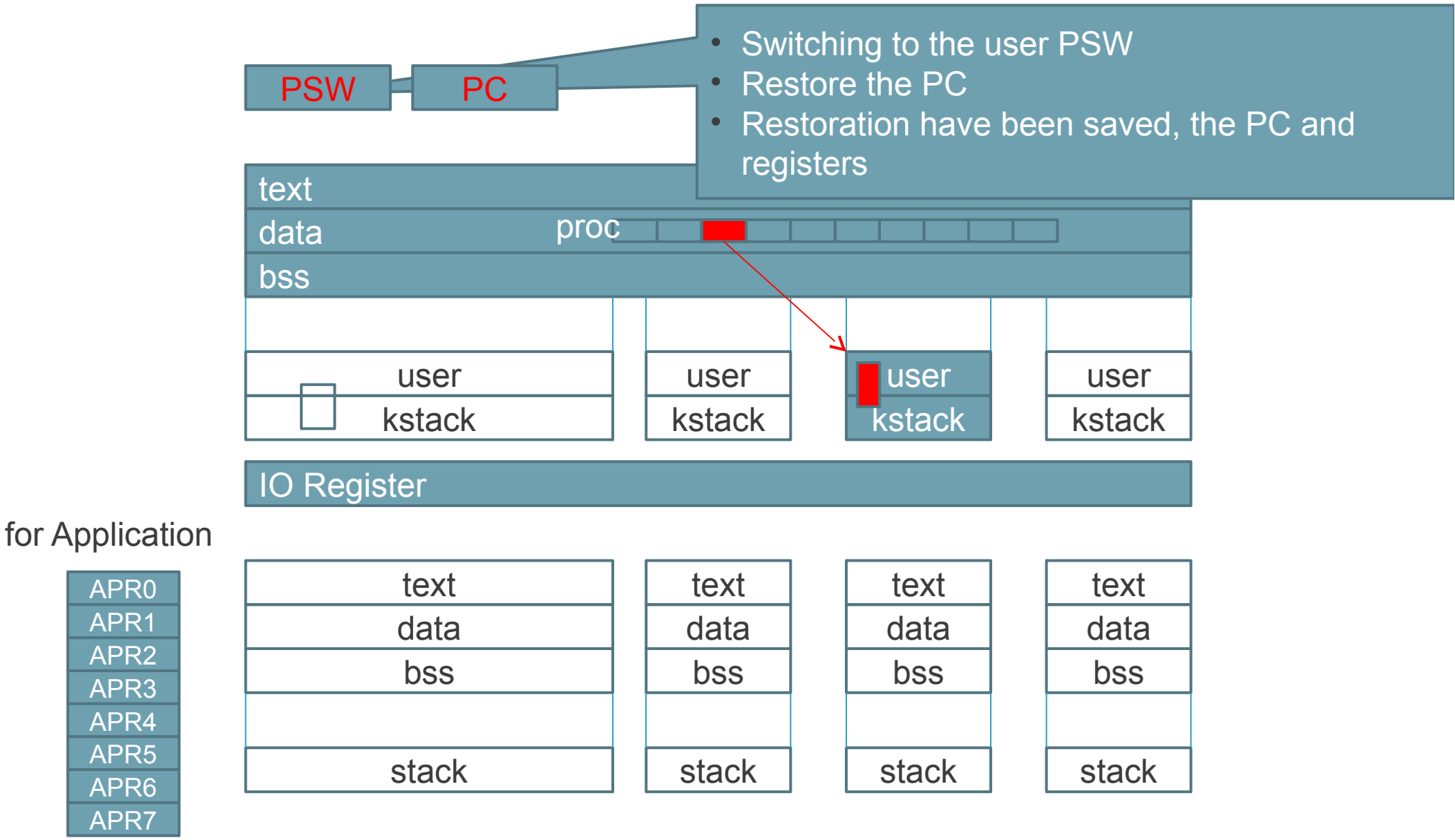


for Application

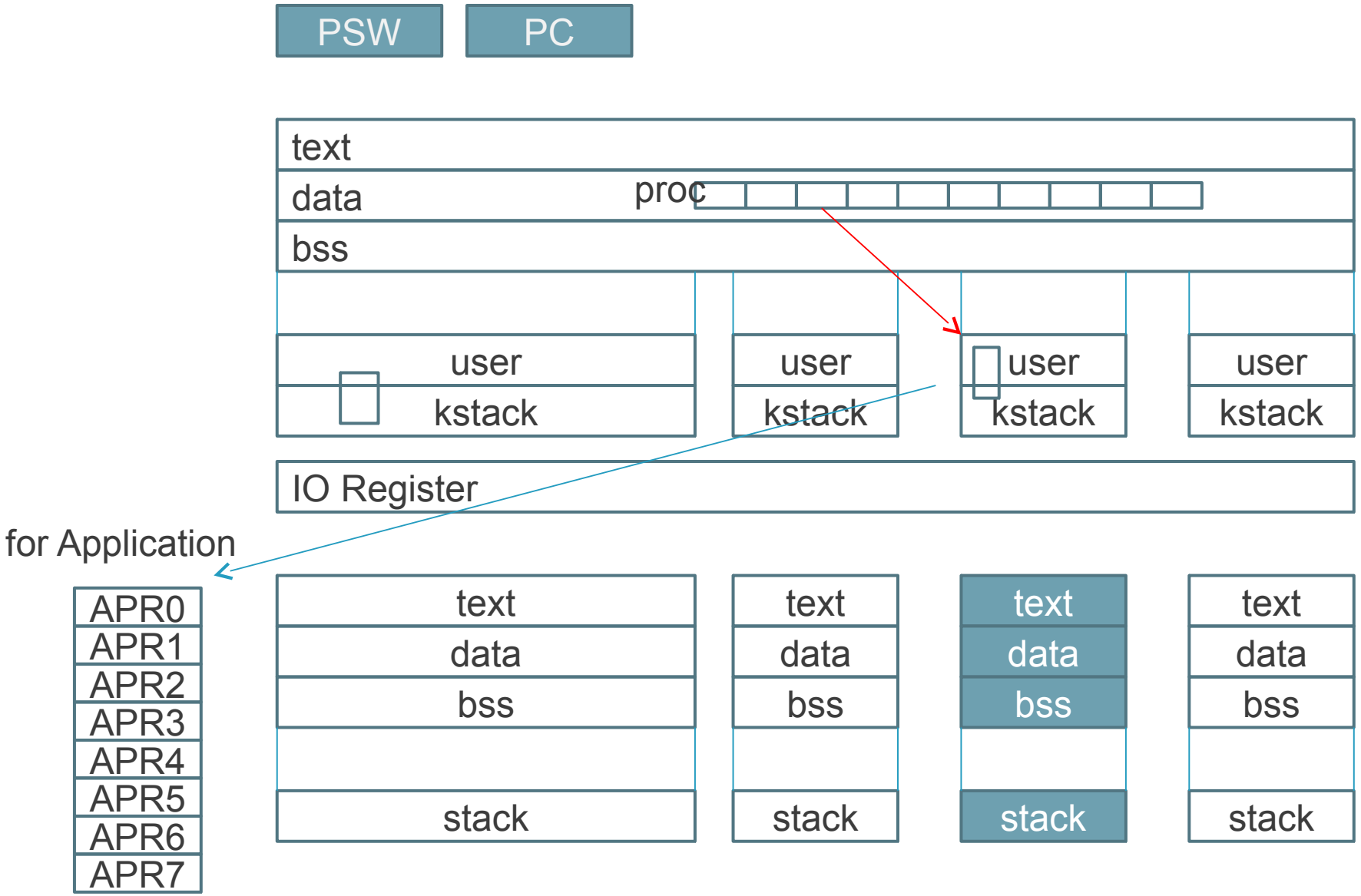
Update the register for space management



Process Switch: mode



Process Switch: done



switch [slp.c]

```
2178: swtch()
2179: {
2180:         static struct proc *p;
2181:         register i, n;
2182:         register struct proc *rp;
2183:
2184:         if(p == NULL)
2185:                 p = &proc[0];
2186:         /*
2187:          * Remember stack of caller
2188:          */
2189:         savu(u.u_rsav); /* save r5,r6 when exchanging stacks */

2190:         /*
2191:          * Switch to scheduler's stack
2192:          */
2193:         retu(proc[0].p_addr); /* address of swappable image */
```

switch [slp.c]

```
2195: loop:
2196:           runrun = 0;
2197:           rp = p;
2198:           p = NULL;
2199:           n = 128;
2203:           i = NPROC;
2204:           do {
2205:                   rp++;
2206:                   if(rp >= &proc[NPROC])
2207:                           rp = &proc[0];
2208:                   if(rp->p_stat==SRUN &&(rp->p_flag&SLOAD)!=0) {
2209:                           if(rp->p_pri < n) {
2210:                                   p = rp;
2211:                                   n = rp->p_pri;
2212:                           }
2213:                   }
2214:           } while(--i);
```

switch [slp.c]

```
2223:           rp = p;
2224:           curpri = n;
2225:           /* Switch to stack of the new process and set up
2226:            * his segmentation registers.
2227:            */
2228:           retu(rp->p_addr);       /* address of swappable image */
2229:           sureg();
2247:           return(1);
```

Suspicious function

```
2189:           savu(u.u_rsav);
```

Stores the values of r5 and r6 to u.u_rsav

```
2193:           retu(proc[0].p_addr);
2228:           retu(rp->p_addr);
```

