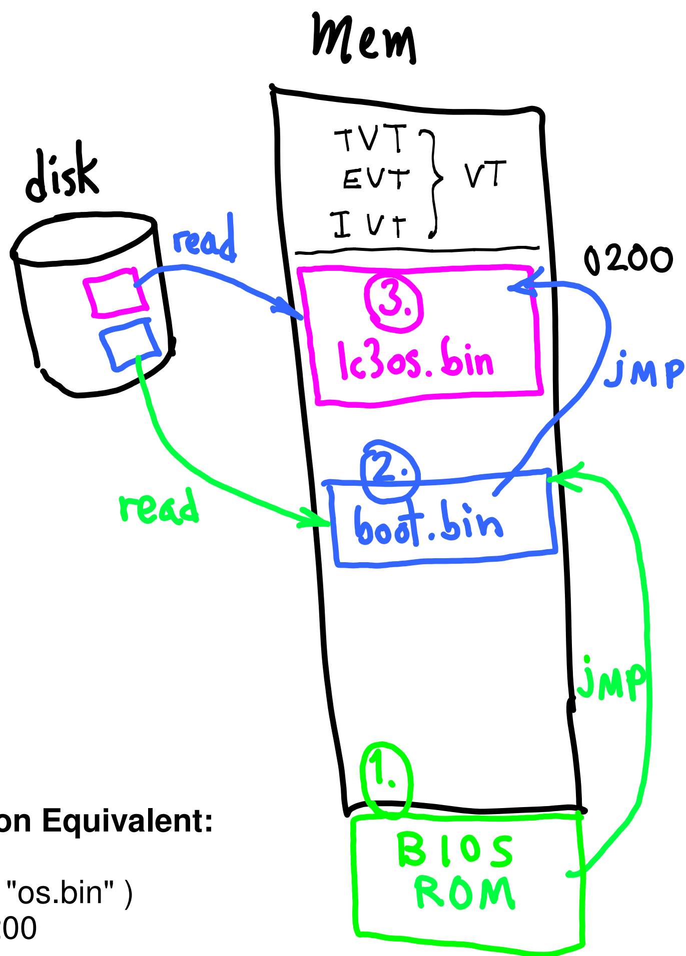
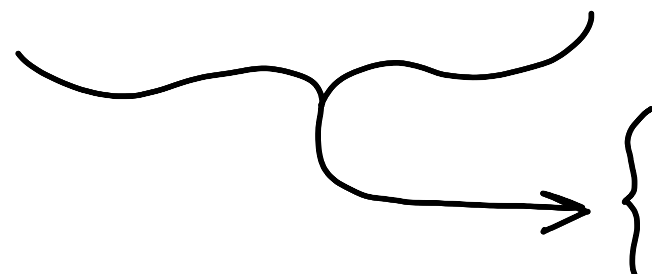
LC3 System Start-Up Assumptions
We will write an OS for the LC3.
What would a real LC3 do at start up?

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
--- 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)

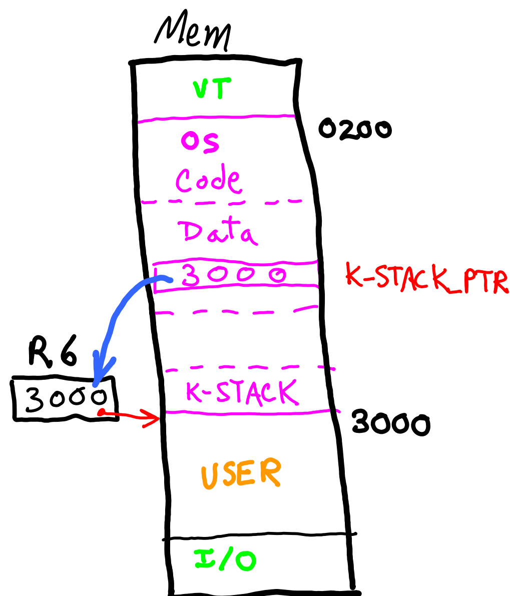
-- **set Supervisor's SP (R6)**
Note: can't use subroutines until SP is set up.

-- **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 "l": do_list(); break;
        }
}
```



;---- OS initialization
;---- Set up Kernel SP

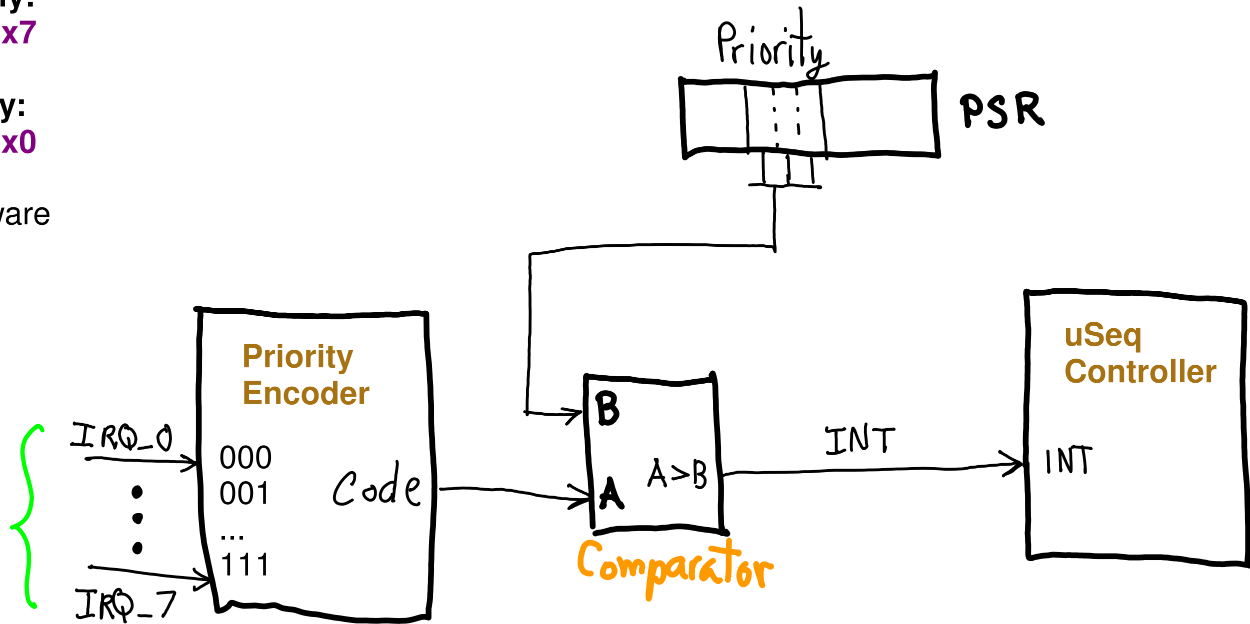
```
LD R6, K-STACK_PTR

