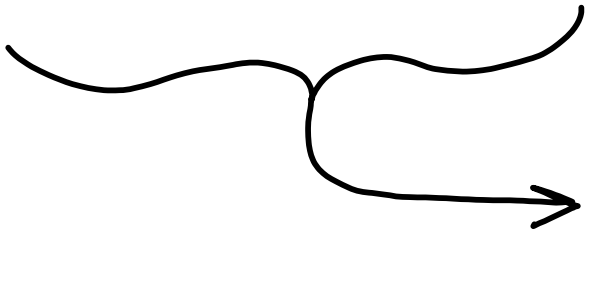
LC3 System Start-Up Assumptions
We will write an OS for the LC3.
What would a real LC3 do at start up?

1. BIOS execution
--- PC points to BIOS (Basic IO System).
--- POST: Test and initialize hardware.
--- BOOT: Read disk block 0 (512B);
--- BOOT: Store boot block at x3000;
--- BOOT: JMP x3000.

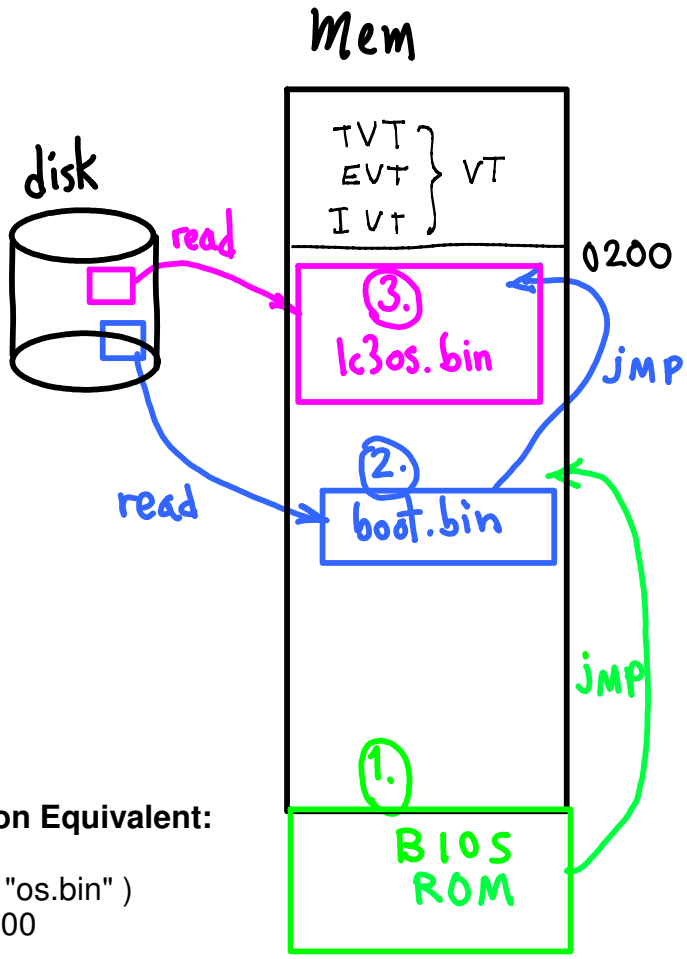
2. Booter execution
--- Read OS disk blocks, load at x0200,
--- JMP x0200

3. OS execution (JMP)
--- OS code begins execution at x0200.



Our Simulation Equivalent:

-- readmemb("os.bin")
-- PC <== x0200



Init OS

OS code intialization
(interrupts are disabled: PSR.Priority == 7)

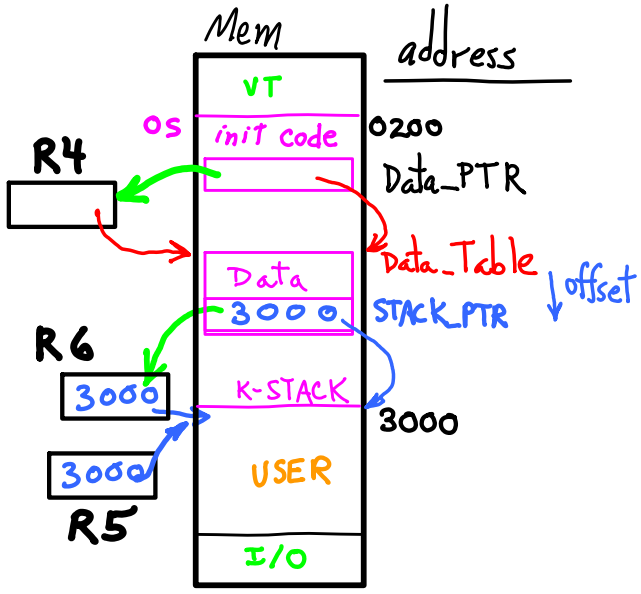
-- **set Supervisor's GDP (R4) SP (R6) BP (R4)**
Note: can't use subroutines until SP is set up.

-- **set up VT**
Note: can't use traps until VT is set up.
Initialize service routines, e.g.,
JSR kbInt_init_BEGIN

-- **turn on INTs**, PSR.priority <== 3'b000

-- **jump to OS command loop (also, service INTs)**

```
while( 1 ) {
        display_prompt();
        command = get_response();
        switch (command) {
            case "r": do_run(); break;
            case "q": do_quit(); break;
            case "h": do_help(); break;
        }
}
```



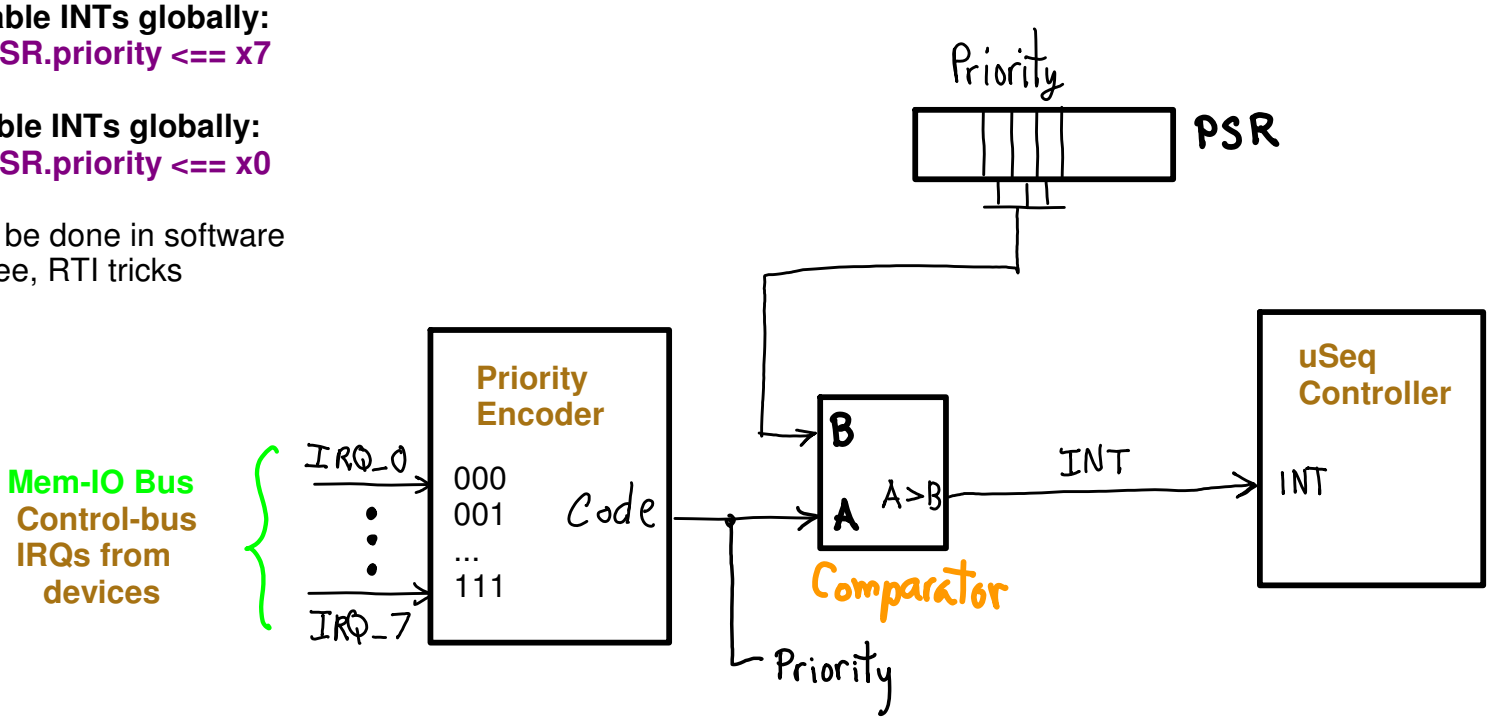
;---- OS initialization, Set up GDP, SP, BP

```
LEA R4, Data_PTR
LDR R4, R4, #0
LDR R6, R4, #(offset to Stack_PTR)
ADD R5, R6, #0

Data_PTR:    .FILL   Data_Table
```

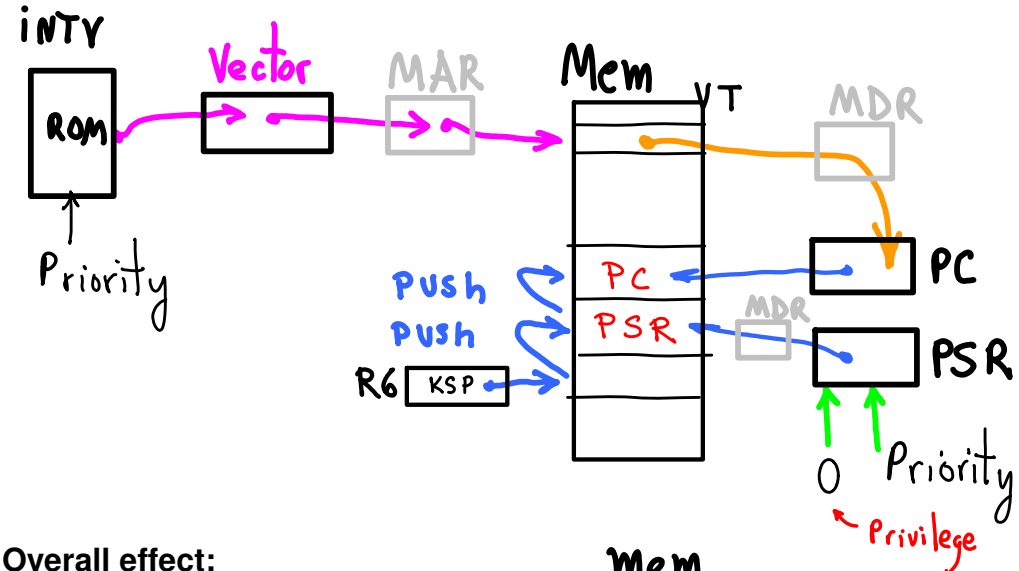
Disable INTs globally:
--- PSR.priority <== x7

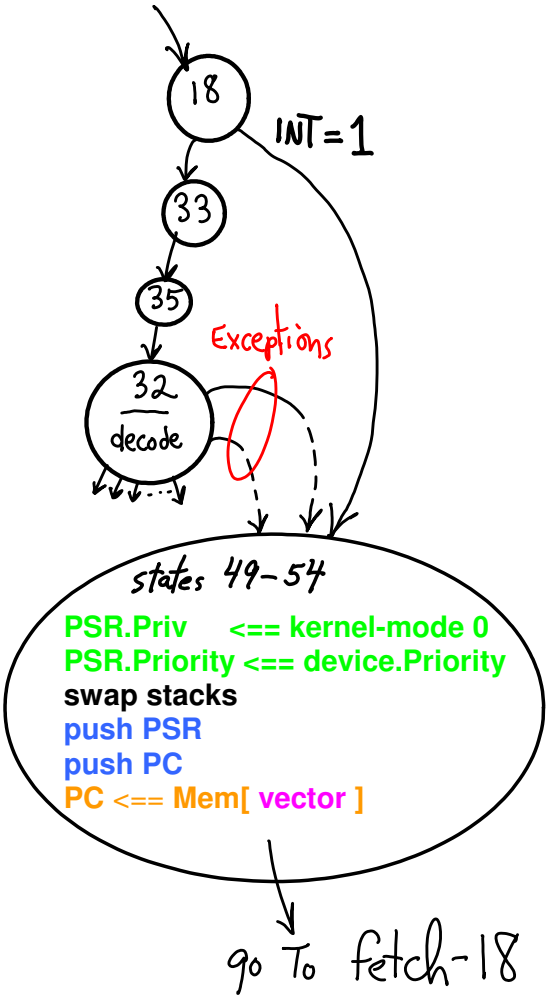
Enable INTs globally:
--- PSR.priority <== x0

Can be done in software
--- see, RTI tricks



--- I/O device i sets **IRQ_i = 1**.
--- Highest priority goes to comparator.
--- **If (Code > PSR.priority)**
------ **INT <== 1**
------ **uSeq branches from state 18 to 49**
------ **uSeq saves state,**
------ **jumps via VT**
------ handler **jumps back via RTI** execution.