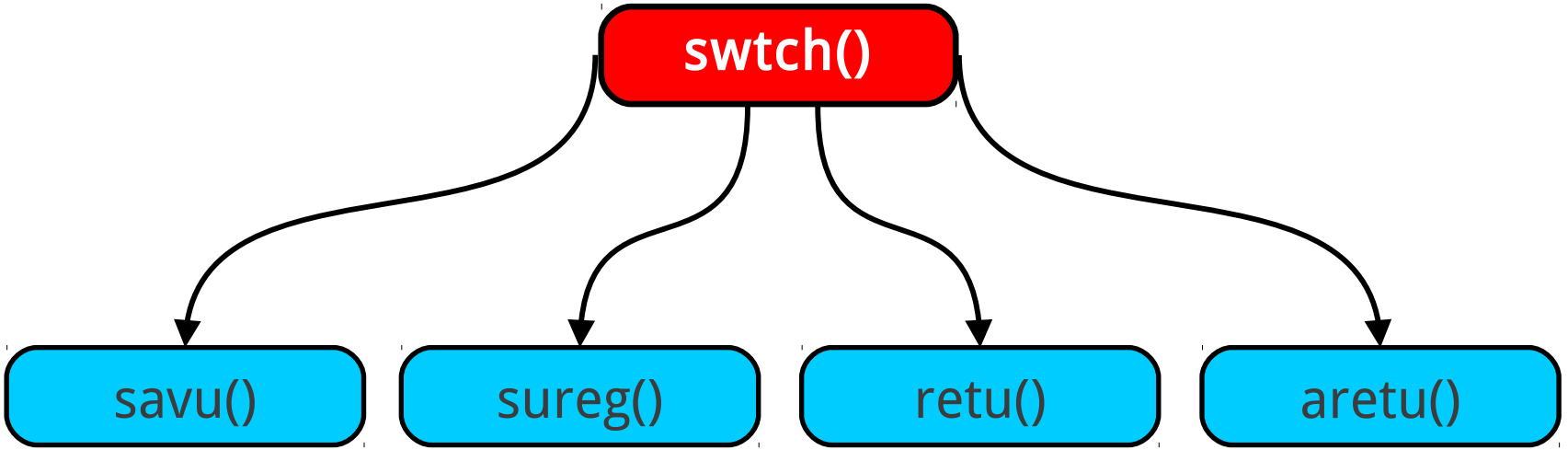
Loads values of r5 and r6 from p->addr

Resets 7th kernel segmentation address register(pc)

```
2229:           sureg();
```

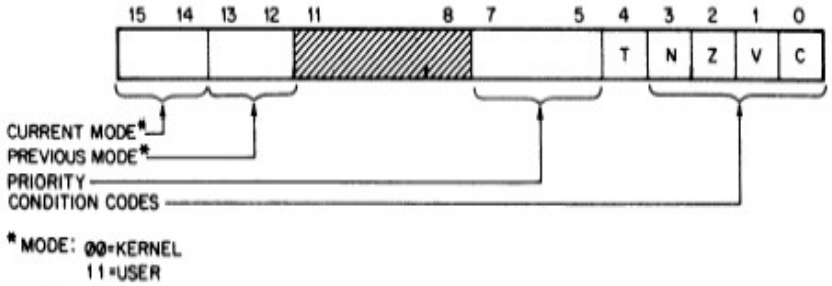
Loads physical segmentation registers with values in user data structure

Begin Flow Chart



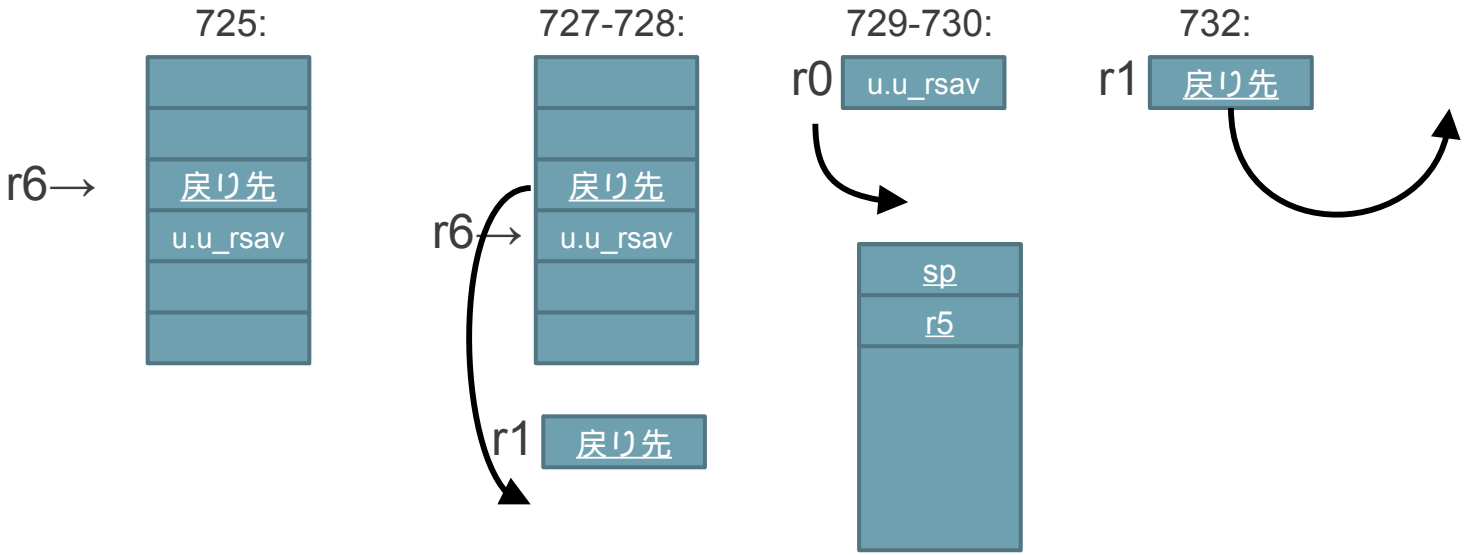
savu [m40.s]

```
0725: _savu:
0726:           bis     $340,PS
0727:           mov     (sp)+,r1
0728:           mov     (sp),r0
0729:           mov     sp,(r0)+
0730:           mov     r5,(r0)+
0731:           bic     $340,PS
0732:           jmp     (r1)
```