K-STACK_PTR:    .FILL x3000
```

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

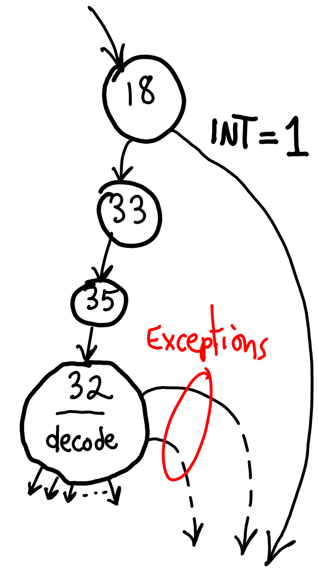
Enable INTs globally:
--- PSR.priority <== x0

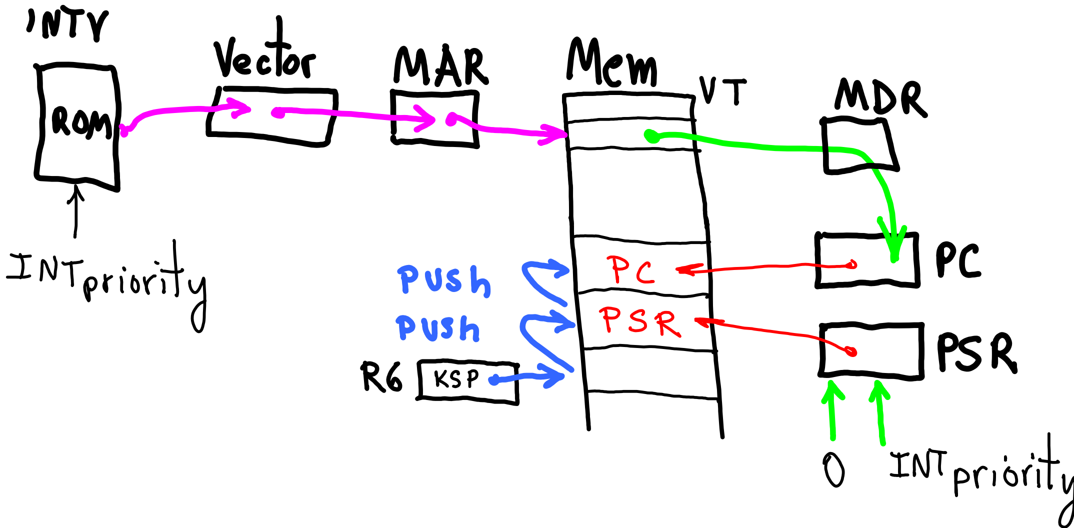
Can be done in software
---RTI tricks

Mem-IO Bus
Control-bus
IRQs from
devices



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





swap stacks
push PSR
PSR.mode <== kernel-mode
PSR.priority <== INTpriority
push PC
PC <== Mem[vector]

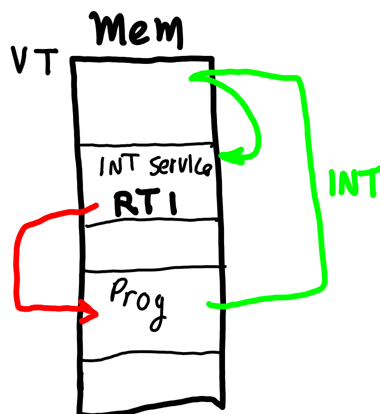
go To fetch-18

Overall effect:

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

--- Service the INT request

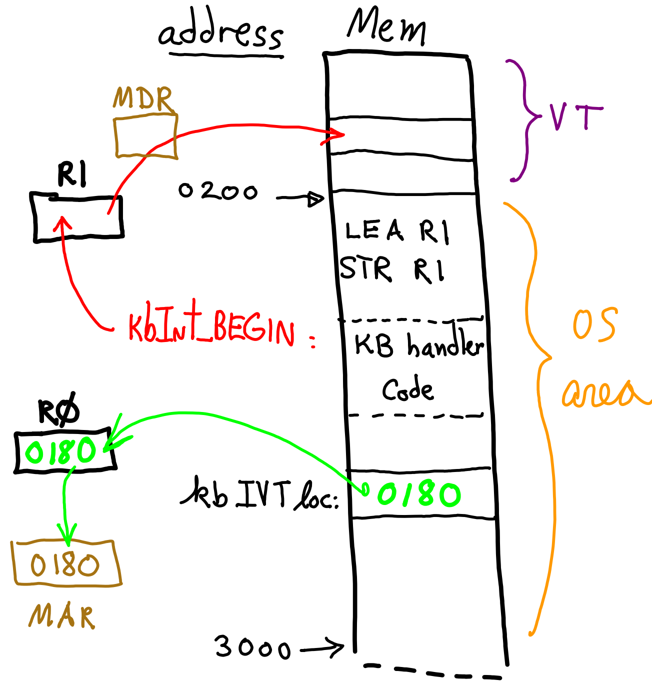
---Restore program's state
Jump back to instruction



Note:

kernel = supervisor
ksp = SSP

Init Service Routine



kbInt_init_BEGIN:

;------------- Set up KB VT slot

```
 LD R0, kbIVTloc            ;-- R0 <== address of KB's IVT slot
 LEA R1, KBint_BEGIN        ;-- R1 <== KB handler's address.
 STR R1, R0, 0              ;-- Mem[R0] <== R1
```

...

kbInt_init_END:

kbInt_BEGIN: push__(R7) ;--- save registers

...

RTI

kbInt_END:

...

kbIVTloc: .FILL x0180

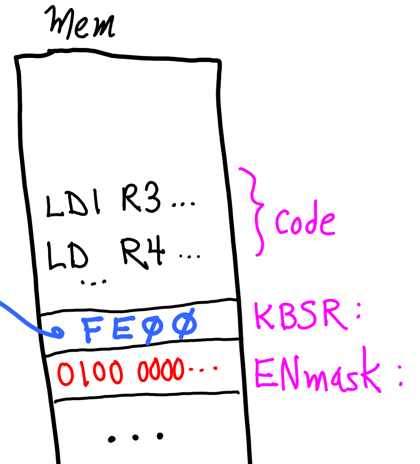
See symbol table for value of "kbInt BEGIN", it's an address.

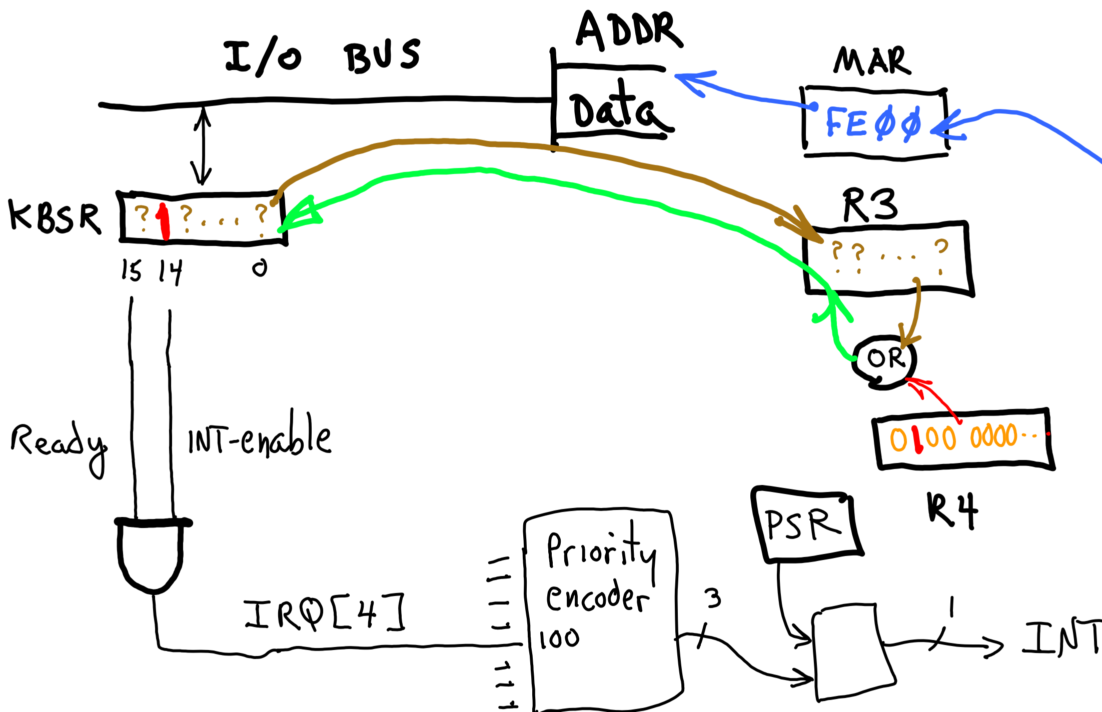
f.asm → lc3as → f.out, f.sym (symbol table)

;--------- Enable KB interrupts