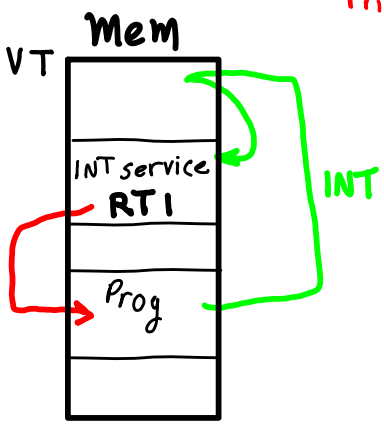


Overall effect:

--- **Save** program's state,
Jump to service routine

--- **Service** the INT request

--- **Restore** program's state
Jump back to instruction



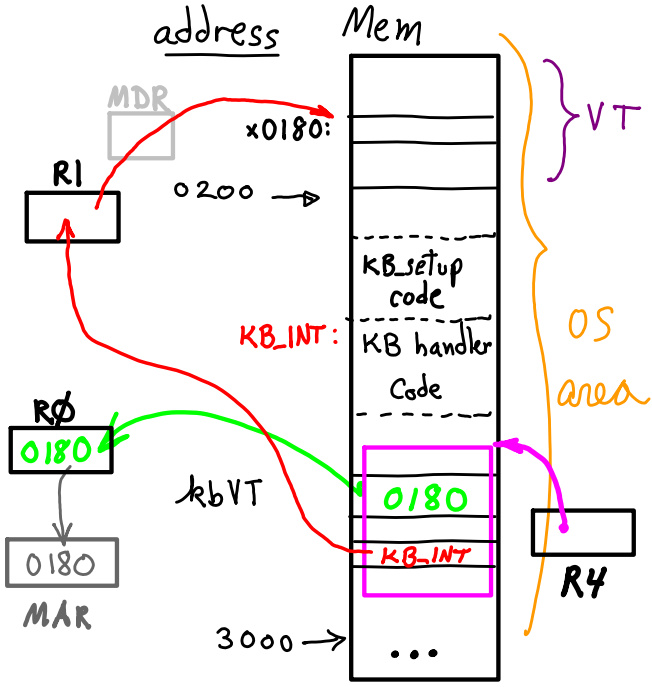
Note:

kernel = supervisor
ksp = SSP

Init the Service Routine

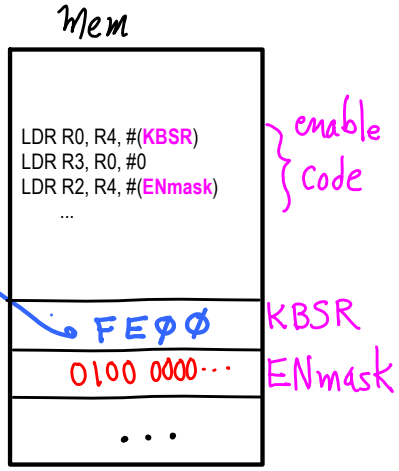
Each service has its own init. routine

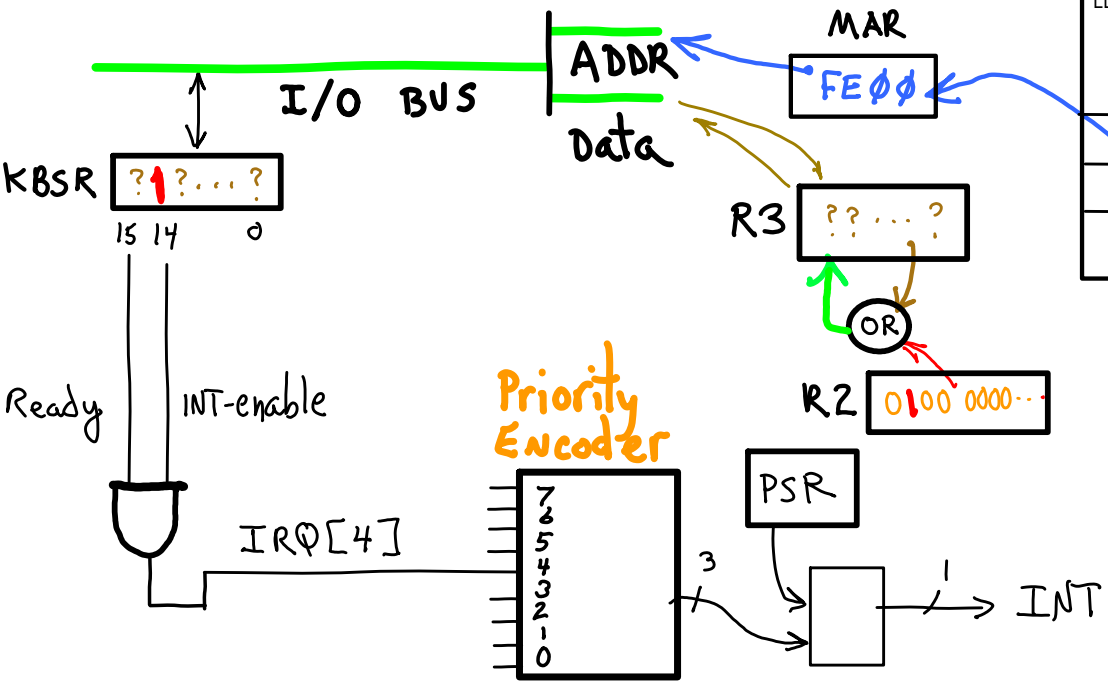
```
KB_setup :    ;---- init the KB interrupt handler

                        ;------------ set up VT for kb
LDR R0, R4, #(KB_VT)    ;--  R0 <== pointer to VT slot.
LDR R1, R4, #(KB_int)   ;--  R1 <== pointer to KB handler
STR R1, R0, #0          ;-- Mem[R0] <== R1
 ...                    ;-- ...
JMP R7                  ;-- return from KB_setup

KB_int :   ;----------------- KB interrupt handler ----

 push__( R7)   ;--- save registers
  ...          ;-- handle KB interrupt: get char, ...
 pop__( R7)    ;--- restore registers
 RTI

;--------- Enable KB interrupts (turn on bit 14)

 LDR R0, R4, #(KBSR)     ;-- R0 <== pointer to KBSR.
 LDR R3, R0, #0          ;-- R3 <== content of KBSR.
 LDR R2, R4, #(ENmask) ;-- R4 <== enable mask.
 ... ?                   ;-- R2 <== OR( R3, R4)
 STR R2, R0, #(KBSR)     ;-- KBSR <== R2
 ...

 Data_Table:
 ...
 KBSR:      .FILL xFE00    ;-- address of KBSR
 ENmask:    .FILL x4000    ;-- bit 14 is 1
```



See symbol table for value of "KB_int", it's an address.

f.asm → lc3as → f.out
f.sym (symbol table)





R3 gets KBSR's data:
R0 <== Mem[**KBSR**]
R3 <== Mem[R0]

R4 gets bit-14 mask:
R4 <== Mem[**ENmask**]

Turn on bit 14:
R3 <== R3 OR R4

R3 data overwites KBSR data

Pre-processing

OR(R1, R2, R3) in LC3 ISA

$R1 \leftarrow x + y = \overline{\overline{x + y}} = \overline{\overline{x} \cdot \overline{y}}$ De Morgan's Law

```
R2 ← x̄      NOT R2, R2
R3 ← ȳ      NOT R3, R3
R1 ← x̄ · ȳ  AND R1, R2, R3
R1 ← ¬(x̄ · ȳ)  NOT R1, R1
```

restore args? → NOT R2, R2 / NOT R3, R3

Simple substitution

SP__ — don't confuse with other strings: SPELL → R6ELL

```
.ORIG x0200
 ADD SP__, SP__, #-1
.END
```

```
cat foo.asm | sed -e ' s/SP__/R6/ ' > foo_pre.asm
```

```
.ORIG x0200
ADD R6, SP__, #-1
.END
```

```
cat foo.asm | sed -e ' s/SP__/R6/g ' > foo_pre.asm
```

```
.ORIG x0200
ADD R6, R6, #-1
.END
```

Arguments

```
cat foo.asm | sed -f  sed_defs > foo_pre.asm
```

s/SP__/R6/g

```
.ORIG x0200
push__(R4)
.END
```

```
cat m_defs foo.asm | m4 > foo_pre.asm
```

back quote

```
define(`push__', `
    add R6, R6, -1
    str $1, R6, 0')dnl
```

```
.ORIG x0200
    add R6, R6, -1
    str R4, R6, 0
.END
```

Pre-processing for data offsets in global data table

```
LDR R0, R4, #1
LDR R1, R4, #4

...

Data_Table:
 .FILL x0000    ;-- VAR foo__        (0)
 .FILL xFE00    ;-- PTR  KBSR__      (1)
 .FILL x5678    ;-- PTR  bar__       (3)
 .FILL x9ABC    ;-- PTR  fubar__     (4)
 .FILL x8000    ;-- VAL  Mask15__    (5)
 .FILL x4000    ;-- VAL  Mask14__    (6)
```

what are these? what do they refer to?

Easier to read?

```
LDR R0, GDP__,  #KBSR__ )
LDR R1, GDP__,  #Mask15__ )
```

```
s/GDP__/R4/g
s/foo__/0/g
s/KBSR__/1/g
s/bar__/2/g
s/fubar__/3/g
s/Mask15__/4/g
s/Mask14__/5/g
```

← sed command file

OR

m4 command file

```
define(`GDP__', `R4')dnl
define(`foo__', `0')dnl
   ...
```

JSR, TRAP?

m4 does substition for args

```
JSR__( KB_setup__ , R0, L1 )

   ...

Data_Table:
  .FILL x0000        ;-- VAR foo__          (0)
  .FILL x5678        ;-- VAL bar__          (1)
  .FILL KB_setup     ;-- PTR KB_setup__     (2)
```

foo.asm

```
JSR__( 2 , R0, L1 )
```

```
LDR R0, R4,  #2
LEA R7, L1
JMP R0
L1:
```

foo_pre.asm

cat m foo.asm | m4 > foo_pre.asm

```
define(`GDP__', `R4')dnl
define(`foo__', `0')dnl
define(`bar__', `1')dnl
define(`KB_setup__', `2')dnl
define(`JSR__',
    LDR $2, GDP__, #$1
    LEA R7, $3
    JMP $2
    $3:')dnl
```

m

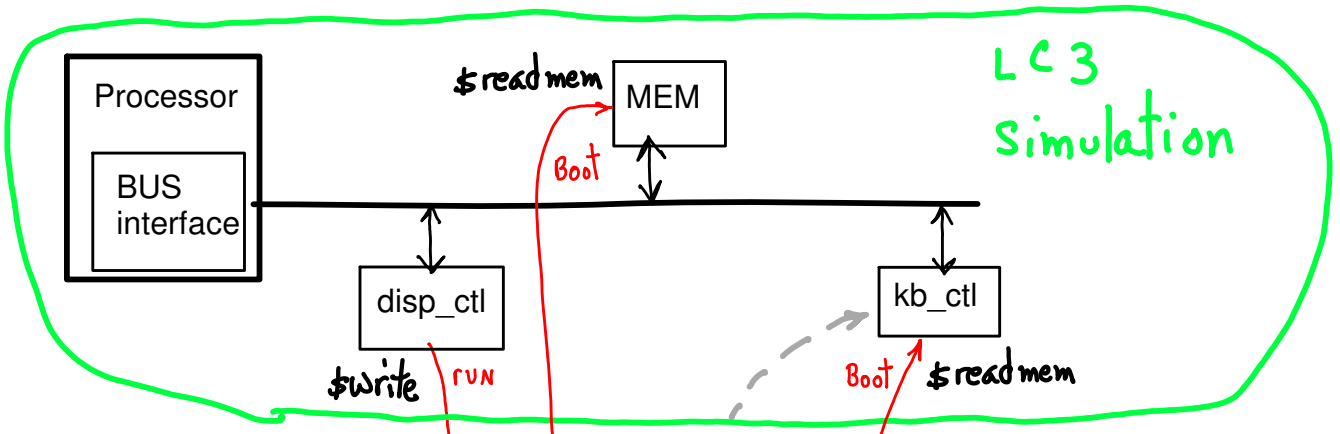
```
TRAP__( x25__ , R0, L2)

Data_Table:
  .FILL x0000  ;-- VAR   foo__    (0)
  .FILL x0025  ;-- VAL   x25__    (1)
```

```
define(`x25__', `1')dnl
define(`TRAP__',
    LDR $2, GDP__, #$1
    LDR $2, $2, #0
    LEA R7, $3
    JMP $2
    $3:')dnl
```

```
LDR R0, R4, #1
LDR R0, R0, #0
LEA R7, L2
JMP R0
L2:
```

Environment

Simulated device controllers could communicate with host system's physical devices through host OS's file system and disk. Execute a second process, "kb.c".



stdin, stdout are **buffered** in host OS.

stdout buffer not flushed for individual char output **until eoln.**

Delays seeing output from LC3.

Have to **send** "/n" **w/ each char**? Does not look good. Oh well.