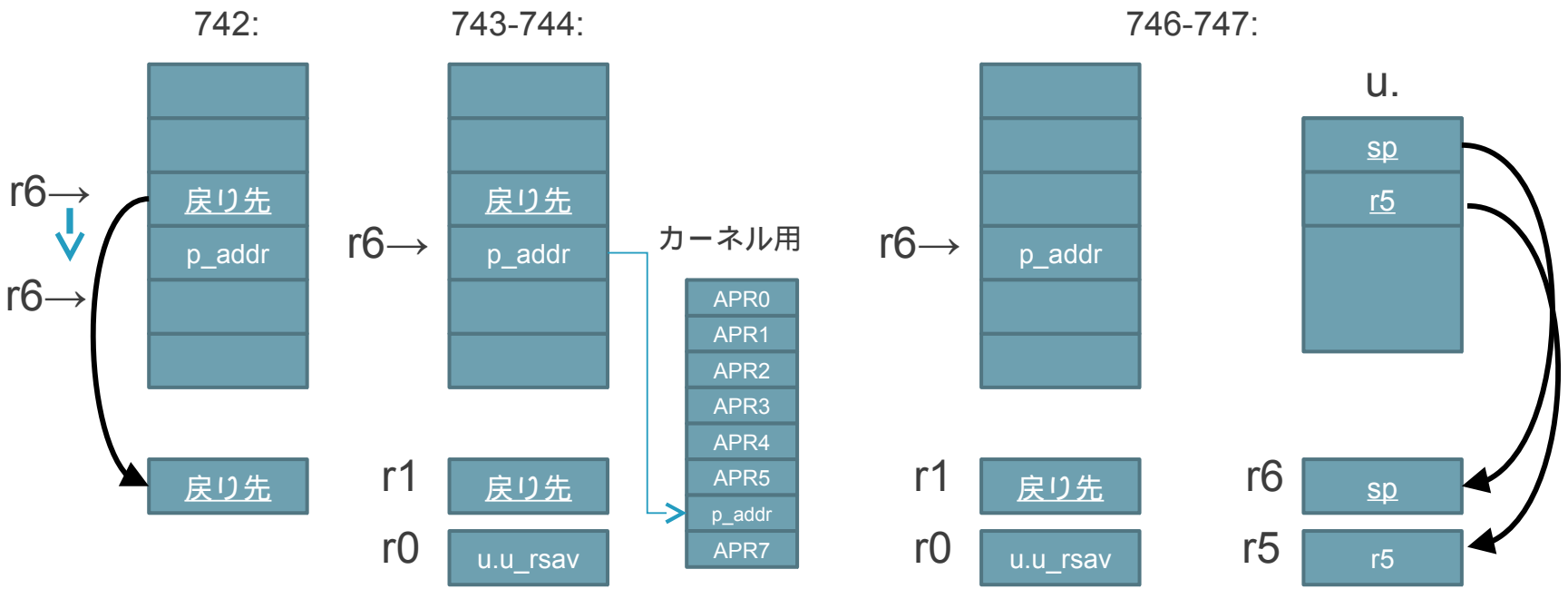
bis　$340,PS で bit5-7 を 1 に

bic　$340,PS で bit5-7 を 0 に



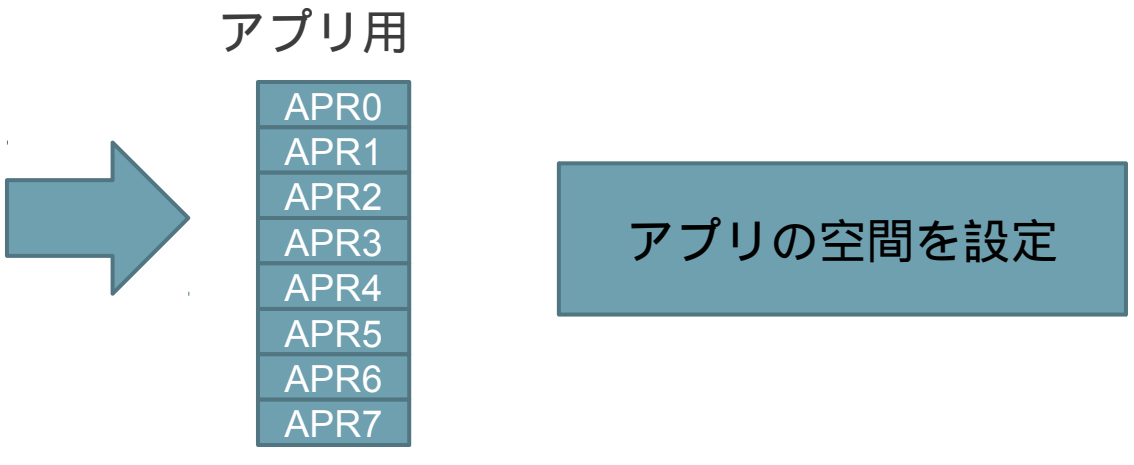
retu [m40.s]

```
0740: _retu:
0741:           bis     $340,PS
0742:           mov     (sp)+,r1
0743:           mov     (sp),KISA6
0744:           mov     $_u,r0
0745: 1:
0746:           mov     (r0)+,sp
0747:           mov     (r0)+,r5
0748:           bic     $340,PS
0749:           jmp     (r1)
```