```
LDI R3, KBSR      ;-- R3     <== content of KBSR.
LD  R4, ENmask    ;-- R4     <== enable mask.
...               ;-- R3     <== OR( R3, R4)
STI R3, KBSR      ;-- KBSR <== R3
...
```

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





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

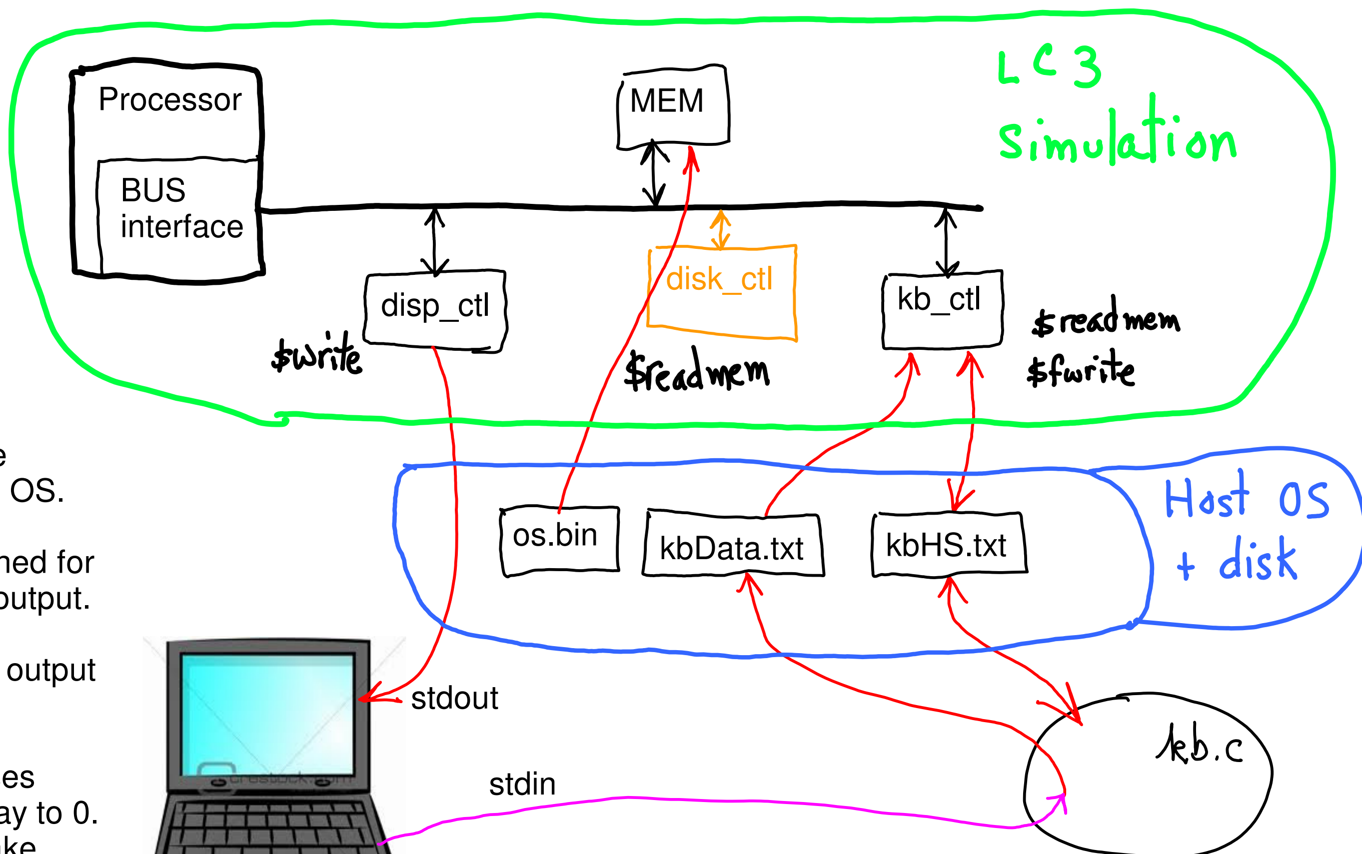
R4 gets bit-14 mask:
R4 <== Mem[**ENmask:**]

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

R3 data overwites KBSR data

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.

Buffers not flushed for individual char output.

Delay in seeing output from LC3.

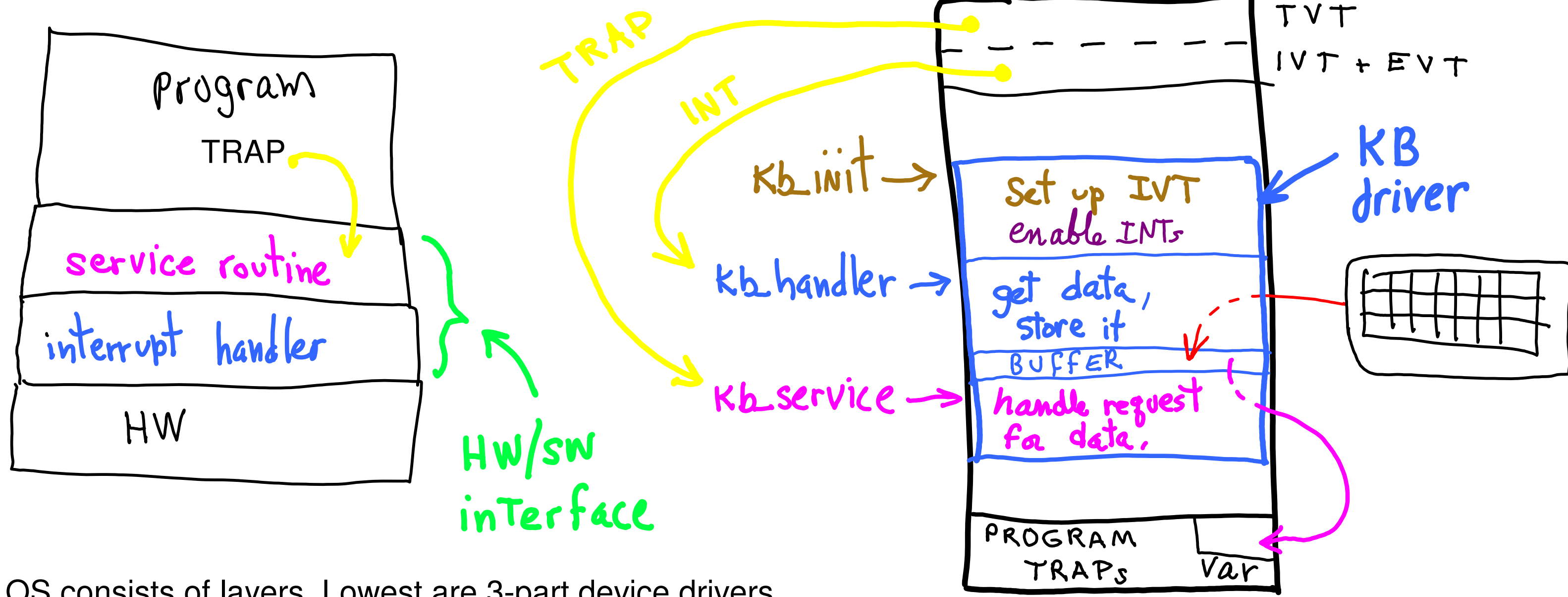
kb.c uses ncurses library, sets delay to 0. Writes handshake.

KBctl polls the file kbHS.txt to see if new data is ready.

w/o kb.c:
---- $write goes to stdout w/ host OS buffering.
---- kb INTs occur with fixed delays between.
input comes from KBctl's array, initialized:
[x21, x22, x23, ... ,] == [; # $ ...]

Use readmemb() to init KBctl's array?

LC3 OS structure, Low



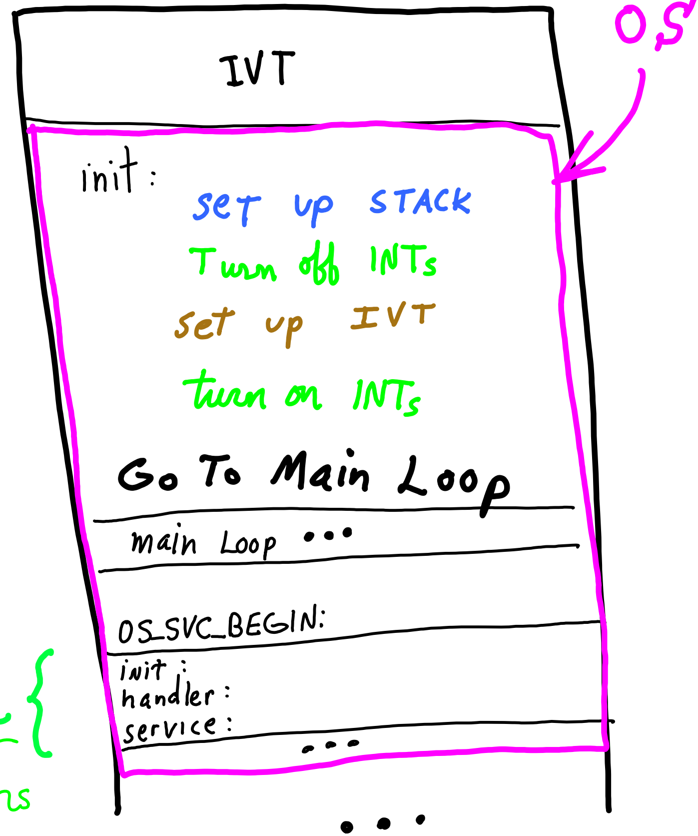
OS consists of layers. Lowest are 3-part device drivers.
(1) init, set IVT addresses, etc. (2) interrupt handler responds to HW device, buffers data (3) service routine handles requests from higher-level software (OS or user) to send data to program's memory area.