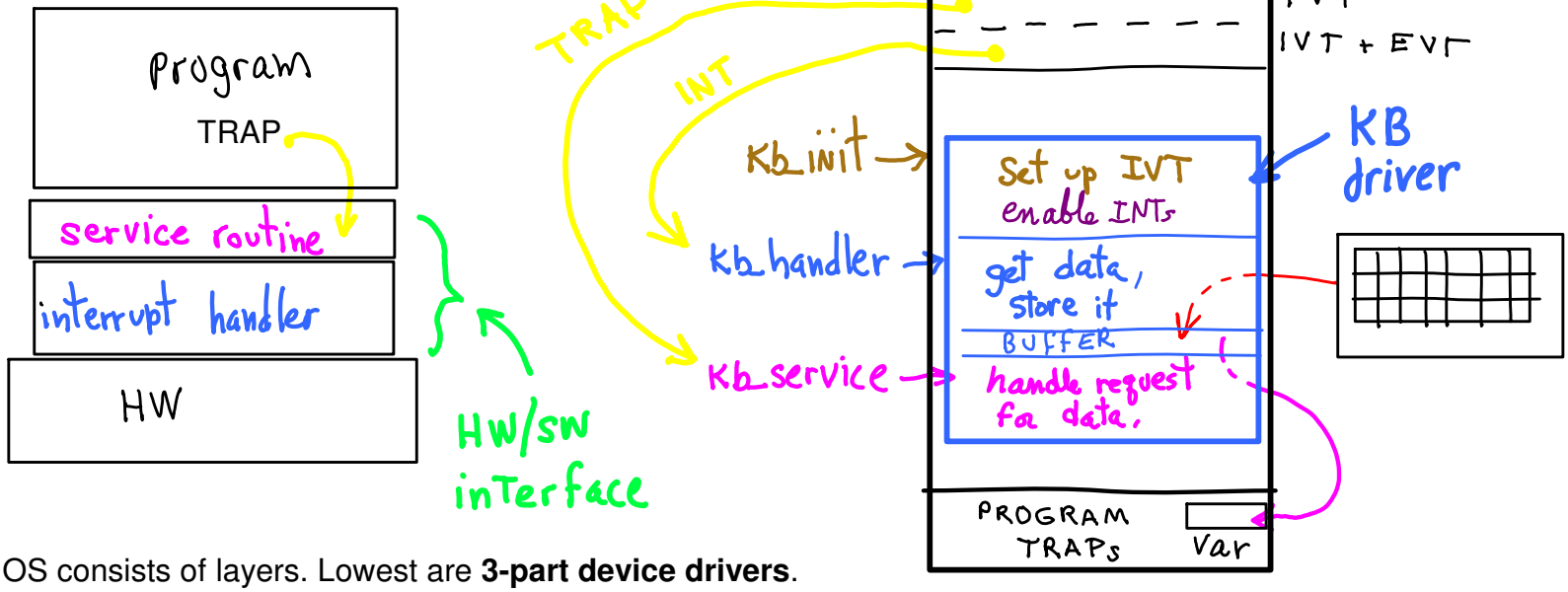
kb_ctl (faking keyboard input):

---- **INTs** occur with **fixed delays**.
---- **char data** comes **from array** of ASCII codes:

[x21, x22, x23, ... ,] == [; # $ ...]

--- **readmemb() initializes array at boot**.

LC3 OS structure, Low



OS consists of layers. Lowest are **3-part device drivers**.
(1) **init**, set IVT addresses, etc. (2) **interrupt handler** responds to HW device, buffers data (3) **service routine** handles requests from higher-level software (OS or user) to send data to program's memory area.

(using m4 pre-process: m4 macro def's cat here)

```
    .ORIG x0200

    GDP_init__(DATA)
    GO_TO_ENTRY__( CODE )

DATA:
    .FILL x3000      define(`STACK_addr__',   `0')dnl
    .FILL x0021      define(`PUTC_vect__      `1')dnl
    .FILL PUTC_init  define(`PUTC_init__',    `2')dnl
    .FILL PUTC_srvc  define(`PUTC_srvc__',    `3')dnl

CODE:
    LDR SP__, GDP__, #STACK_addr__
    LDR BP__, GDP__, #STACK_addr__

    JSR__( PUTC_init__, R0, L1)
    JSR__( GETC_init__, R0, L2)

    TRAP__( PUTC_srvc__ , R0, L3)
    ...

PUTC_init:
    JMP R7
PUTC_srvc:
    JMP R7
```

setup GDP

DATA SEGMENT

define offset w/ data

CODE SEGMENT

set up STACK

set up IVT

use a service

Turn on INTs

Go To Main Loop

A service

init:
handler:
service:

device drivers
print services
network services
...

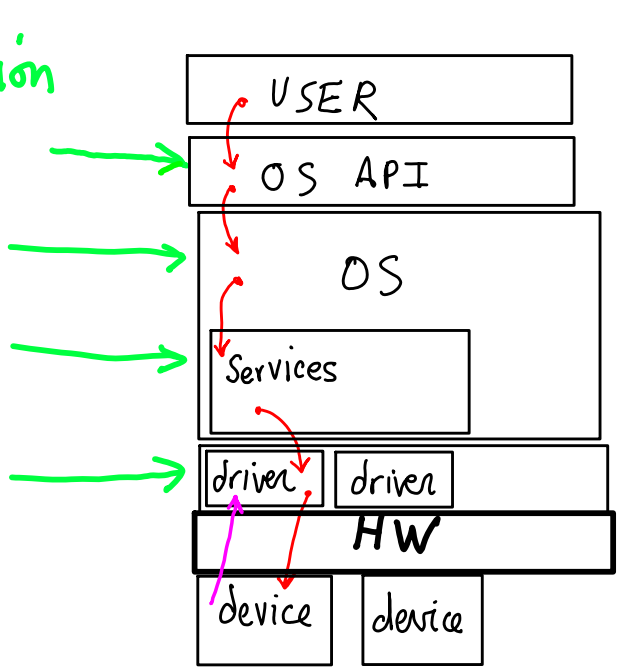
Hierarchical OS

Low-level:
- basic abstract interface
  - -- read, write

Mid-level
- built on low-level abstract structures
  - -- pages, disk blocks
  - -- buffers, connections
  - -- data structures
  - -- queuing

Higher-level
- -- files, documents
- -- pipes, streams
- -- objects, remote/local
- -- RPC, RMI
- -- policies, scheduling

abstraction Layers



load m4 defs and process w/ m4

```
cat m4-defs foo.asm | m4 > foo_pre.asm
```

m4 defs

```
changecom(`;')

define(`GO_TO_ENTRY__', `  ;-- GO_TO_ENTRY__( CODE )
   LEA R7, $1_PTR          ;--   LEA R7, CODE_PTR       ;-- R7 points.
   LDR R7, R7, #0          ;--   LDR R7, R7, #0         ;-- R7 gets addr.
   JMP R7                  ;--   JMP R7                 ;-- go to CODE.
   $1_PTR: .FILL $1')dnl   ;--   CODE_PTR: .FILL CODE   ;-- data.

define(`GDP__',`R4')dnl    ;-- GDP__ == R4
define(`BP__' ,`R5')dnl    ;-- BP__  == R5
define(`SP__' ,`R6')dnl    ;-- SP__  == R6

define(`GDP_init__', `     ;-- GDP_init__( Table )
   LEA GDP__, $1_PTR       ;--   LEA GDP__, Table_PTR   ;-- GDP points.
   LDR GDP__, GDP__, #0    ;--   LDR GDP__, GDP__, #0   ;-- GDP gets addr.
   BR  init_GDP_done       ;--   BR  init_GDP_done      ;-- jump over,
   $1_PTR: .FILL $1        ;--   Table_PTR: .FILL Table ;--      data,
   init_GDP_done: ')dnl    ;--   init_GDP_done:         ;--         to here.

define(`JSR__', `          ;-- JSR__( func__, R0, L1 )
   LDR $2, GDP__, #$1      ;--   LDR R0, GDP__, #func__   ;-- R0 gets addr
   LEA R7, $3              ;--   LEA R7, L1               ;-- linkage
   JMP $2                  ;--   JMP R0                   ;-- call
   $3:')dnl                ;--   L1:                      ;-- return here

define(`TRAP__', `         ;-- TRAP__(vect__, R0, L2)
   LDR $2, GDP__, #$1      ;--   LDR R0, GDP__, #vect__   ;-- R0 aimed at VT
   LDR $2, $2, #0          ;--   LDR R0, R0, #0           ;-- R0 gets addr
   LEA R7, $3              ;--   LEA R7, L2               ;-- linkage
   JMP $2                  ;--   JMP R0                   ;-- call
   $3:')dnl                      L2:                      ;-- return here

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; m4-test.asm
;;
;; test our m4-commands as a preprocessor.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   .ORIG x0200

   GDP_init__(DATA)
   GO_TO_ENTRY__( CODE )

DATA:
   .FILL x3000     define(`STACK_addr__',       `0')dnl
   .FILL x0021     define(`PUTC_vect__',        `1')dnl
   .FILL PUTC_init define(`PUTC_init__',        `2')dnl

CODE:
   LDR SP__, GDP__, #STACK_addr__
   LDR BP__, GDP__, #STACK_addr__

   JSR__( KB_setup__, R0, L2)
   TRAP__(x25__ , R0, L1)

KB_setup:
   JMP R7
```

foo.asm starts here

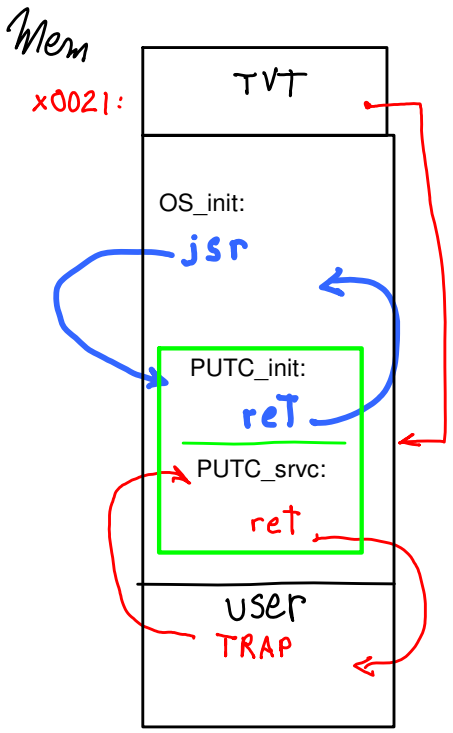
Aside: stripping out blank space after pre-processing.
cat m4-def

OS Layers

Modular development

Module consist of two parts:
--- PUTC_init
Called (JSR) by OS init.
Return via "RET", "JMP R7".
--- PUTC_srvc
Called via TRAP
Returns via RET.

```
;-------------------------------------
;-- putc.asm, a module
;-------------------------------------
  ;================================================
  ;-- .CODE
  ;------------------------------------------------
  PUTC_init:
    ldr r0, GDP__, #PUTC_vect ;-- r0 <== slot addr
    ldr r1, GDP__, #PUTC_srvc ;-- r1 <== svc addr
    str r1, r0, #0            ;-- IVT[r0] <== svc addr
    ret

  ;;;===============================================
  ;;;-- PUTC_srvc() - trap x21:  R0 == char
  ;------------------------------------------------
  PUTC_srvc: ;-----------------------------------
    putc_POLL:              ;-- do
      ...                   ;--    status <== DSR
      BRzp putc_POLL        ;-- until(status == READY)
    ...                     ;-- DDR <== char (print char)
  ret
  ;------------------------------------------------
  ;;;===============================================
  ;;;-- .DATA
  ;------------------------------------------------
  DSR:              .FILL xFE04   ;-- display status
  DDR:              .FILL xFE06   ;-- display data
  DSR_READY_MASK:   .FILL x8000   ;-- DSR[15]=1?
```

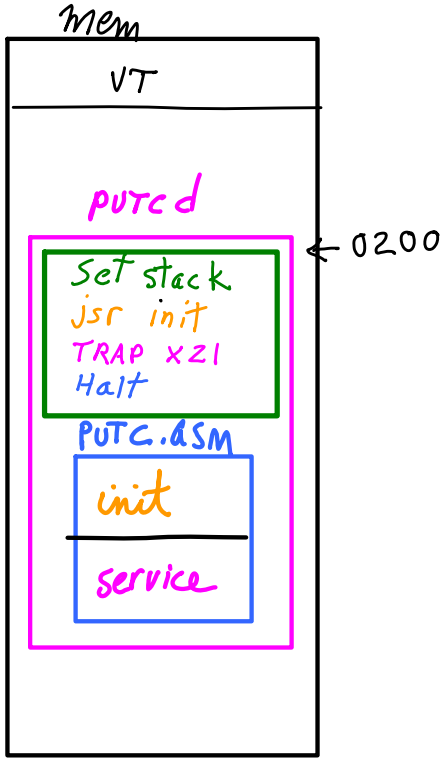


module is linked in To OS
- via source code, or
- via link editing

wouldn't it be nice if...

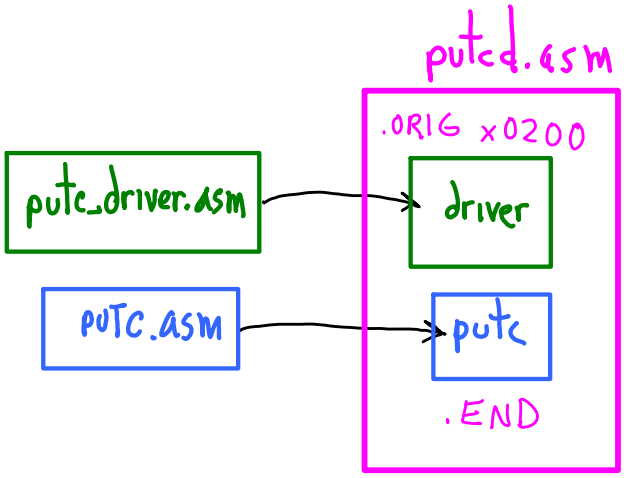
TESTING, independent of OS code

```
;-------------------------------------
;-- putc_driver.asm
;-------------------------------------
  ;-- set GDP
  ;-- Set stack.
  ;-- init putc
  ;-- arg <== MY_CHAR
  ;-- trap( arg )
  done:
  ;-- R4  <== STOP_CLOCK
  ;-- MCR <== R4 (halts LC3)
  ;;--- .DATA -------------------------------
  STOP_CLOCK:    .FILL x8000  ;-- MCR[15]=1
  MCR:           .FILL xFFFE  ;-- mach. ctl reg
  MY_CHAR:       .FILL x0041  ;-- ASCII 'A'
```



Build & Run

1. edit



2. Pre-process, assemble, convert, copy to ../run

has both sed and m4 defs (?)