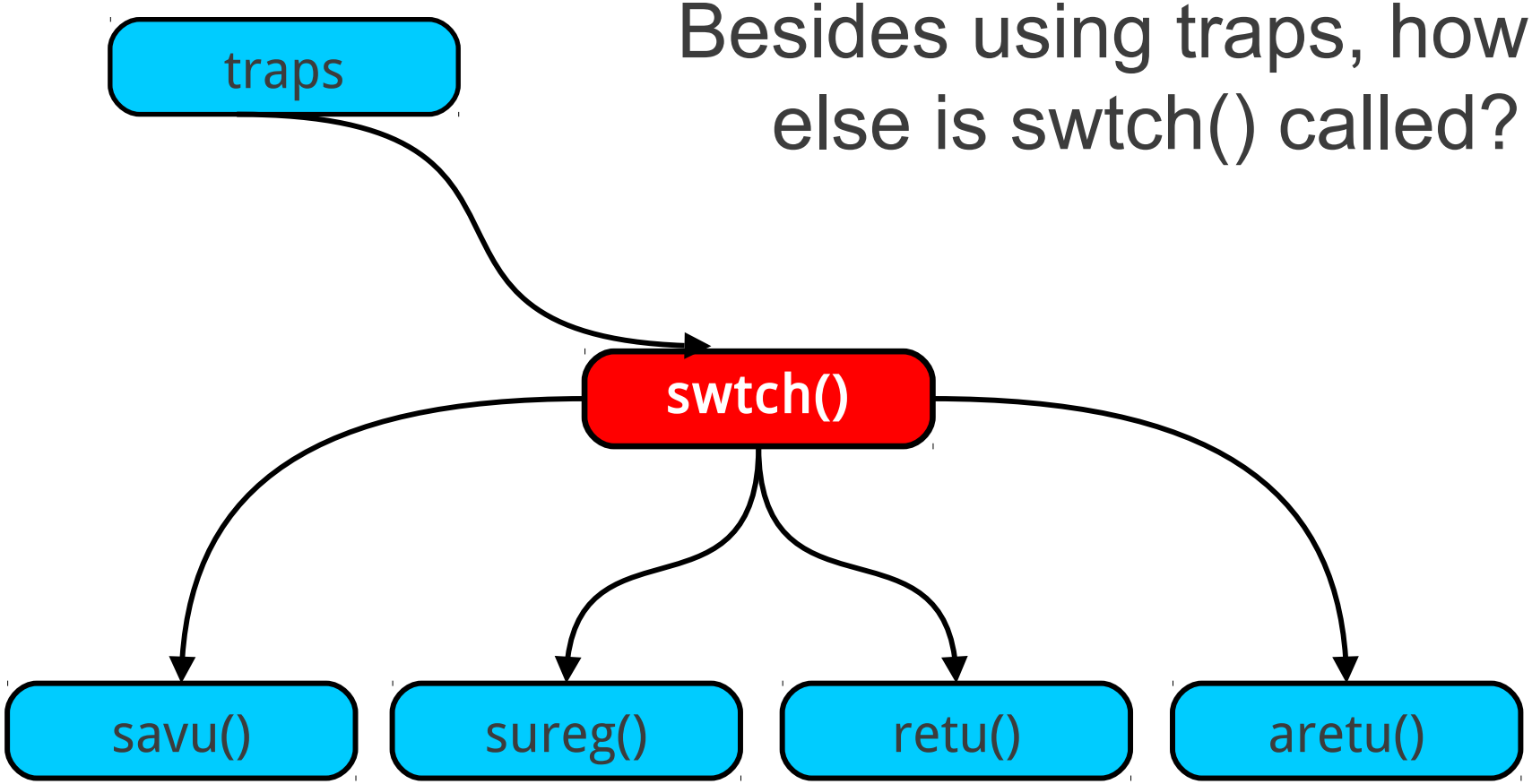


sureg [main.c]

```
1739: sureg()
1740: {
1741:           register *up, *rp, a;
1742:
1743:           a = u.u_procp->p_addr;
1744:           up = &u.u_uisa[16];
1745:           rp = &UISA->r[16];
1746:           if(cputype == 40) {
1747:                   up =- 8;
1748:                   rp =- 8;
1749:           }
1750:           while(rp > &UISA->r[0])
1751:                   *--rp = *--up + a;
1754:           up = &u.u_uisd[16];
1755:           rp = &UISD->r[16];
1765: }
```



Current Flow Chart



Besides using traps, how else is swtch() called?

sleep()

```
sleep(chan, pri)
{
   register *rp, s;

   s = PS->integ;
   rp = u.u_procp;
   if(pri >= 0) {
       if(issig())
              goto psig;
       spl6();
       rp->p_wchan = chan;
       rp->p_stat = SWAIT;
       rp->p_pri = pri;
       spl0();
       if(runin != 0) {
              runin = 0;

   wakeup(&runin);
       }
       ⋮
```