```
initOS_BEGIN:

 ld sp__, SUPER_STACK_ADDR

 jsr__( intsOff_BEGIN )

 jsr__( intsOn_init_BEGIN )
 jsr__( intsOn_init_BEGIN )
 jsr__( kbInt_init_BEGIN )
 jsr__( putc_init_BEGIN )
 ...

 intsOn__

 lea r7, mainOS_BEGIN
 jmp r7

 SUPER_STACK_ADDR: x3000
```

can't use trap

one call per for TRAPS, INTs, Exceptions

Now, can use Trap



A service

device drivers
print services
network services
...

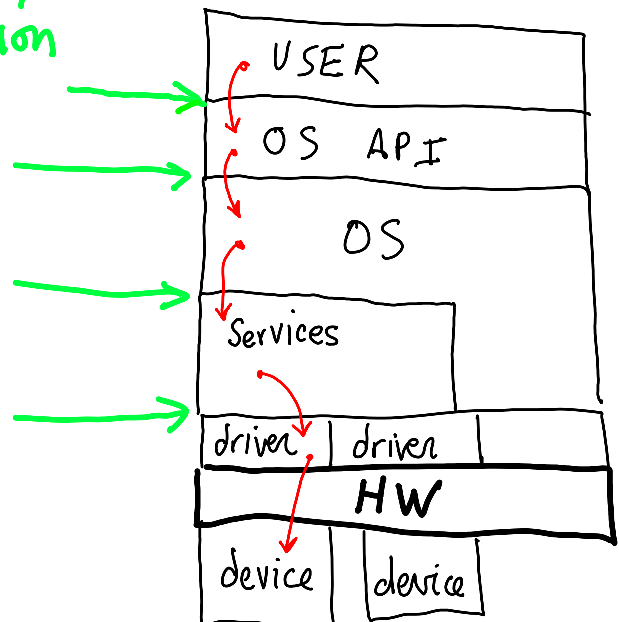
Hierarchical OS

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

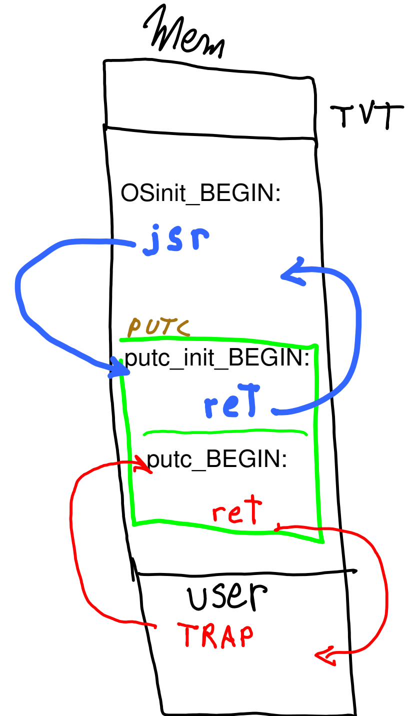


OS Layers | lowest-level

Module consist of two parts:
--- putc_init
Called as a subroutine (JSR) by OS init.
Returns via "RET", i.e., "JMP R7".
--- putc
Called as a subroutine via TRAP
Returns to user or kernel code via RET.

orig doesn't matter, we'll do linking later.

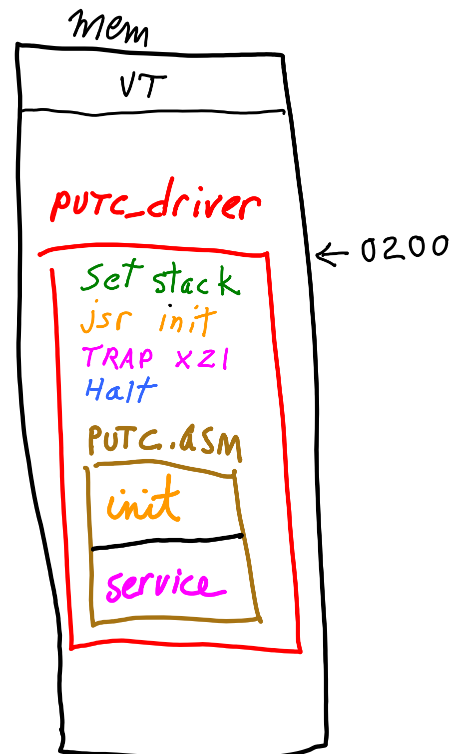
```
;------------------------------------
;-- putc.asm
;------------------------------------

  ;;;==================================================
  ;;;-- putc_init
  ;--------------------------------------------------
  putc_init_BEGIN: ;------------------------------------
    ld  r0, PUTC_TVT_LOC   ;-- r0 <== vector slot addr
    lea r1, putc_BEGIN     ;-- r1 <== svc routine addr
    str r1, r0,  0         ;-- IVT[r0] <== routine addr
  putc_init_END: ret ;------------------------------
  ;--------------------------------------------------
  ;;;==================================================
  ;;;-- putc(R0) - trap x21:
  ;--------------------------------------------------
  putc_BEGIN: ;----------------------------------------
    putc_POLL:             ;-- do
      ldi r2, DSR          ;--    status <== DSR
      brzp putc_POLL       ;-- until(status == READY)
    sti r0, DDR            ;-- DDR <== char (print char)
  putc_END: ret;-----------------------------------
  ;--------------------------------------------------
  ;;;==================================================
  ;;;-- Data area
  ;--------------------------------------------------
  PUTC_TVT_LOC:   .FILL x0021  ;-- putc IVT slot.
  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

TESTING

```
;------------------------------------
;-- putc_driver.asm
;------------------------------------

ld sp__, SUPER_STACK_ADDR    ;-- Set stack.
jsr__( putc_init_BEGIN )     ;-- init putc
ld r0, MY_CHAR               ;-- arg <== MY_CHAR
putc__                       ;-- trap( arg )
done:
ld  r4, STOP_CLOCK     ;-- R4  <== STOP_CLOCK
sti r4, MCR            ;-- MCR <== R4 (halts LC3)
SUPER_STACK_ADDR: .FILL x3000
STOP_CLOCK:    .FILL x8000  ;-- MCR[15]=1
MCR:           .FILL xFFFE  ;-- mach. ctl reg
MY_CHAR:       .FILL x0041  ;-- ASCII 'A'
```