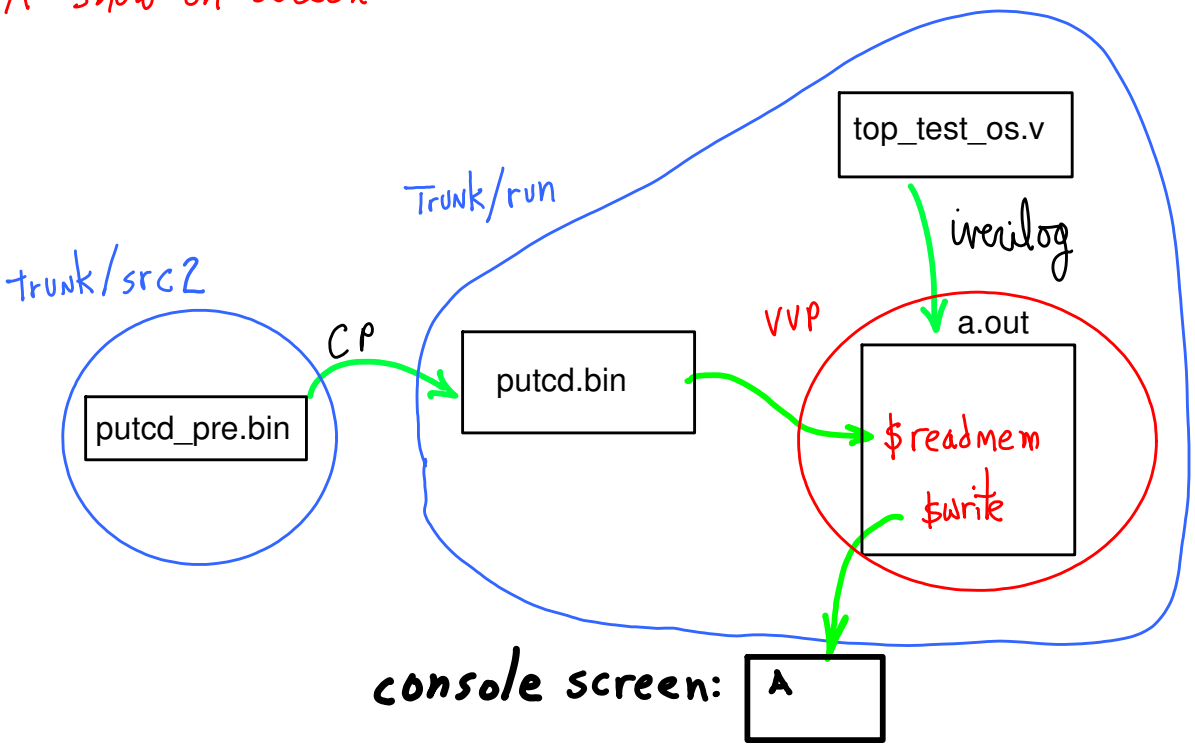
```
% make os.bin OS=putcd
cat m4-defs putcd.asm | m4 > putcd_pre.asm
lc3as putcd_pre.asm
cat putcd_pre.obj | obj2bin | sed '1d' > putcd_pre.bin
cp putcd_pre.bin ../run/putcd.bin
```

```
cat putcd.asm | lc3pre > putcd_pre.asm
```

if using PennSim, stop here

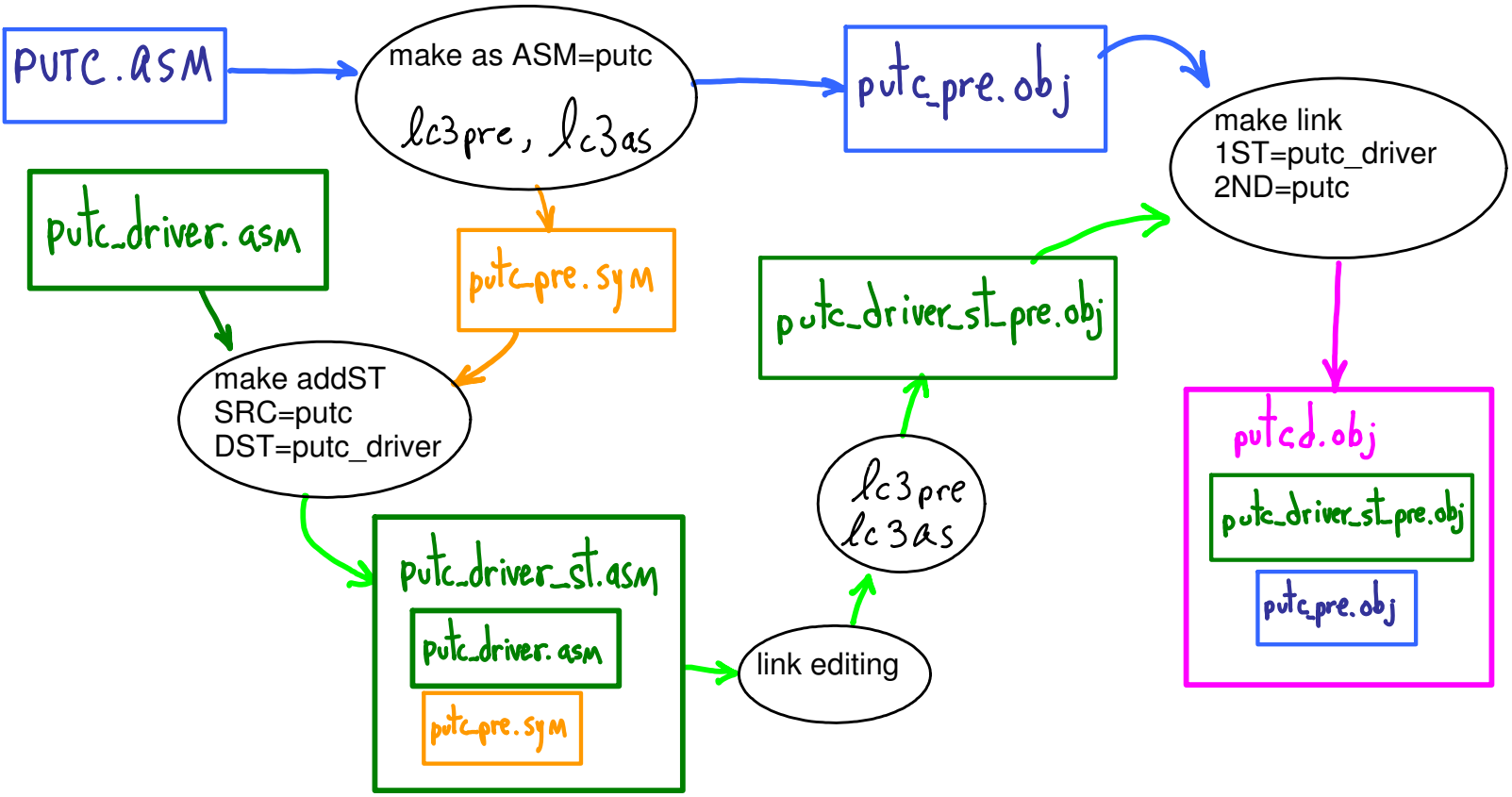
3. Run does 'A' show on screen?

```
cd ../run
vvp a.out
```



Aside: link editor ?

First Try:
Edit .asm source code. Concatenate .obj files.
Need symbol table to adjust addresses.



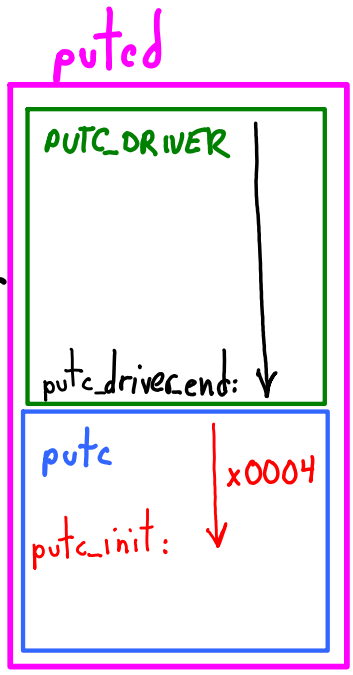

```
;-------------------------------------
;-- putc_driver.asm
;-------------------------------------
        .ORIG x0200
      ...
   LDR R1, GDP, #driver_end__
   LDR R2, GDP, #putc_init___ )
   ADD R2, R2, R2
   JSRR R2
    ...
  .FILL 0     ;-- putc_init
  .FILL driver_end
driver_end:
        .END
// putc.asm Symbol table
// Scope level 0:
//     Symbol Name        Page Address
//     ----------------   ------------
//     putc_init          0004
//     putc_init_END      0007
//     putc_svc           0007
//     putc_POLL          000B
//     end_putc_POLL      000F
//     putc_END           0012
//     PUTC_TVT_LOC       0018
//     DSR                0019
//     DDR                001A
//     DSR_READY_MASK     001B
//     END                001C
```

jump To putc_init

run Time

link edit Time

copy



putc_driver_end: ==
size_of(**putc_driver**)

x0004 ==
offset to putc_init
relative to start of **putc**

putc_driver_end + x0004 ==
address of **putc_init**

EDITING: **Copy** the value of

putc_init

from **symbol table** to

.FILL 0

---- lables need to be identified
.external putc_init

C? assumes

---- handle data references similarly

---- next step: **edit .obj files** instead of .asm
need each .obj to **include its symbol table**
need table to identify **where to edit**

Advantages of linking

1. Code kept **separate, independent**

2. **Test** module **once, reuse** as tested component

3. Compile/Assemble separately, **smaller builds** (Make, avoid re-compile/re-assemble

Link methods

1. **by hand**: include **all source code** into one unit, then **hand edit** to fix.

2. **linker**: **write an .obj linker**

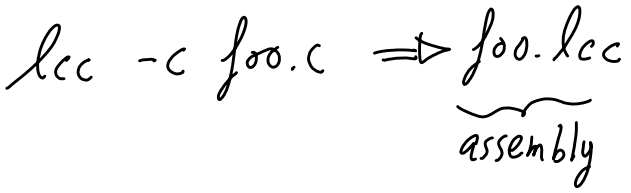
3. **C compiler**: it **includes a linker** (and a pre-processor)

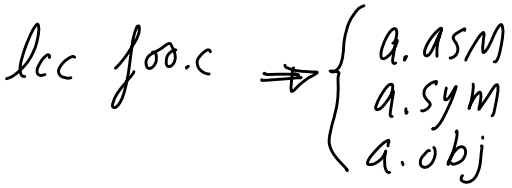
    Write LC3 assembly (main.asm, foo.asm)
    ---- Use lcc C conventions
    ---- rename to match lcc (main.lcc, foo.lcc)
    ---- let lcc do the linking for you:

        lcc main.lcc foo.lcc

.OBJ linking Advantages:
--- separate module development: **name independence**
--- **partial builds** w/ linking (what Make is really for)
--- easy to **link C w/ assembly**
--- **stack discipline:** **reentrant code, multiple threads**

(see trunk/src/lcc_annotate)

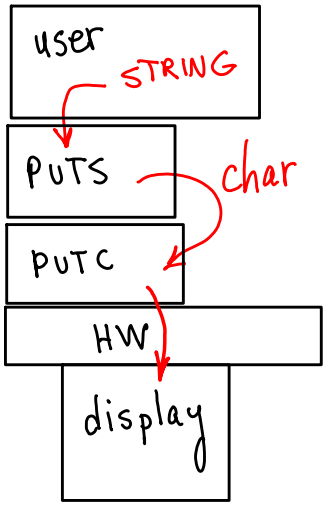




OS, 2nd layer user service

2nd layer service
uses 1st layer abstraction.

print a string to display.