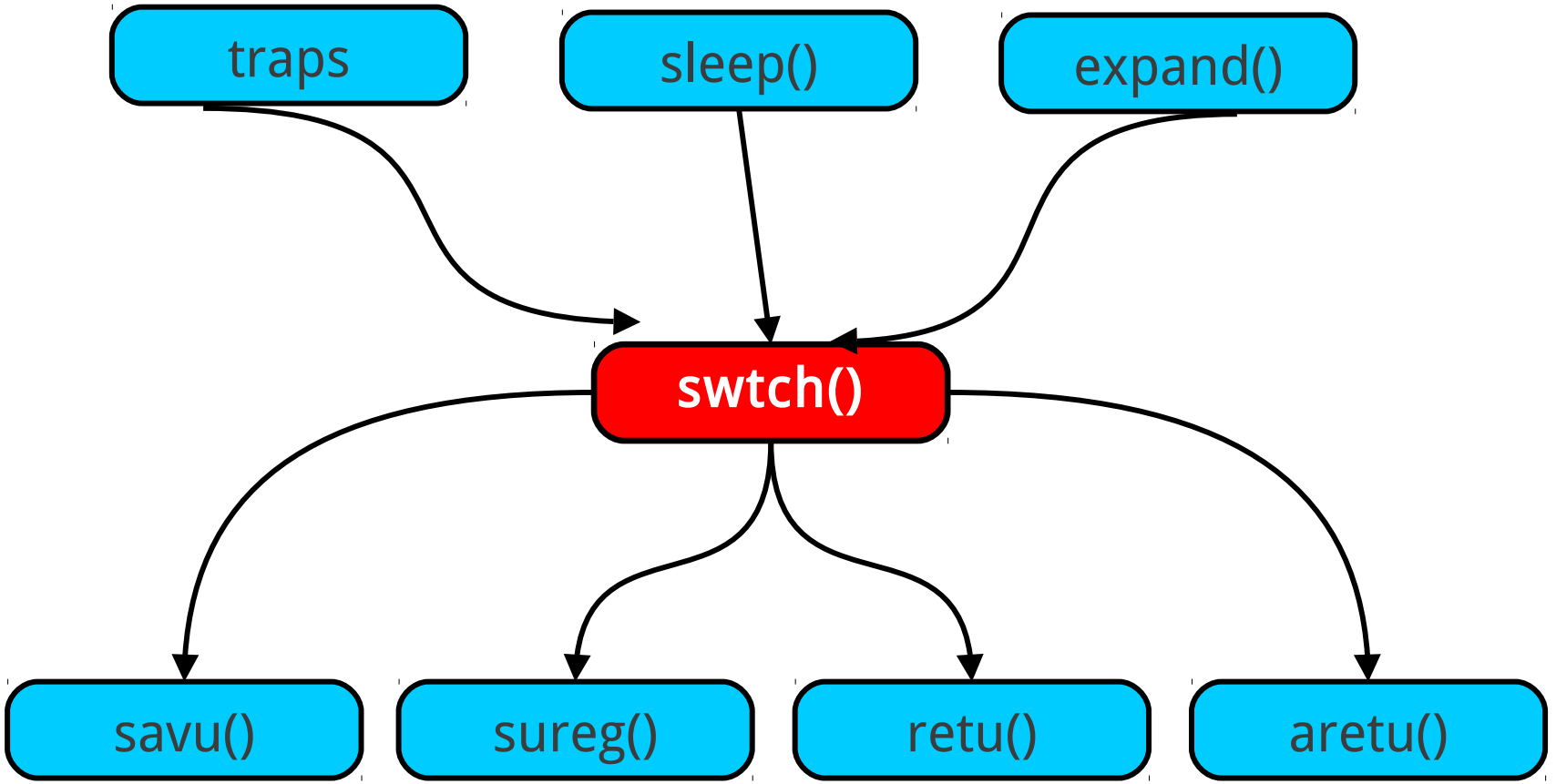
```
       ⋮
       swtch();
       if(issig())
              goto psig;
   } else {
       spl6();
       rp->p_wchan = chan;
       rp->p_stat =
   SSLEEP;
       rp->p_pri = pri;
       spl0();
       swtch();
   }
   PS->integ = s;
   return;
psig:
   aretu(u.u_qsav);
}
```

expand()

```
expand(newsize)
{
    int i, n;
    register *p, a1, a2;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    a1 = p->p_addr;
    if(n >= newsize) {
        mfree(coremap, n-newsize,
              a1+newsize);
        return;
    }
    savu(u.u_rsav);
    a2 = malloc(coremap, newsize);
    if(a2 == NULL) {
          savu(u.u_ssav);
          xswap(p, 1, n);
          p->p_flag =| SSWAP;
          swtch();  /* no return */
    }
    p->p_addr = a2;
    for(i=0; i<n; i++)
    copyseg(a1+i, a2++); 2292
    mfree(coremap, n, a1);
    retu(p->p_addr);
    sureg();
}
```

Completed Flow Chart



simh 操作

Simh のインストールと起動

- Ubuntu の場合

$ sudo apt-get install simh

- pdp-11 のシュミレーション

$ pdp11

PDP-11 simulator V3.8-1

sim>

Unix v6 のダウンロード

- simh 用 Disk イメージのアーカイブ
  - http://simh.trailing-edge.com/software.html
- V6 のイメージは以下
  - http://simh.trailing-edge.com/kits/uv6swre.zip
- ダウンロード

$ wget http://simh.trailing-edge.com/kits/uv6swre.zip

Disk イメージの展開

```
$ unzip uv6swre.zip
Archive:  uv6swre.zip
  inflating: README.txt
  inflating: unix3_v6_rk.dsk
  inflating: unix1_v6_rk.dsk
  inflating: unix2_v6_rk.dsk
  inflating: unix0_v6_rk.dsk
  inflating: AncientUnix.pdf
$
```

設定ファイルの作成

- uv6swre.zip を展開したディレクトリで以下のファイル unixv6.cf を作成

```
$cat unixv6.cfg

set cpu 11/40

set cpu u18

att rk0 unix0_v6_rk.dsk

att rk1 unix1_v6_rk.dsk

att rk2 unix2_v6_rk.dsk

att rk3 unix3_v6_rk.dsk

boot rk0

$
```

Simh の起動

```
$ pdp11 unixv6.cfg

PDP-11 simulator V3.8-1
Disabling XQ
@unix

Login: root
#
```