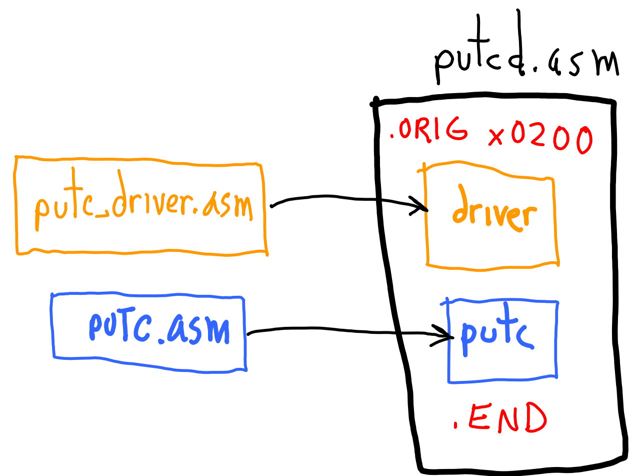


Build & Run

1.

1. edit

"make putc_driver" ===> putcd.asm



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

% make os.bin OS=putcd

cat putcd.asm | lc3pre > putcd_pre.asm

lc3as putcd_pre.asm

cat putcd_pre.obj | obj2bin | sed '1d' > putcd_pre.bin

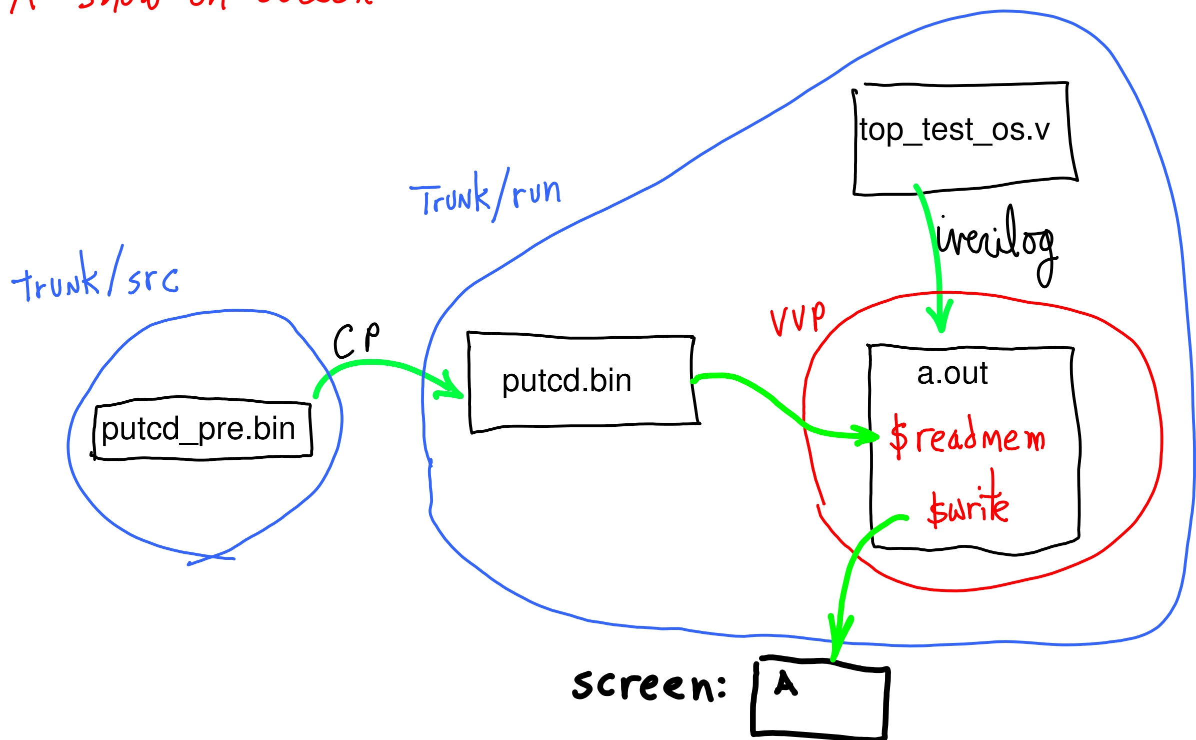
cp putcd_pre.bin ../run/putcd.bin

"make usage" in both run + src: hints for commands

3. Run does 'A' show on screen?

cd ../run

vvp a.out



Aside: write a link editor?

With a little bit of editing .asm source code we can link .obj files. We need the symbol table information to adjust addresses.

PUTC.ASM → make as ASM=putc (lc3pre, lc3as) → putc_pre.obj, putc_pre.sym

putc_driver.asm → make addST SRC=putc DST=putc_driver (with putc_pre.sym) → putc_driver_st.asm (putc_driver.asm, putc_pre.sym)

putc_driver_st.asm → link editing → lc3pre, lc3as → putc_driver_st_pre.obj

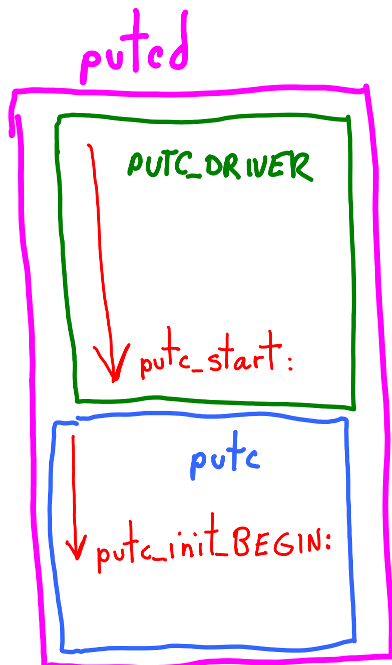
putc_pre.obj, putc_driver_st_pre.obj → make link 1ST=putc_driver 2ND=putc → putcd.obj (putc_driver_st_pre.obj, putc_pre.obj)

```
;-----------------------------------------
;-- putc_driver.asm
;-----------------------------------------
        .ORIG x0200
      ...
   lea R1, putc_start
   ld R2, putc_init_BEGIN_off
   add R1, R1, R2
   jsrr R1
    ...
putc_init_BEGIN_off:  .FILL 0
putc_start:
        .END
```

} JUMP To putc_init

```
// putc_pre.asm Symbol table
// Scope level 0:
//    Symbol Name          Page Address
//    ----------------     ------------
//    putc_init_BEGIN      0004
//    putc_init_END        0007
//    putc_BEGIN           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
```

COPY



putc_start: ==
size_of(**putc_driver**)

x4 ==
offset to putc_init_begin
relative to start of **putc**

putc_start + x4 ==
address of **putc_init_BEGIN**

EDITING: Copy the value of

putc_init_BEGIN

from **putc's symbol table** to

.FILL 0

---- lables need to be identified as "external"

---- can also handle data references similarly

---- next step: do editing of .obj files, not .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

4. Avoid re-compile/re-assemble

How to Link?

1. by hand: include source code into one unit

2. use a linker: write a low-level linker for linking .obj files

3. use C compiler, it includes a linker

Alternatives to hand linking:

1. Write your own LC3 .obj linker!

2. Write LC3 assembl
    ---- Use lcc C-style conventions
    ---- let lcc do the linking for you:

    lcc main.lcc foo.lcc

  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)

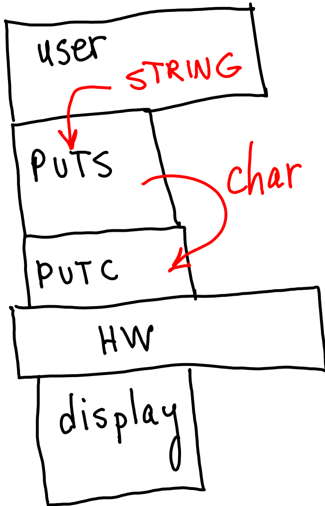
lcc -c foo.c ⇒ foo.lcc (assembly)

lcc foo.c ⇒ { a.asm, a.sym, a.obj }

OS, 2nd layer user service

2nd layer service built on top of 1st layer abstraction.