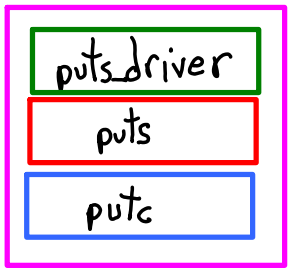



```
;-----------------------------------
;-- puts.asm
;-----------------------------------
;.CODE
  ;;;=============================================
  ;;;-- puts_init
  ;-----------------------------------------------
     ...

  ;;;=============================================
  ;;;-- puts- trap x22:
  ;;;-- Display string, R0 == address
  ;-----------------------------------------------
  puts_BEGIN: ;-------------------------------------
    push__( R7 )

    mov__( R1, R0 )                ;-- R1 <== char_ptr
    LDR R0, R1, 0                  ;-- char <== *char_ptr

    while_nonNUL:                  ;-- begin_while
      BRz end_puts_while           ;-- until( char == NUL )
      TRAP x21                     ;--   call putc
      ADD R1, R1, 1                ;--   char_ptr++
      LDR R0, R1, 0                ;--   char <== *char_ptr
      BR while_nonNUL              ;--
    end_puts_while:                ;-- end_while

    pop__( R7 )
    JMP R7                         ;-- RET
```

Save R7

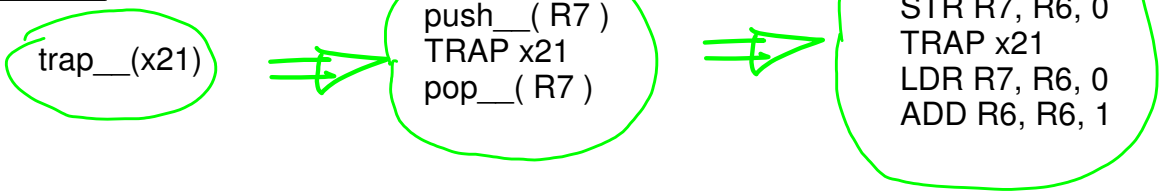
Test same as putc

Link

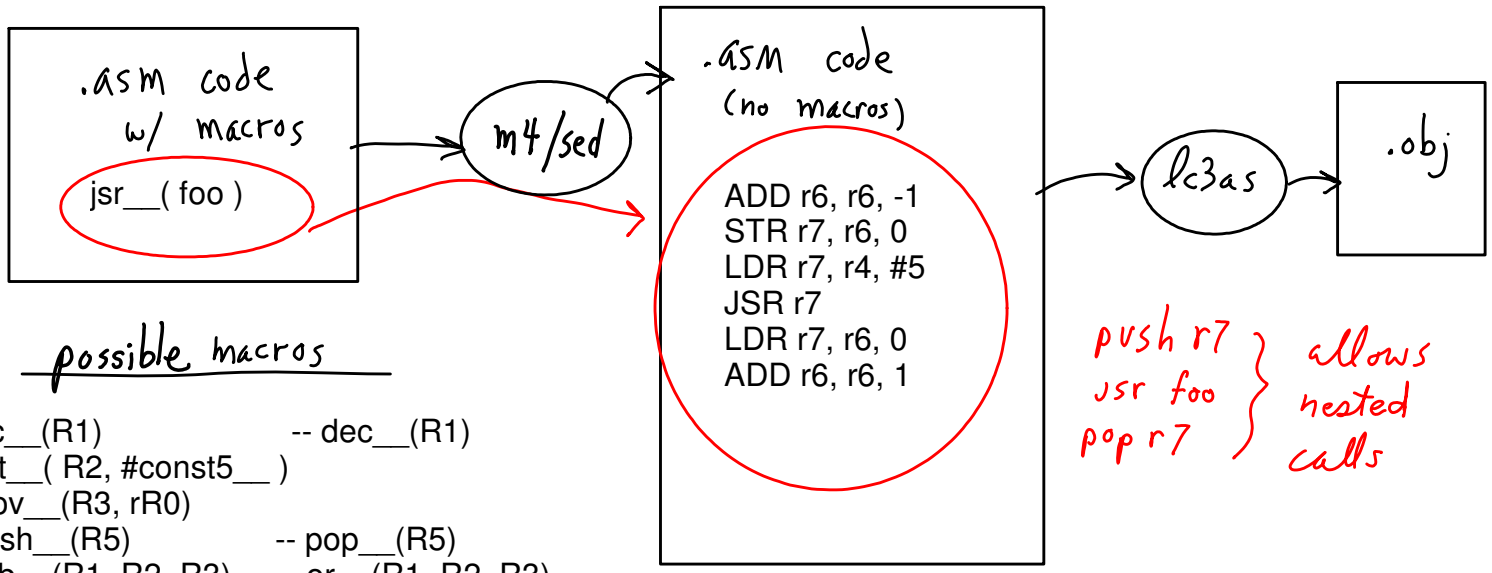


Nested Calls

Saving R7



In general, may need to save ALL registers.
===> CALLEE SAVE (called code save registers)
===> CALLER SAVE (code that makes the call saves its own registers



push r7
jsr foo
pop r7
} allows nested calls

possible macros

```
-- inc__(R1)                  -- dec__(R1)
-- set__( R2, #const5__ )
-- mov__(R3, rR0)
-- push__(R5)                 -- pop__(R5)
-- sub__(R1, R2, R3)          -- or__(R1, R2, R3)
-- jsr__( #mySub__)           -- trap__(x13)
-- getc__                     -- putc__
-- halt__                     -- puts__
-- intsOff__                  intsOn__
```

aka, pseudo-instructions

Provides mechanism for nested calls/recursion

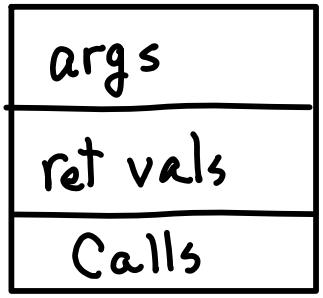
All macros have "__" double underscore

Condition codes are set as expected

Argument registers are unchanged, only destination changes

example	translation
mov__(R3, R5)	ADD R3, R5, 0
zero__(R2)	AND R2, R2, 0
inc__(R7)	ADD R7, R7, 1
push__(R5)	dec__(SP__) STR R5, SP__, 0
pop__(R5)	LDR R5, SP__, 0 inc__(SP__) mov__(R5, R5) ;;-- sets NZP CCs
sub__(R1, R2, R3)	not__(R3) inc__(R3) ADD R1, R2, R3 dec__(R3) not__(R3) ;;-- R2, R3 unchanged, mov__(r1, r1) ;;-- sets NZP CCs

OS design



Establish Uniform Conventions
Program interface to OS

--- Where are args, return values, return addresses?
  --- Stack usage: push all to stack?
  --- R0 points to data struct, R7 has return address?

--- General abstract device interface:
  --- read, write, append, ...
--- Mode changes, OS module independence
  --- user/supervisor swap, messages to modules

higher-level services

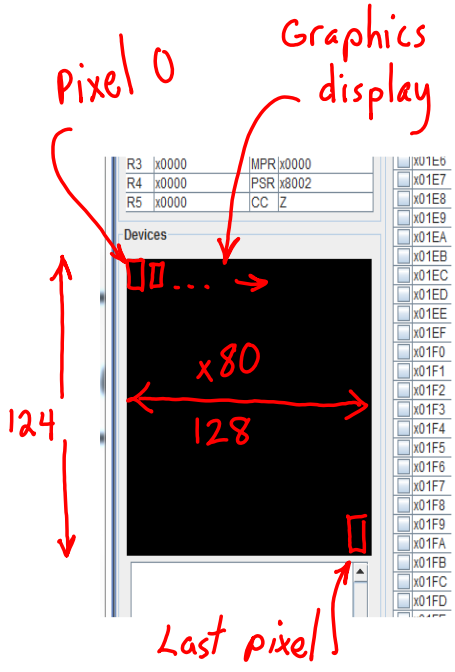
- Clear screen ⇒
  - fill display w/ spaces
  - how many?
  - what's screen size?
- home cursor ⇒
  - add position counter to putc
  - rewrite entire screen, need buffer
- sub-windows ⇒
  - define virtual screen objects, hierarchy
  - define "current window"
- Processes/owners {
  - mechanisms for switching, user input

Graphics I/O

16-bit word refers to one pixel,
Each pixel has three intensities R, G, B:

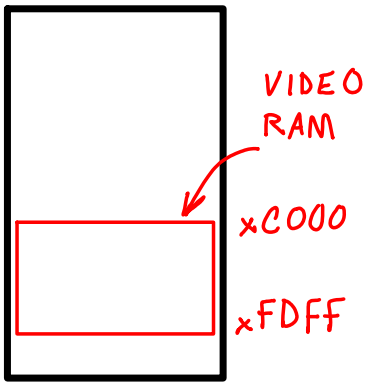
0 R R R R R G G G G G B B B B B

5-bit Red 5-bit Green 5-bit Blue



PennSim

LC3 Mem



0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0

7 C

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 4 0 0 inc. Red

0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0

3 E

0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 2 0 inc. Green

0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1

1 F

add R+G → pixe(R0)

```
.FILL x7FFF    ;---- White
.FILL x0000    ;---- Black
.FILL x7C00    ;---- Red
.FILL x03E0    ;---- Green
.FILL x001F    ;---- Blue
.FILL x0_0_1_0_0_0_0_1_0_0_0_0_1_0_0_0  ;---- Gray
```

macros?

```
LDR R1, GDP, #Red__
LDR R2, GDP, #Green__
ADD R1, R1, R2
STR R1, R0, #0    ;---- R0 points to pixel
```

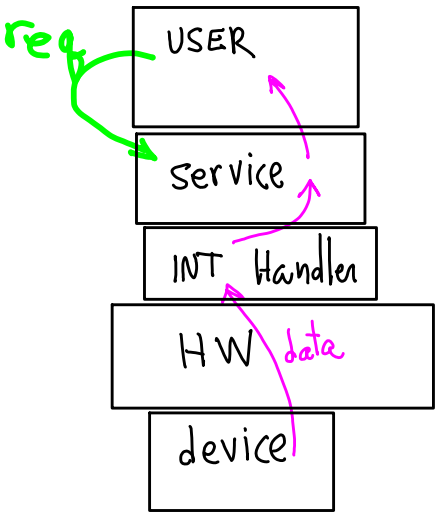
```
.FILL xC000   ;;------ VRAM__
.FILL x0080   ;;------ VRAM_row_inc__

LDR R0, GDP__, #VRAM__
LDR R1, GDP__, #VRAM_row_inc__
LDR R2, GDP__, Green__

;;========== Draw diagonal green line
Loop:

        inc( R0 )                ;;---- move to next pixel (right)
        ADD R0, R0, R1           ;;---- move down one row
        STR R2, GDP__, #0        ;;---- write green pixel
```

Provide traps for

- low-level primitives
- high-level ops (e.g. clear)
- objects
  - windows, ...
- pseudo-mouse?
- selection?

Interrupt-driven I/O

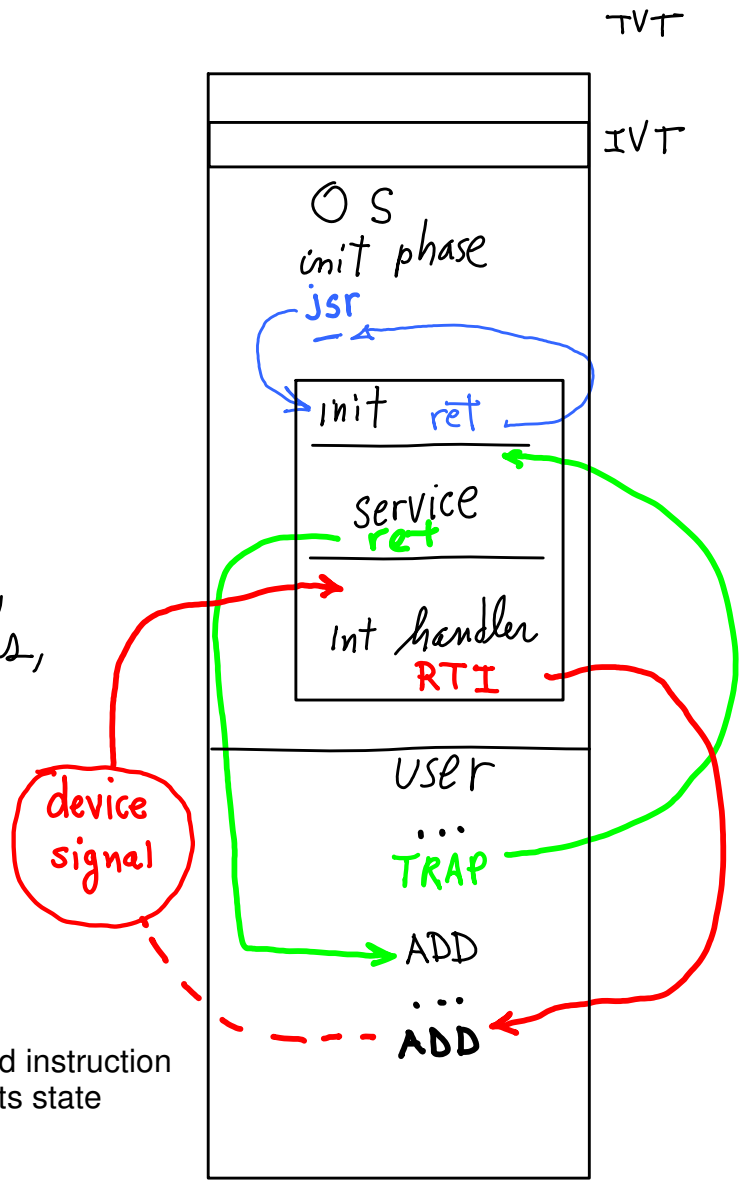
OS layer w/ 3 parts



— user requests.

— service checks status, waits.

— handler delivers, changes status, wakes up service



Interrupts any executing code,
-- might not be requester,
-- another trap or interrupt handler?

Must restart interrupted instruction
-- w/o state changing its state

-- Both handler and user transistion to-from coma state when interrupt occurs.

-- Is either totally or partially aware of state change of other?

-- If state is completely restored, how do handler and service communicate?