Simh の debug 機能

- simh モードへの移行と復帰

Ctrl+E

Simulation stopped, PC: 021630 (MOV (SP)+,177776)

sim> c

#

  - simh モードへは Ctrl+E で
  - シミュレーションモードへは c で

ディバック機能を使う

- カーネルのシンボルアドレスを調べる

```
# chdir /
# nm unix | grep savu
021636T _savu
#
```

- savu() のアドレスは021636, breakpoint を指定する

```
# Ctrl+E
Simulation stopped, PC: 021630 (MOV (SP)+,177776)
sim> break 021636
sim> c

Breakpoint, PC: 021636 (SPL 6)
sim>
```

step 実行

```
Breakpoint, PC: 021636 (SPL 6)
sim> e r5,sp,pc
R5:   141742
SP:   141730
PC:   021636
sim> s

Step expired, PC: 021640 (MOV (SP)+,R1)
sim> s

Step expired, PC: 021642 (MOV (SP),R0)
sim> s

Step expired, PC: 021644 (MOV SP,(R0)+)
sim> s

Step expired, PC: 021646 (MOV R5,(R0)+)
sim> s

Step expired, PC: 021650 (SPL 0)
sim> s

Step expired, PC: 021652 (JMP (R1))
```

```
0725: _savu:
0726:       bis    $340,PS
0727:       mov    (sp)+,r1
0728:       mov    (sp),r0
0729:       mov    sp,(r0)+
0730:       mov    r5,(r0)+
0731:       bic    $340,PS
0732:       jmp    (r1)
```

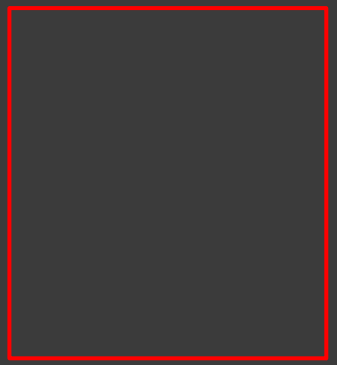
レジスタの調べ方

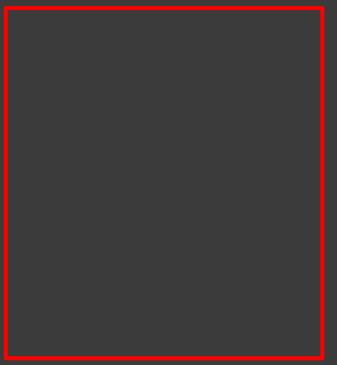
/usr/share/doc/simh/simh_doc.pdf で説明

e {xamine} <list> examine memory or registers

sim> e state　　// レジスタ等すべてを表示

```
PC:       021630
R0:       140004
R1:       034272
R2:       005336
R3:       000200
R4:       000000
R5:       141724
SP:       141710
R00:      140004
R01:      034272
R02:      005336
R03:      000200
R04:      000000
R05:      141724
R10:      000000
R11:      000000
R12:      000000
R13:      000000
R14:      000000
R15:      000000
KSP:      141710
SSP:      000000
USP:      177756
PSW:      030000
CM:       0
PM:       0
RS:       0
FPD:      0
IPL:      0
T:        0
N:        0
Z:        0
V:        0
C:        0
PIRQ:     000000
STKLIM:   000000
FAC0H:    00000000000
FAC0L:    00000000000
FAC1H:    00000000000
FAC1L:    00000000000
FAC2H:    00000000000
FAC2L:    00000000000
FAC3H:    00000000000
FAC3L:    00000000000
FAC4H:    00000000000
FAC4L:    00000000000
FAC5H:    00000000000
```





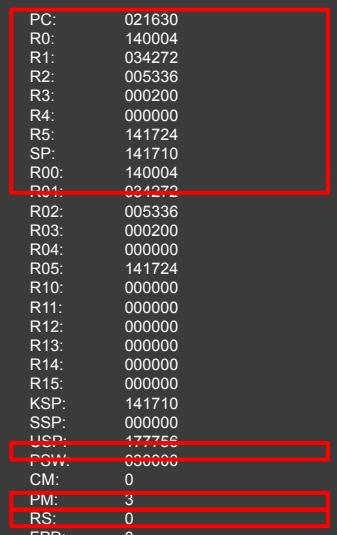
必要な内容に絞り込む

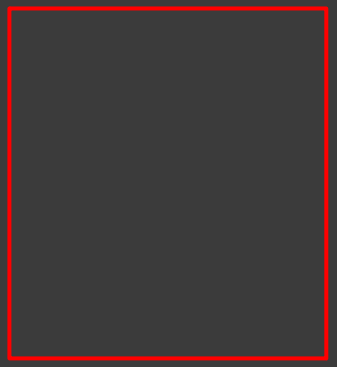
PC,R0 ~ R5,SP(KSP,USP)

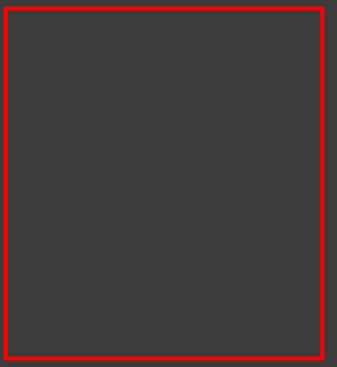
PSW

KDPAR0 ~ 7

UDPAR0 ~ 7







レジスタの指定方法

State の表示順に’ -’ で範囲を指定

```
sim> e pc-sp
PC:   021630
R0:   140004
R1:   034272
R2:   005336
R3:   000200
R4:   000000
R5:   141724
SP:   141710
```

MMU レジスタの指定方法

Stateの表示順に’-’で範囲を指定

```
sim> e KDPAR0-KDPDR7
KDPAR0:  000000
KDPDR0: 077506
KDPAR1:  000200
KDPDR1: 077506
KDPAR2:  000400
KDPDR2: 077506
KDPAR3:  000600
KDPDR3: 077406
KDPAR4:  001000
KDPDR4: 077406
KDPAR5:  001200
KDPDR5: 077406
KDPAR6:  001171
KDPDR6: 077506
KDPAR7:  007600
KDPDR7: 077506
```