A service to print a string to the display.




```
;--------------------------------------
;-- puts.asm
;--------------------------------------

  ;;;===================================================
  ;;;-- puts_init
  ;---------------------------------------------------
     ...

  ;;;===================================================
  ;;;-- puts- trap x22:
  ;;;-- Display string, R0 == address
  ;---------------------------------------------------
  puts_BEGIN: ;-------------------------------------
    push__( r7 )

    mov__( r1, r0 )          ;-- R1 is char_ptr
    ldr r0, r1, 0            ;-- char <== *char_ptr

    while_nonNUL:            ;-- while( char != NUL )
      brz end_puts_while     ;-- begin_while_non_NUL
      putc__                 ;--   print char
      add r1, r1, 1          ;--   char_ptr++  (next char)
      ldr r0, r1, 0          ;--   char <== *char_ptr
      br while_nonNUL        ;--
    end_puts_while:          ;-- end_while_non_NUL

    pop__( r7 )
  puts_END: ret ;------------------------------

  ;;;===================================================
  PUTS_TVT_LOC:   .FILL x0022  ;-- puts IVT slot.
END:
```

Test same as putc

→ need putc linked.

→ driver units both putc and puts.

Save R7

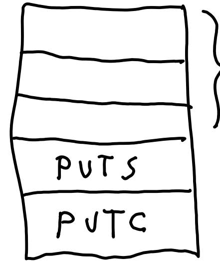
putc__ ⇒ Trap x21

And then?

To do List

Provide higher-level abstractions for display?

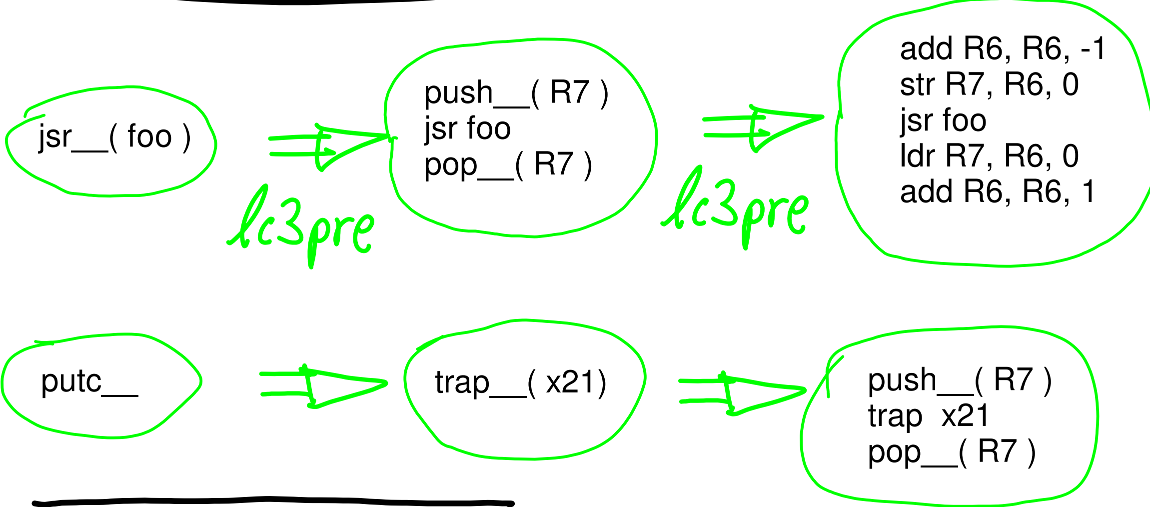
Windows, events?



display utilities

Nested Calls

Saving R7



lc3pre

Defines useful "psuedo" language extensions: String substitutions in source code.

Other extension: save all registers, increment a register, ...

OS design

push onto stack? Send address of struct in RØ? Define general interface w/ uniformity across different devices?

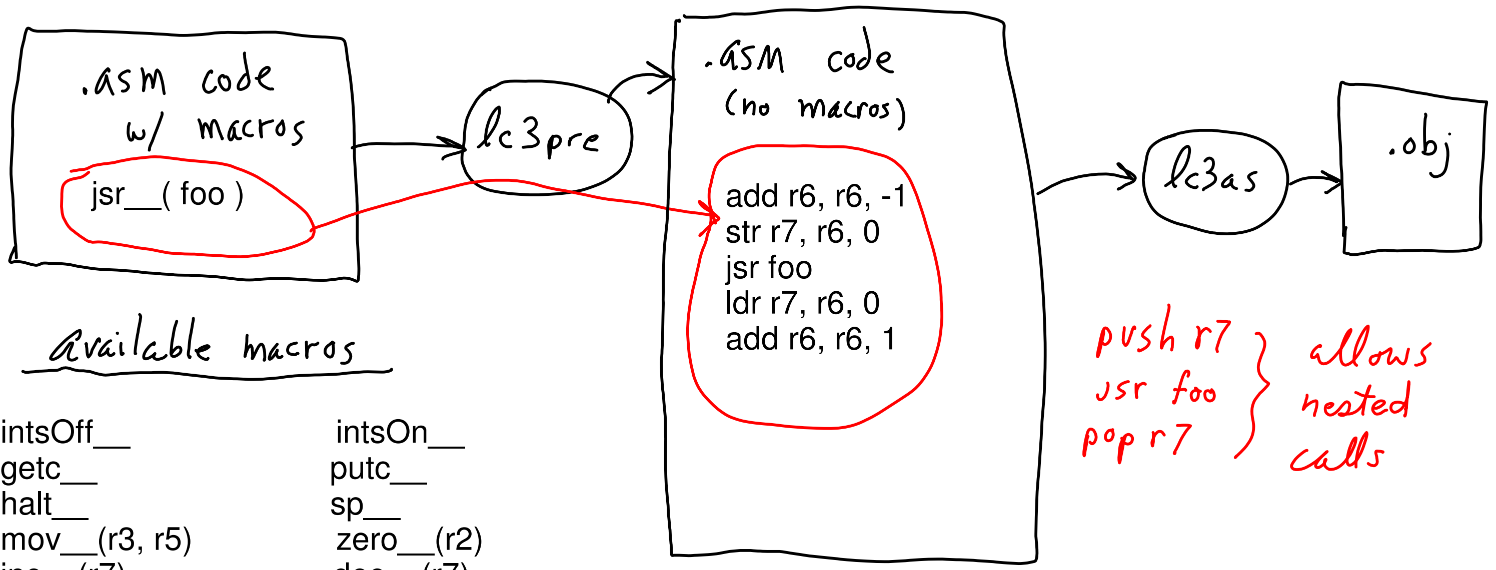
Mode changes? (Use Illegal Opcode Exception?)

higher-level services

- Clean 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

Aside: lc3pre

Provides language extensions:
translates macros to ASM code.



available macros

```
-- intsOff__                  intsOn__
-- getc__                     putc__
-- halt__                     sp__
-- mov__(r3, r5)              zero__(r2)
-- inc__(r7)                  dec__(r7)
-- push__(r5)                 pop__(r5)
-- sub__(r1, r2, r3)          or__(r1, r2, r3)
-- jsr__(mySub_BEGIN)         trap__(x13)
-- set__( r0, FALSE__ )
-- set__( r1, 5 )
```