Keyboard interrupt handler

In Hardware: push(PSR, PC), swap stacks, PSR.Priority = 4
In handler: save regs as needed

--- save state, turn off interrupts

--- get data from keyboard, put in buffer

--- handshake kb_ctl: "got data, all done"

In kbctl: a read from KBDR causes KBSR[15] ← 0

--- change status } let world know buffer has data

--- handshake kb_ctl (enable kb interrupts) } kbctl never turns them off (but it should)

--- restore state, turn on interrupts } In handler: restore regs
HW: interrupts turned on via pop(PSR)

--- RTI (pop PC, PSR, swap stacks, change mode)

↑ depends on popped PSR[15]

↑ via pop(PSR)

==========
Keyboard service
==========

Forever()

--- check queue for requests

if none, sleep (jump to OS)

--- check status

--- if status == ready

copy data to requester
update buffer
return from call

--- else

wait: sleep (jump to OS)

INT handler changes kbsvc's state, OS restarts kbsvc

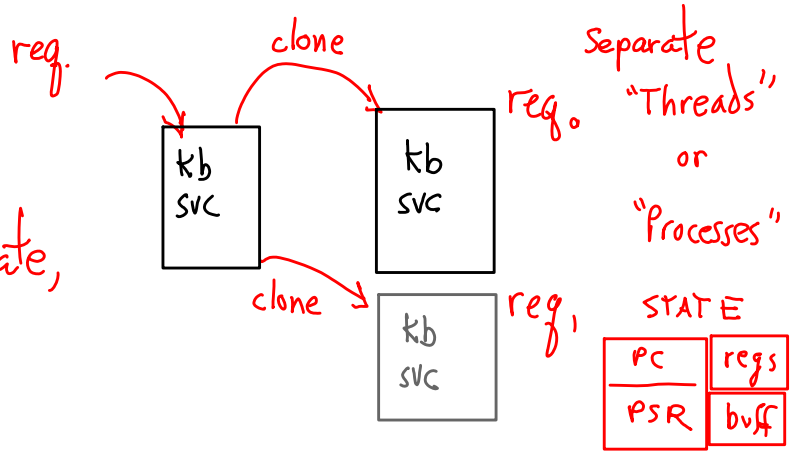
==========
Requester
==========

--- format request
set number of chars
set local buffer location

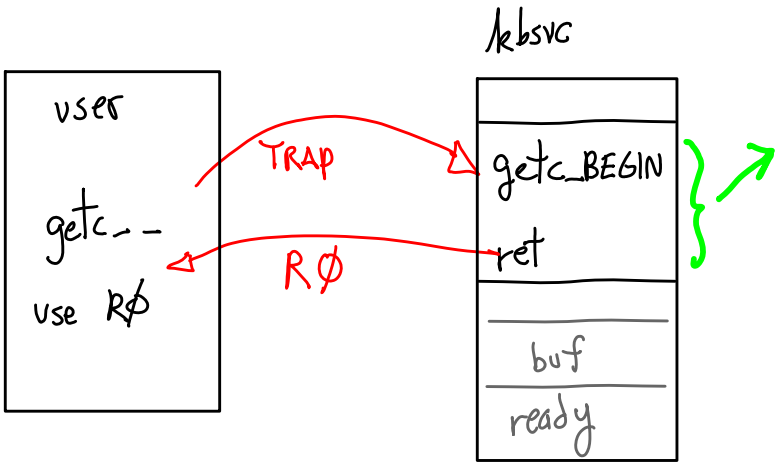
--- send request
trap to service

--- on return
use data in local buffer

multiple requestors



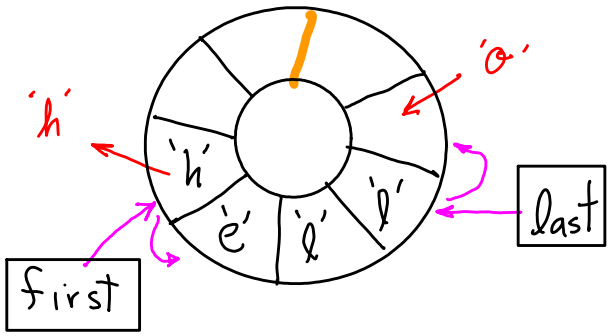
Our svc Req structure

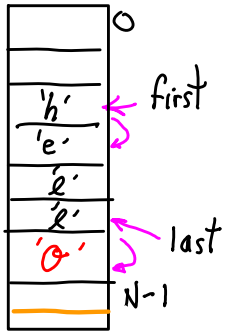


ready?
yes: R0 ← buf
buf now empty? ready ← 'no'
ret
no: wait (sleep?)

kb_int:
buf not full?
- buf ← char
- ready ← 'yes'
- wake_up (requestor)
else
(ignore input?)

Circular Buffer





put_in: last++ (%N) ; buf[last] = char

take_out: char = buf[first] ; first++ (%N)

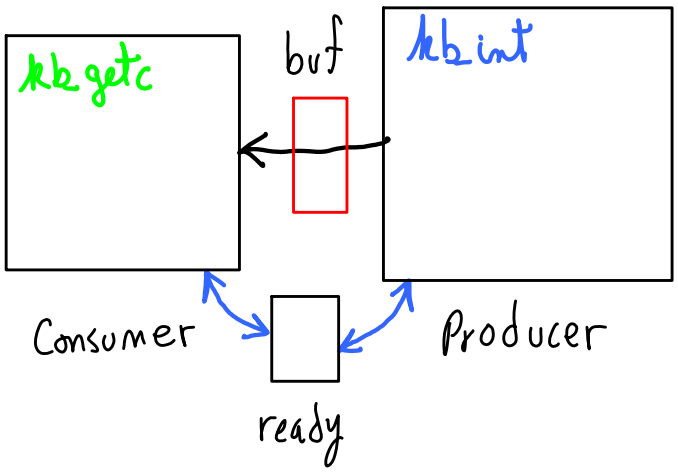
Pointers walk around buffer, inserting at "first", removing at "last"

Buffer Details.
--- What to do when empty?
--- How to avoid overflow?
if(first == last) ...

Managing shared variables.
int handler, 1 and **trap service, 2**

v: .FILL x0001
1-Reads v , R0 <== 1
(state saved, registers saved, switch)
2-Reads v , R0 <== 1
(state saved, registers saved, switch)
1-Writes v <== R0+1
(state saved, registers saved, switch)
2-Writes v <== R0+1

What's in v? "first"? "last"?



kb_getc and **kb_int** are cooperating, concurrent, asynchronous sequential processes:

--- Sharing a resource "buf";
--- Controlling access via "ready";
--- w/ Waiting (sleeping).

Concurrency

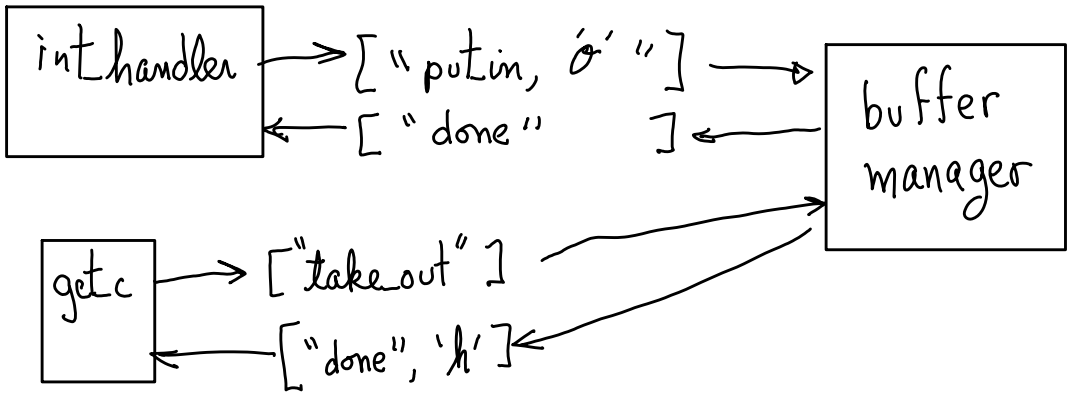
Two general structures

- locks:
  - get sole possession of shared resource
  - other waits until lock released

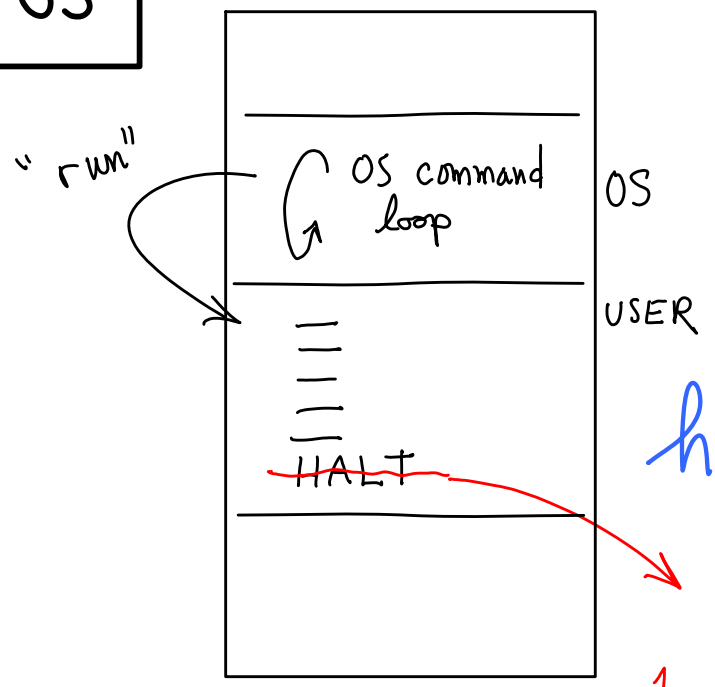
```
getc
    wait(bufLock)
        char ← takeOut()
    unlock(bufLock)
```

```
int_handler
    wait(bufLock)
        putIn(char)
    unlock(bufLock)
```

- messages:
  - wait for message, send reply
  - send message, wait for reply



Return To OS



What to do when user program finishes?

--- Act like a function exited and return to the place in the OS where the jump occurred?

--- Restart OS at a fixed location?

halt: turn off sys_clk?

NO!

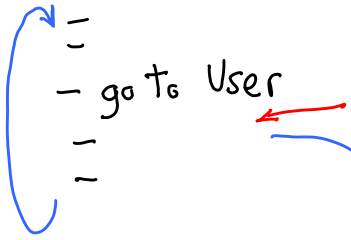
ret_to_OS

1. How, what mechanism?
2. Where, and do what?

Possibilities

1. – use ret
   - set R7 before jmp to User. Needs user code to remember to save R7. Or push return address, User must pop(R7) then RET. BUT mode change? See (3).

command loop



2. – use RTI
   - If super-mode, first need to set up stack w/ dummy PC, PSR
   - If User, will cause privilege exception (Re-enters OS w/ mode change)

3. – return_to_OS__
   - Define a new trap. Trap routine can decide where to jump to in OS. Can also clean up, reinitialize, But, can't change mode back to kernel mode?

   Play a trick: use illegal opcode exception to switch modes: set a flag before.

Jump To User Code

Things To do:

- Set return mechanism
- Set arguments, environment vars, etc.
- Set mode to user
- Set up user stack
- jmp to User code

NOTE:

We will only run kernel-mode programs. No need to

--- set user stack
--- change mode

when jumping to a kernel-mode program. But, we will load kernel programs to the user memory area.

Insert a preamble, preamble jumps to main.
User code does RET back to preamble.
Preamble code jumps back to OS.

Switching between code "entities" (concurrent execution)

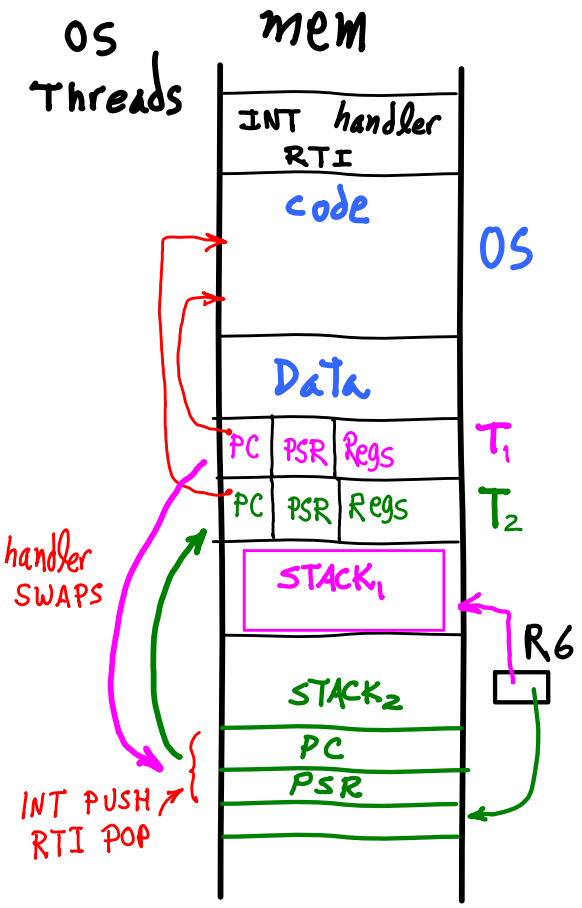
interrupt/exception
--- **saves state** of currently executing code
--- **transfers** execution **to other code**

RTI
--- **restores state**
--- **transfers** execution **back**

Waiting/Sleeping needs same mechanism.

Switch Context
--- Use a hardware (timer) interrupt (involuntary switch)
--- Use illegal opcode (voluntary switch)
interrupt/exception handler does:
--- **save current** code's **state** (**save registers**, stack)
--- **select next context (scheduler)**
--- **restore state** of code to switch to (**fill registers**, stack)
--- **push** saved **PSR**
--- **push** saved **PC**
--- **RTI (restores PSR, PC)**

NEED:
--- **List of currently "in" execution entities** (**processes/threads**)
--- for **each item** in list, **a data structure**:
  --- **PSR, PC, R0, R1, ... , R7, (+ why it is waiting)**
--- **Scheduler: select next thread to execute from list**



Linking, Link/Load objects

%> cat f.c

```
#include <stdio.h>
#define MAX_NUMS 10
   ...
int main()
{
  int index;
  int numbers[MAX_NUMS];

  printf("Enter %d numbers.\n", MAX_NUMS);
```

==> f.c uses operating system's services.

==> HALT and OUT.

==> NOT in f.c's C code!

==> **printf**()? IS THAT C TOO?