Pdp11/40 では、 KPAR,KPDR と呼んでいたが、 simh では上位機種のレジスタ名で表示されるので、 KDPAR,KDPDR と読み替える

メモリの調べ方

```
sim> e 0        //  アドレス 0 を表示
0: 000417
sim> e 0/4      //  アドレス 0 から 4 バイトを表示
0: 000417
2: 000004
sim> e 0-2      //  アドレス 0 から 2 までを表示
0: 000417
2: 000004
```

表示フォーマットの指定

-a　ASCII で表示
-c　文字列で表示
-m　命令列で表示
-o　8 進で表示
-x　16 進で表示

```
sim> e -m 54104/20
54104:  000013
54106:  MOV R0,-(SP)
54110:  BIC #177400,(SP)
54114:  JSR PC,@#22100
54120:  ADD #6,SP
```

シンボルからアドレスを調べる

```
sim> c                  // back to UNIX
# nm /unix | grep main   // address of main()
022272T _main
# nm /unix | grep proc
034476T _newproc
005206B _proc
043170T _procxmt
```

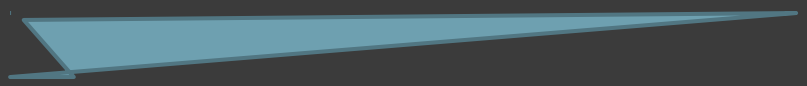
examine the MMU to break in main()

```
sim>  break 022272
sim> boot rk0
@unix

Breakpoint, PC: 022272 (JSR R5,22240)
sim> e KDPAR0-KDPDR7
KDPAR0:     000000
KDPDR0:     077506
KDPAR1:     000200
KDPDR1:     077506
KDPAR2:     000400
KDPDR2:     077506
KDPAR3:     000600
KDPDR3:     077406
KDPAR4:     001000
KDPDR4:     077406
KDPAR5:     001200
KDPDR5:     077406
KDPAR6:     001171
KDPDR6:     077506
KDPAR7:     007600
KDPDR7:     077506
sim>
```

自分の proc を見る #1

```
# nm /unix | grep _write
054124T _write
031700T _writei
050176T _writep
#
Simulation stopped, PC: 021630 (MOV (SP)+,177776)
sim> break 54124
sim> c
[Enter]
Breakpoint, PC: 054124 (JSR R5,22240)
sim>
```



Shell place to break in to write a #
(entry of the kernel)

自分の proc を見る #2

- proc のアドレスは u.u_procp でわかる
- u は常にカーネル仮想の 140000 番地
- 一方で simh の debug 機能は仮想アドレスを扱えない
- 自分で変換する

```
sim> e KDPAR6
KDPAR6:  001477      // 64 バイトを 1 単位とした値
```

- 64 バイト = 6 ビットシフト = 8 進数で 00
- User は 00147700 にあるはず

自分の proc を見る #3

```
sim> e 147700/100
147700:  141746   u_rsav[0]
147702:  141756   u_rsav[1]
147704:  000200   u_fsav[0]
147706:  000000   1
147710:  000000   2
147712:  000000   3
147714:  000000   4
147716:  000000   5
   :
   :
147764:  000000   24
147766:  000000   u_segflag/u_error
147770:  001400   u_uid/u_gid
147772:  001400   u_ruid/u_rgid
147774:  005262   u_procp
147776:  177737
```

```
0413: struct user
0414: {
0415:       int    u_rsav[2];      /* save r5,r6 when exchanging stacks */
0416:       int    u_fsav[25];     /* save fp registers */
0417:                               /* rsav and fsav must be first in structure */
0418:       char   u_segflg;       /* flag for IO; user or kernel space */
0419:       char   u_error;        /* return error code */
0420:       char   u_uid;          /* effective user id */
0421:       char   u_gid;           /* effective group id */
0422:       char   u_ruid;          /* real user id */
0423:       char   u_rgid;          /* real group id */
0424:       int    u_procp;         /* pointer to proc structure */
```

自分の proc を見る #4

```
sim> e -h 5262/40
5262:   0103  p_flag/p_stat
5264:   0064  p_sig/p_pri
5266:   7F00  p_time/p_uid
5270:   0000  p_cpu
5272:   42F2  p_ttyp
5274:   000E  p_pi
5276:   0001  p_ppi
5300:   033F
5302:   0048
5304:   0000
5306:   0EDC
5310:   0000
5312:   00CE
5314:   0000
5316:   0000
5320:   0000
```

pid は 0xe=14(10 進)

```
0358: struct  proc
0359: {
0360:       char    p_stat;
0361:       char    p_flag;
0362:       char    p_pri;        /* priority, negative is high */
0363:       char    p_sig;        /* signal number sent to this process */
0364:       char    p_uid;        /* user id, used to direct tty signals */
0365:       char    p_time;       /* resident time for scheduling */
0366:       char    p_cpu;        /* cpu usage for scheduling */
0367:       char    p_nice;       /* nice for scheduling */
0368:       int     p_ttyp;       /* controlling tty */
0369:       int     p_pid;        /* unique process id */
0370:       int     p_ppid;       /* process id of parent */
0371:       int     p_addr;       /* address of swappable image */
0372:       int     p_size;       /* size of swappable image (*64 bytes) */
0373:       int     p_wchan;      /* event process is awaiting */
0374:       int     *p_textp;     /* pointer to text structure */
0375:
0376: } proc[NPROC];
```

自分の proc を見る #5

Breakpoint, PC: 054104 (JSR R5,22240)

sim> nobreak 54104

sim> c

ps

 14 -

 29 ps

#