Provides mechanism for nested calls/recursion

All macros have "__" double underscore

Condition codes are set as expected

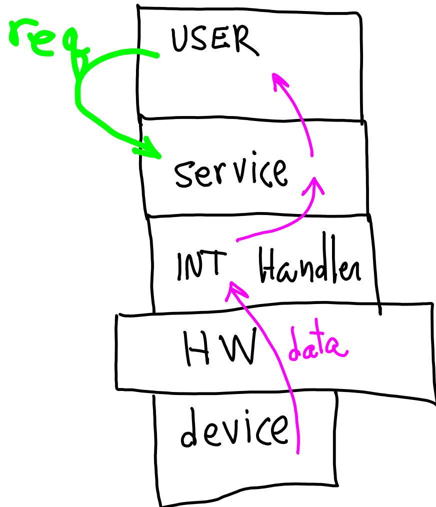
Argument registers are unchanged, only destination changes

```
example code        translation
-----------------   ------------------------

mov__(r3, r5)       add r3, r5, 0        ;;-- r3 <== r5

zero__(r2)          and r2, r2, 0        ;;-- r2 <== 0

inc__(r7)           add r7, r7, 1        ;;-- r7 <== r7 + 1

dec__(r7)           add r7, r7, -1       ;;-- r7 <== r7 - 1

push__(r5)          add sp__, sp__, -1
                    str r5, sp__, 0

pop__(r5)           ldr r5, sp__, 0
                    add sp__, sp__, 1
                    mov__(r5, r5)        ;;-- sets NZP CCs per r5

sub__(r1, r2, r3)   not r3, r3           ;;-- r1 <== (r2 - r3)
                    add r3, r3, 1
                    add r1, r2, r3
                    add r3, r3, -1
                    not r3, r3           ;;-- r2 and r3 unchanged,
                    mov__(r1, r1)        ;;-- sets NZP CCs per r1
```

(see, src/lc3pre)

Interrupt-driven I/O

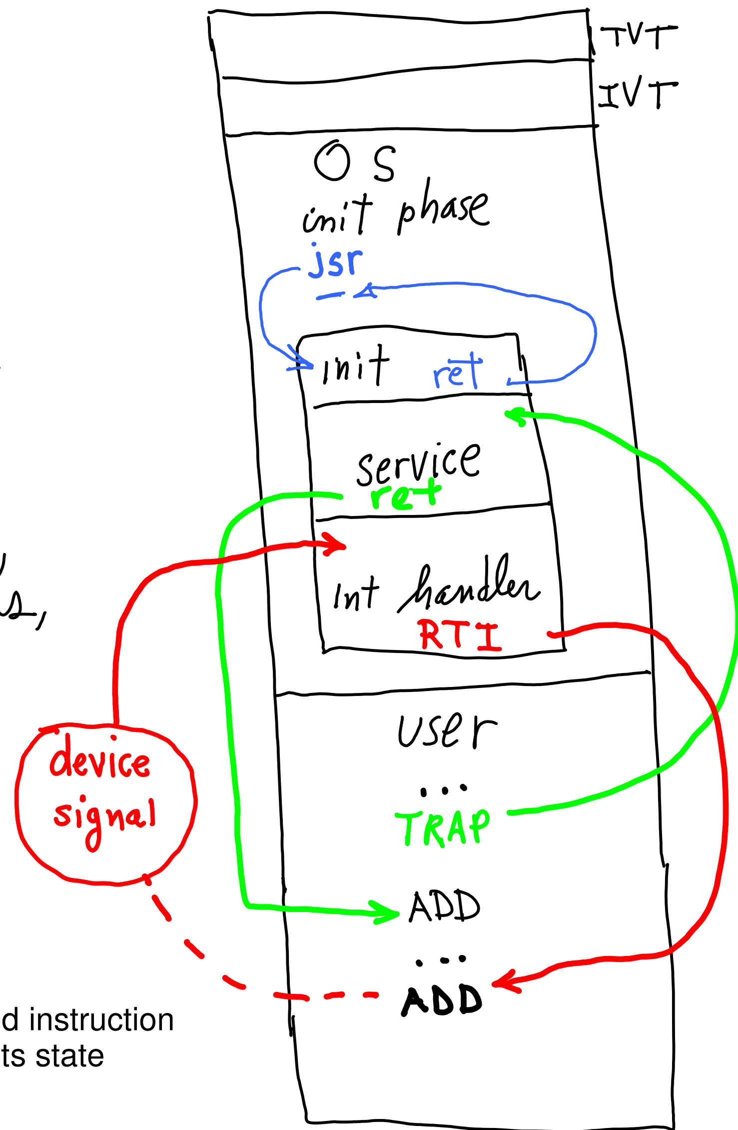
OS layer w/ 3 parts



— 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:

--- save state, turn off interrupts { push(PSR, PC), swap stacks, PSR.Priority = 4; In handler: save regs as needed

--- 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)

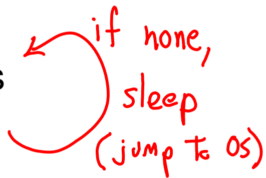
--- restore state, turn on interrupts } In handler: restore regs; HW: interrupts turned on via pop(PSR)

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

↑ depends on popped PSR[15] ↑ via pop(PSR)

=========================
Keyboard service
=========================
Forever()

--- check queue for requests



--- 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

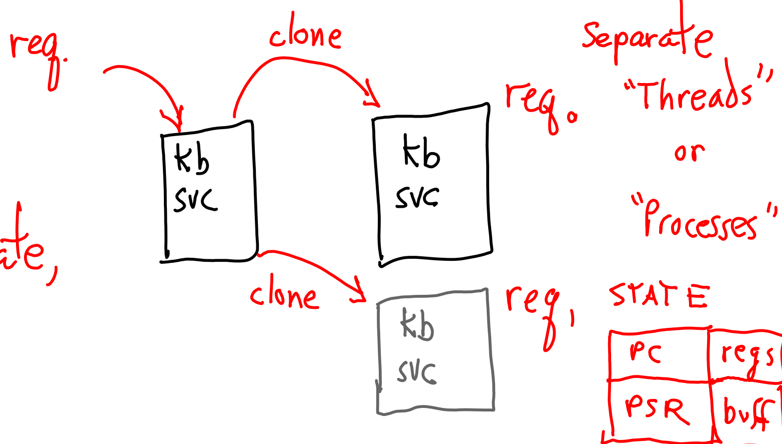
=================
Requester
=================

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

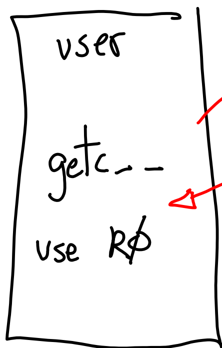
--- send request
trap to service

--- on return
use data in local buffer

multiple requestors



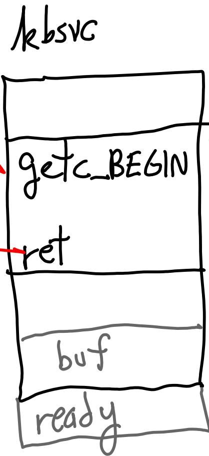
Our svc Req structure



TRAP

R0

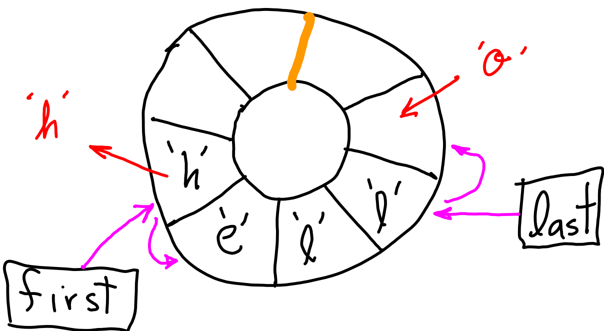
kbsvc

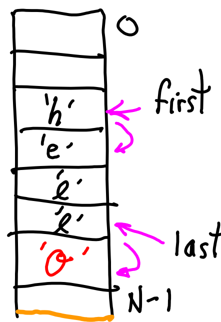


ready?
yes: R0 ← buf
buf now empty? ready ← 'no'
ret
no: wait (sleep?)