==> **printf.asm** is linked in.

cd bin/lcc-1.3/lc3lib/

ls

getchar.asm	printf.asm
putchar.asm	scanf.asm
stdio.asm	stdio.h

Usually, **linking not done on .asm files**,
===> link .obj file ("link objects").

Link objects (may be collected in **"library" files**.)
Unix: **cc -c f.c ===> f.o**
LC3: **lcc -c f.c ===> f.obj**

Linking
Unix: **cc f.o g.o bar.o ===> f.out**
LC3: **lcc f.obj g.obj bar.obj ===> f.obj**

Static Linking:

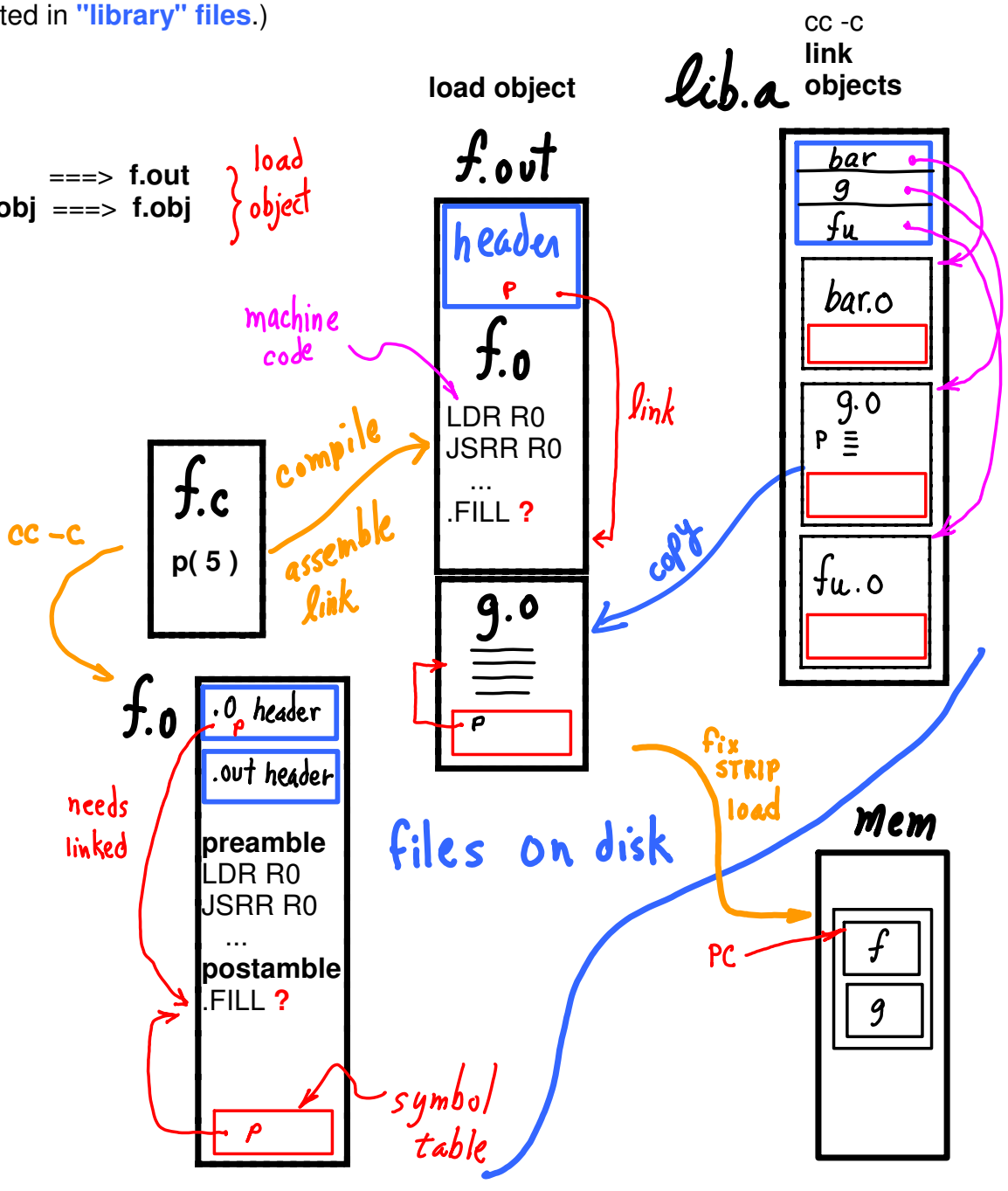
Library header provides **pointers to** sections of **code.**

Sections extracted and **copied to form one file**.

Loader:

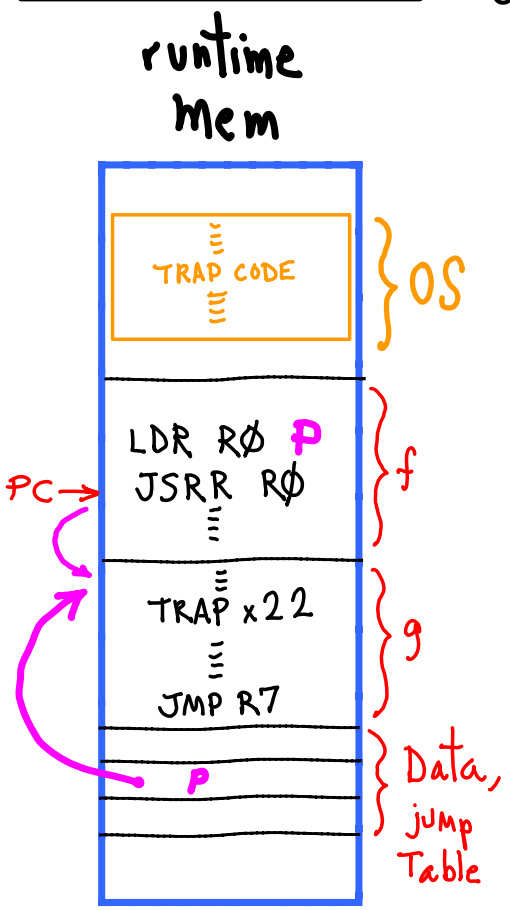
Headers stripped (.out headers), executable **code copied to memory**, along w/ preamble.

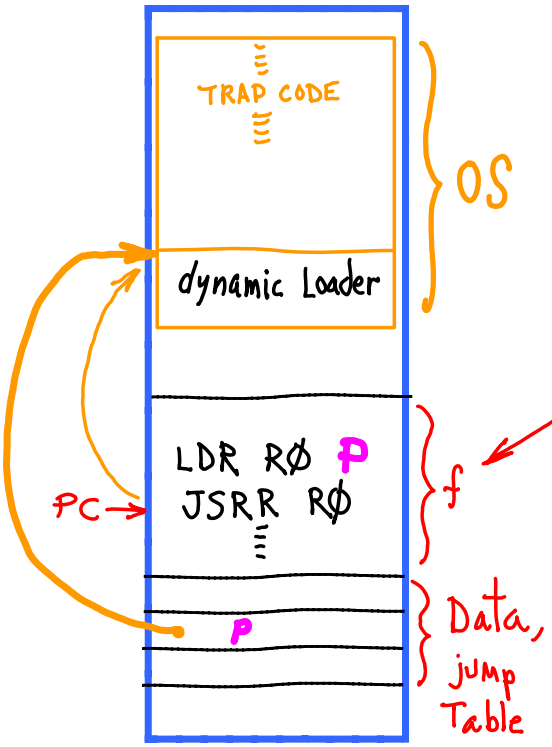
References (addresses) **fixed**
---- **at link time**
---- **at load time**



Dynamic Load

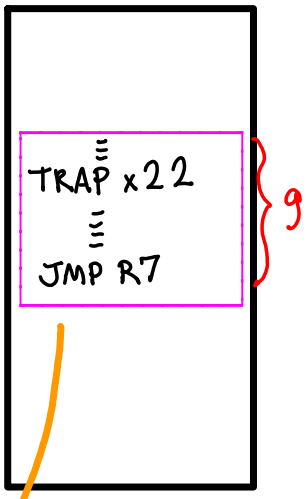
How much memory space wasted by g?
What if f never makes jump to p, completely wasted.
Can we do better?



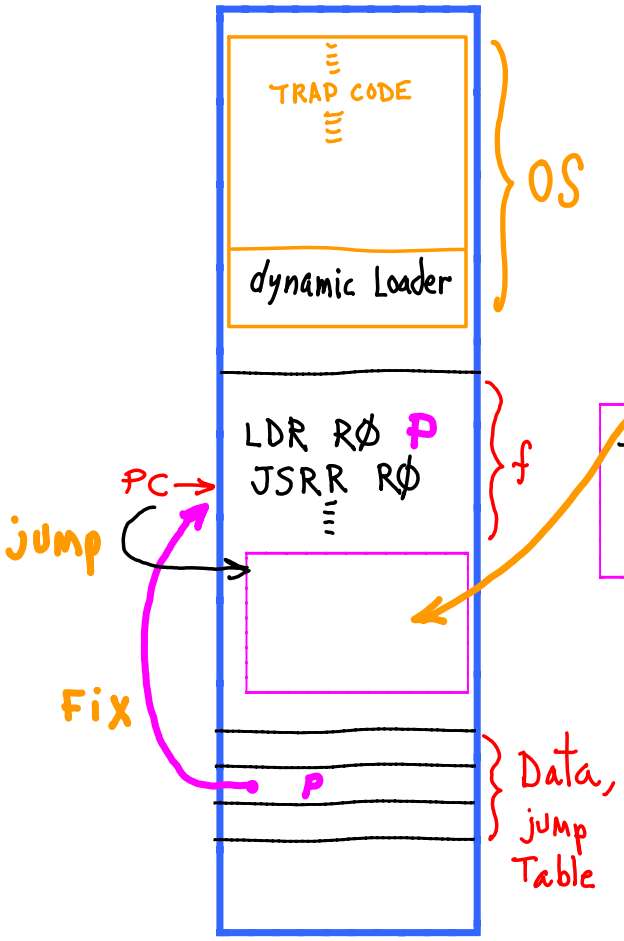


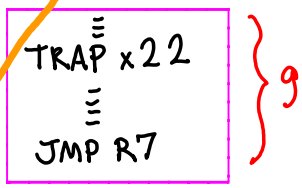
Compiled for dynamic load

lib.dll



COPY, LOAD





Contrast:

Dynamic linking (.DLL)

--- call is via a jump table

--- jump table filled in as needed at runtime

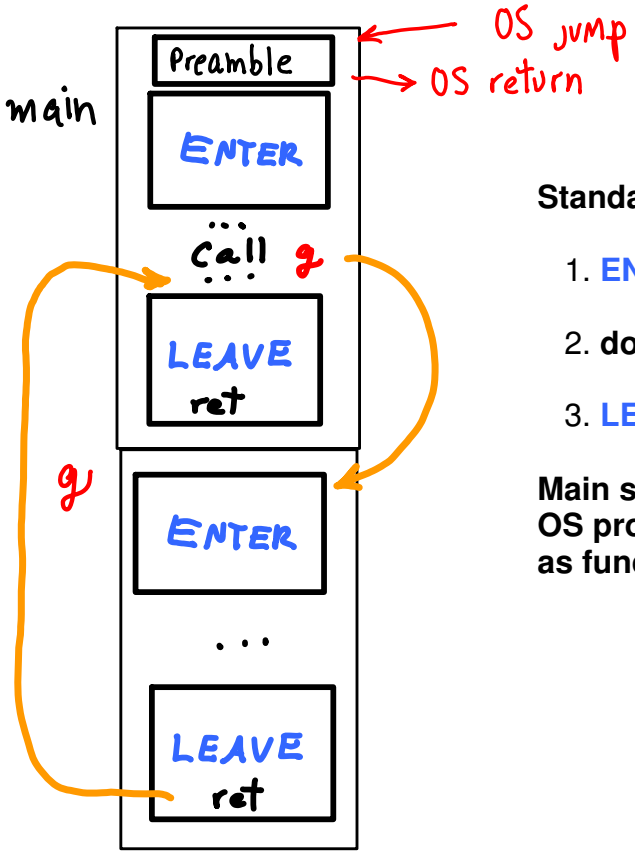
--- 1ST jump goes to loader

--- executable loaded

--- next time, jumps go to loaded executable, P

--- Pay time for first access
--- after that, same as static.

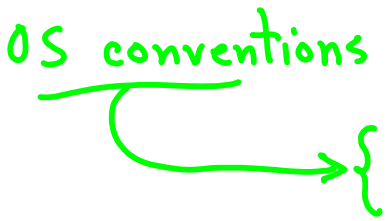
C conventions, lc3 lcc style



Standardized CALL and LEAVE protocol

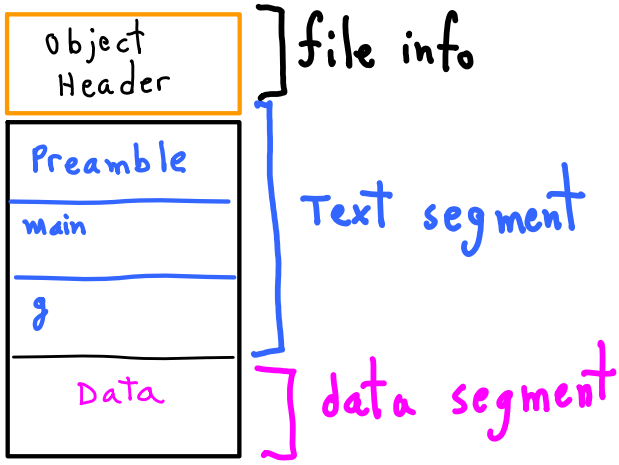
1. **ENTER: set up stack**
2. **do stuff**
3. **LEAVE: unwind stack**

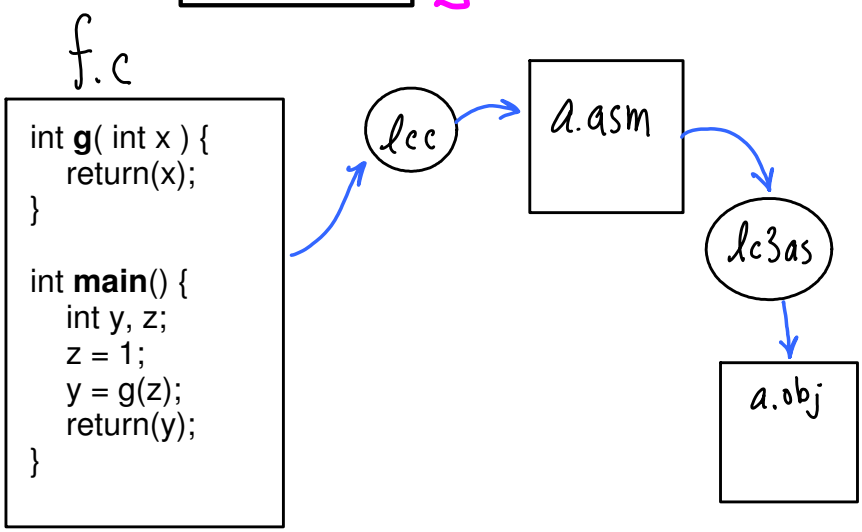
Main setup just like a function call
OS provides arguments in same way as function call.



Object structure (.o or .obj)

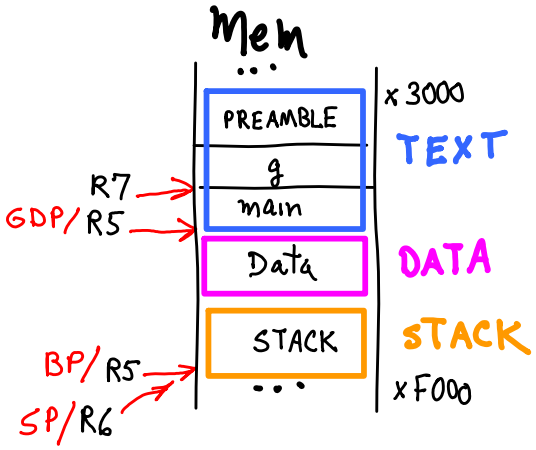
0. **Header(s)**
   Pointers, offsets, types
1. **Preamble** inserted
   Handles OS conventions
2. **Text Segment**(s)
   machine instructions
3. **Data Segment**(s)
   pointers to functions
   constants' data
   global variables
   variables' initial values



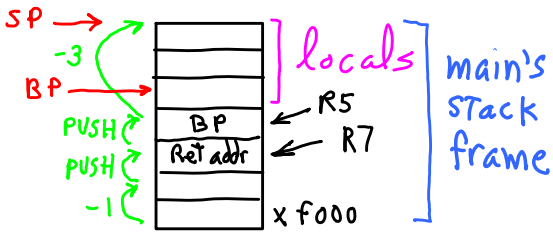


```
.Orig x3000
INIT_CODE    ;;----------------- PREAMBLE
LD R6, STACK_POINTER
LD R5, STACK_POINTER
LD R4, GLOBAL_DATA_POINTER
LD R7, GLOBAL_MAIN_POINTER
jsrr R7
HALT
GLOBAL_DATA_POINTER .FILL GLOBAL_DATA_START
GLOBAL_MAIN_POINTER .FILL main ;;-- pointer var.
STACK_POINTER .FILL xF000

;;---------------------------- TEXT SEGMENT
    ...   ( main's and g()'s text) ...

;;---------------------------- DATA SEGMENT
GLOBAL_DATA_START:
g     .FILL lc3_g  ;;-- Pointer variable to g()
L1_f  .FILL lc3_L1_f
L4_f  .FILL lc3_L4_f
L3_f  .FILL #2     ;;-- CONST 2
L5_f  .FILL #1     ;;-- CONST 1
L2_f  .FILL #5     ;;-- CONST 5
.END
```

```
int g( int x, int w ) {
    int y, z;
    y = x+5+w;
    z = y+2;
    return(z);
}

int main(void){
    int a, b, c;
    a = 1;
    b = 2;
    c = a+b;
    return( g(b, c) );
}
```

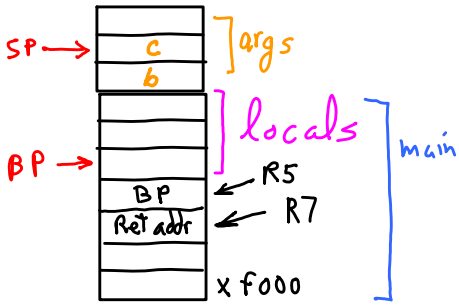



```
main
    ;;------- BEGIN ENTER --------
ADD R6, R6, #-1  ;;-- allocate ret val space
ADD R6, R6, #-1  ;;-- SP--
STR R7, R6, #0   ;;-- push ret addr
ADD R6, R6, #-1  ;;-- SP--
STR R5, R6, #0   ;;-- push BP
ADD R5, R6, #-1  ;;-- set new BP
                 ;;----- allocate locals
ADD R6, R6, #-3
    ;;------- END ENTER ----------
```



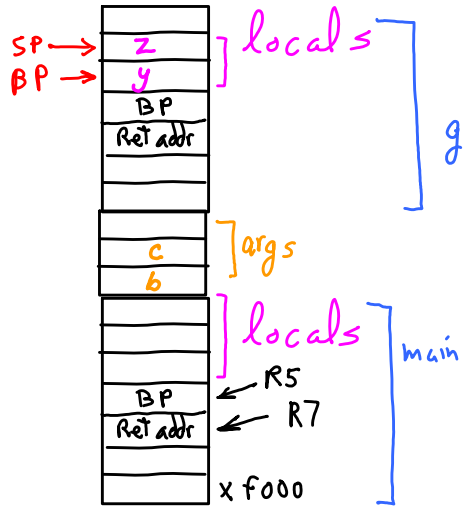

```
ldr R7, R5, #0  ;-- R7 <== b
ldr R3, R5, #-1 ;-- R3 <== a
add R3, R3, R7  ;-- R3 <== a+b
str R3, R5, #-2 ;-- c  <== R3
ldr R3, R5, #-2 ;--
ADD R6, R6, #-1 ;-- sp--
STR R3, R6, #0  ;-- push c
ADD R6, R6, #-1 ;-- sp--
STR R7, R6, #0  ;-- push b
ADD R0, R4, #0  ;-- R0 <== address of g() pointer
LDR R0, R0, #0  ;-- R0 <== address of g()
jsrr R0         ;-- call g()
```

} do arithmetic

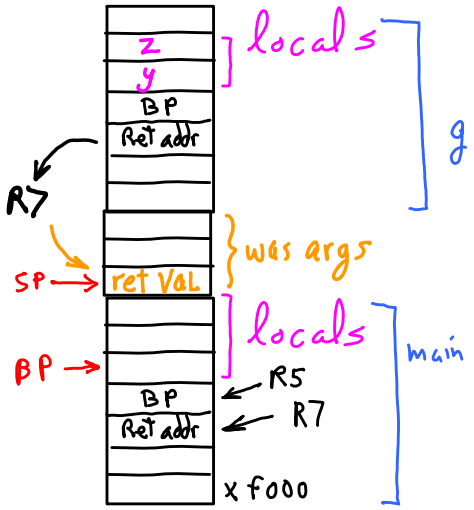


on call, SP points to 1st arg

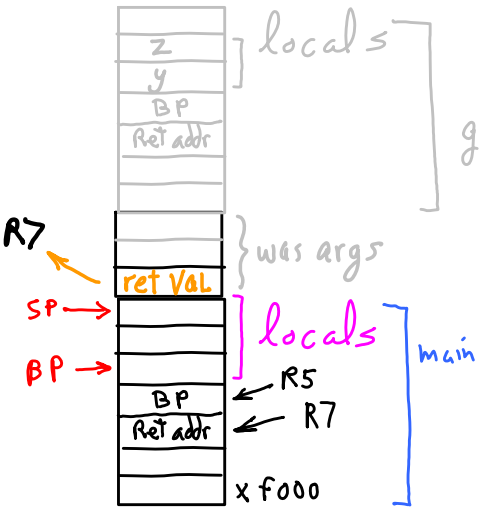
```
lc3_g
    ;;------- BEGIN ENTER --------
ADD R6, R6, #-1   ;;-- allocate ret val space
ADD R6, R6, #-1   ;;-- SP--
STR R7, R6, #0    ;;-- push ret addr
ADD R6, R6, #-1   ;;-- SP--
STR R5, R6, #0    ;;-- push BP
ADD R5, R6, #-1   ;;-- set new BP
                  ;;----- allocate locals
ADD R6, R6, #-2
    ;;------- END ENTER ----------
```




```
    ;;------- BEGIN-LEAVE --------
LDR R6, R5, #5   ;;-- SP to last arg
STR R7, R6, #0   ;;-- ret val to last arg
LDR R7, R5, #2   ;;-- get saved ret addr
LDR R5, R5, #1   ;;-- restore BP
    ;;------- END-LEAVE ---------
RET
```

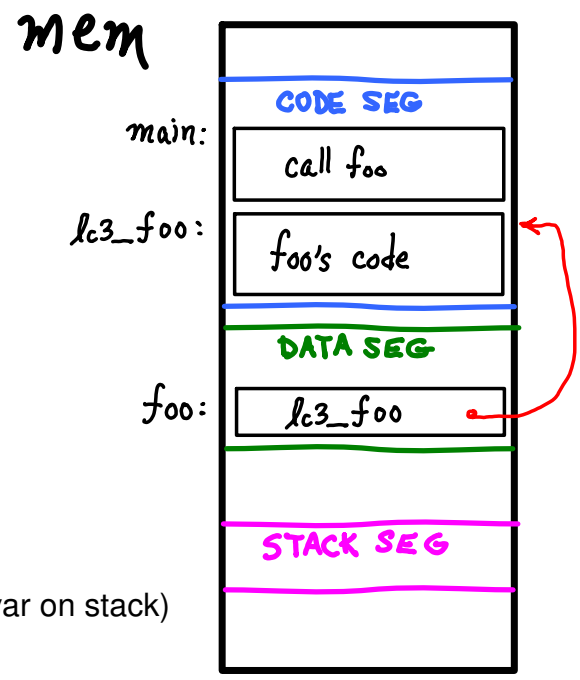



```
LDR R7, R6, #0    ;-- pop ret val to R7
ADD R6, R6, #1    ;-- SP++
```



On return, pop result
(or pop void result)

What is a function's name?



It's a pointer variable (a location in memory that stores an address)

```
void foo (void) {...}
```

You can pass it as an argument

```
dosomething( foo );
```

pass by value? (value on stack)

pass by reference? (address of var on stack)

How do we use it? An argument

Variable declaration:

```
void dosomething( void (* foo )( void ) ) {...}
```

a formal parameter, a variable

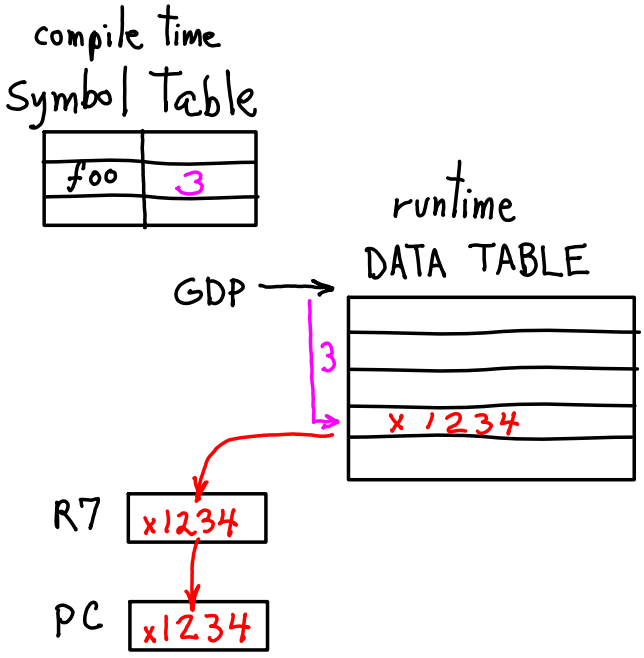
That can be de-referenced, it's value is an address

and that value used as a function address to jump to.

Now we can make the call:

```
( *foo )( );
```

For the compiler, the name is an offset into the DATA segment. The call uses it:




```
LDR R7, GDP__, #3   ← get the pointer value
JSRR R7             ← use its value: derefence
```

Here's a .o file produced by gcc.

you won't find printf here, it hasn't been linked yet.

Assembly is in ATT syntax: destination on right.

You can guess at meaning.

```
%> gcc -S fo.c
%> more f.s

        .file   "f.c"
        .def    ___main;    .scl    2;      .type   32;
        .endef
        .section .rdata,"dr"
LC0:
        .ascii "Enter %d numbers.\12\0"

        .text
.globl _main
        .def    _main;      .scl    2;      .type   32;
        .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $104, %esp
        andl    $-16, %esp
        movl    $0, %eax
```

ASCII x0A = LF

ASCII x00 = NUL

— save BP: push BP

— new BP: SP → BP

— make space: SP − 104

— align: x0 → SP[3:0]

... — more preamble