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

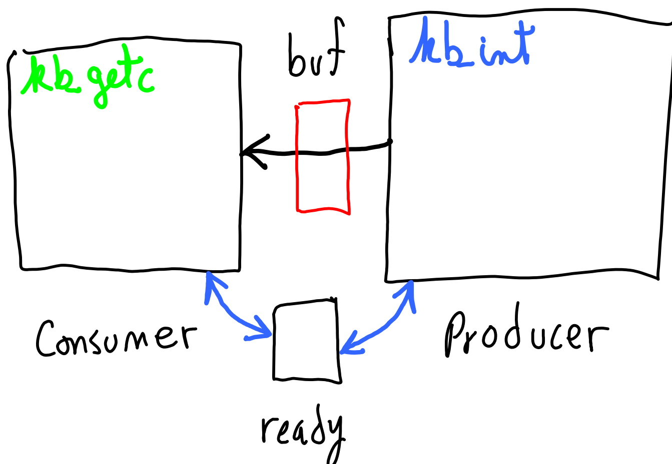
(state saved, registers saved, switch)

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";

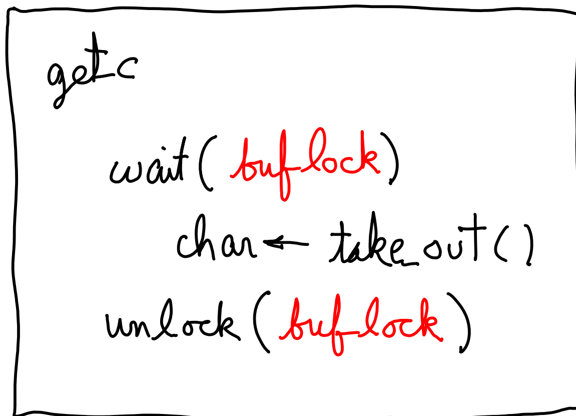
--- Controlling access via "ready";

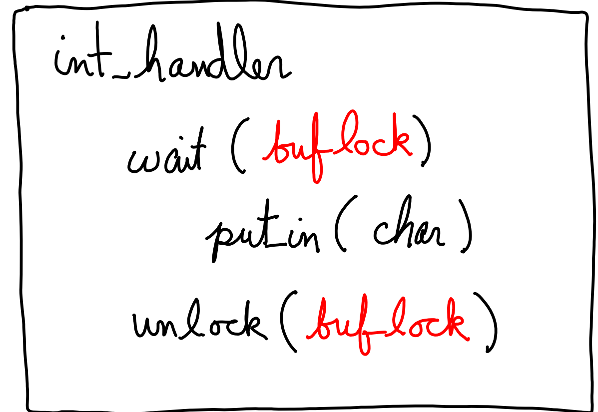
--- w/ Waiting (sleeping).

Concurrency

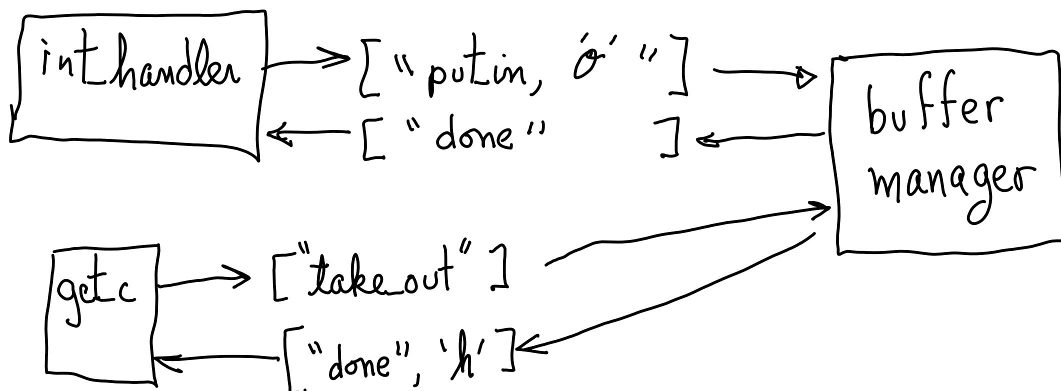
Two general structures

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

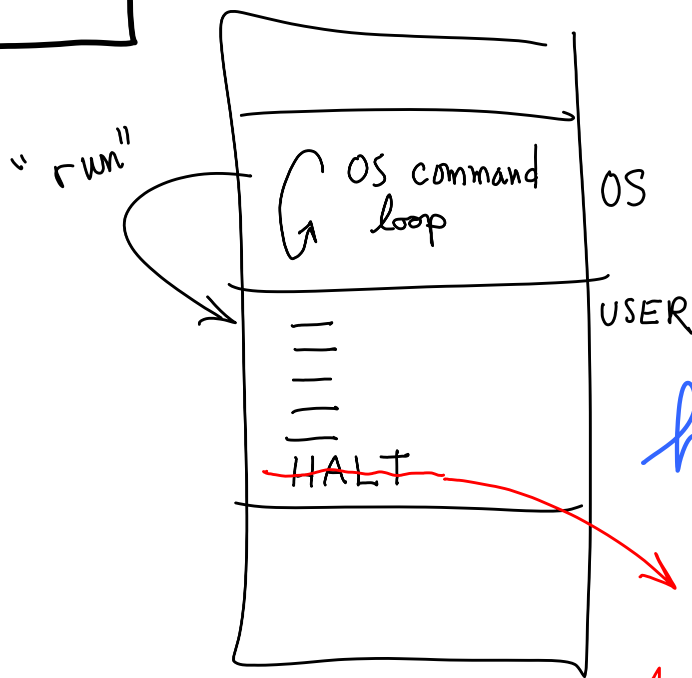




- messages:
  - wait for message, send reply
  - send message, wait for reply



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?

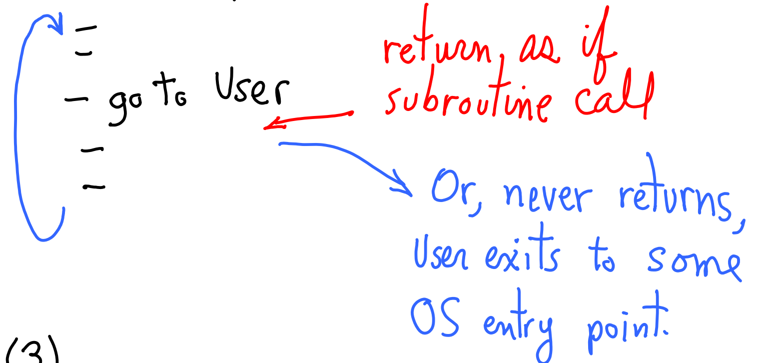
halt: turn off sys_clk?

NO!

ret_to_OS

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

Possibilities



1. – 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).
2. – 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 (update lc3pre). 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)

An interrupt
--- saves state of currently executing code
--- transfers execution to other code

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

IN GENERAL, Waiting/Sleeping needs same mechanism.

Switch_Context()
--- save current code's state (save registers, PC, PSR, plus memory?)
--- set up state of code to switch to (fill registers, set PC and PSR, plus memory?)
--- jump to new code

NEED: a list of code that is currently "in" execution
--- each entry stores state of concurrently executing code (a "thread" or "process")
--- "scheduler" to select next thread to execute, and a way to run run scheduler
--- a way to select/signal a process/thread to wake up, i.e. execute

NESTED INTERRUPTS is similar:
--- switches from one interrupt handler to another (scheduled by higher priority)
--- switches back to interrupted handler

AHA! Provide interrupts to cause switching: Hardware Timer
--- This interrupt handler simply runs the scheduler

OUR LC3 OS ---- the bottom line
--- Too much overhead for us to implement wait/sleep and switching
--- Our service routine (getc) will busy-wait (polling) until an interrupt handler sets a ready flag,
which in effect puts the requesting program to sleep.