# "How the Multics Call / Save / Return sequence got that way"

7094 call

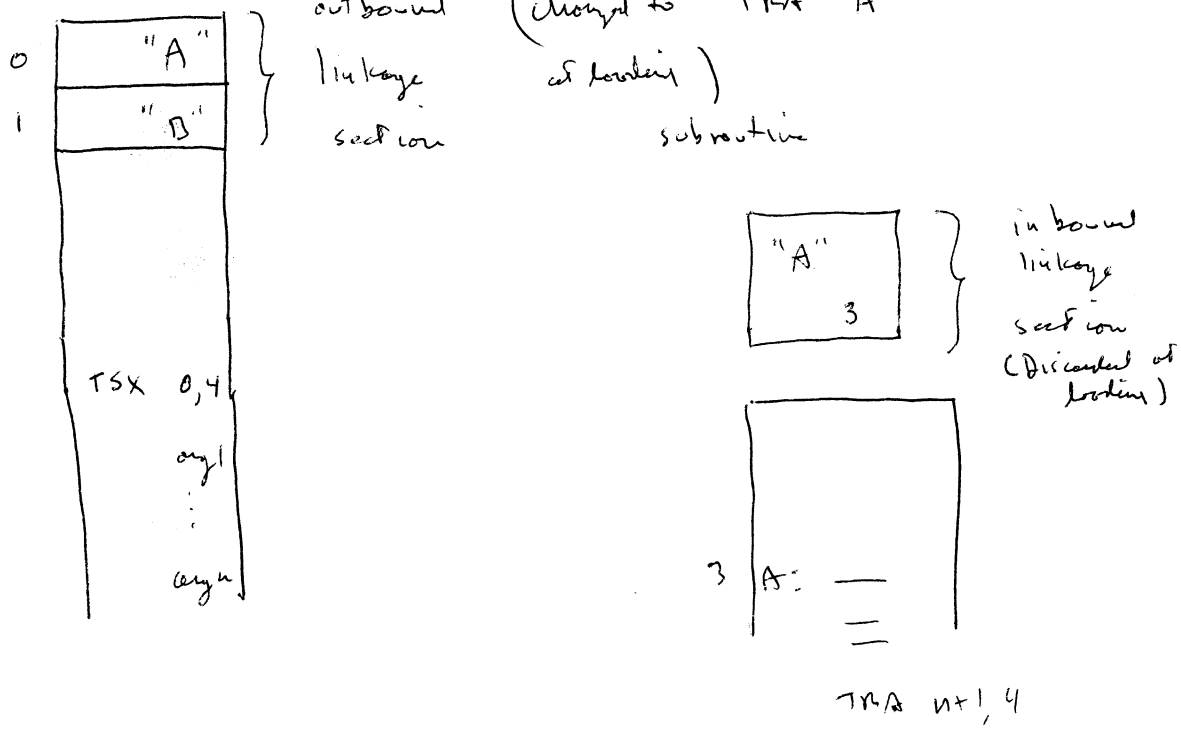
```
                    :
                    :
1   →
2   →   TSX     A, 4
            —       arg1
            —       arg2
                    :
            —       argn
3   →
8   →
```

```
3   →   A:  —                   (optional save -- if you use the index regs, save them)
4   →       —
5   →       —
6   →           TRA     n+1, 4      (return)
```

Objectives:

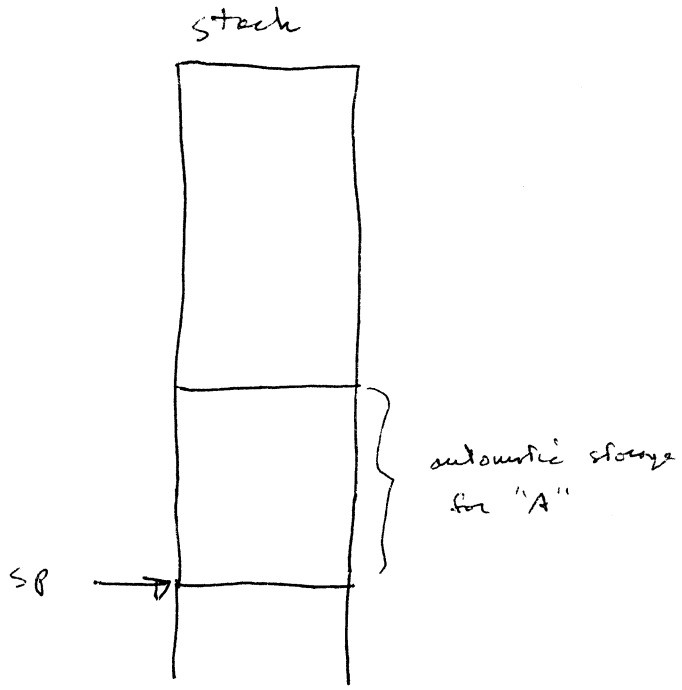
1. ~~Fast~~ Closed subroutines

2. Separately translated subroutines



3. Distinct static and automatic storage

Use a stack for auto; must push + pop at call time
a. dedicate a stack register ("sp")




subroutine

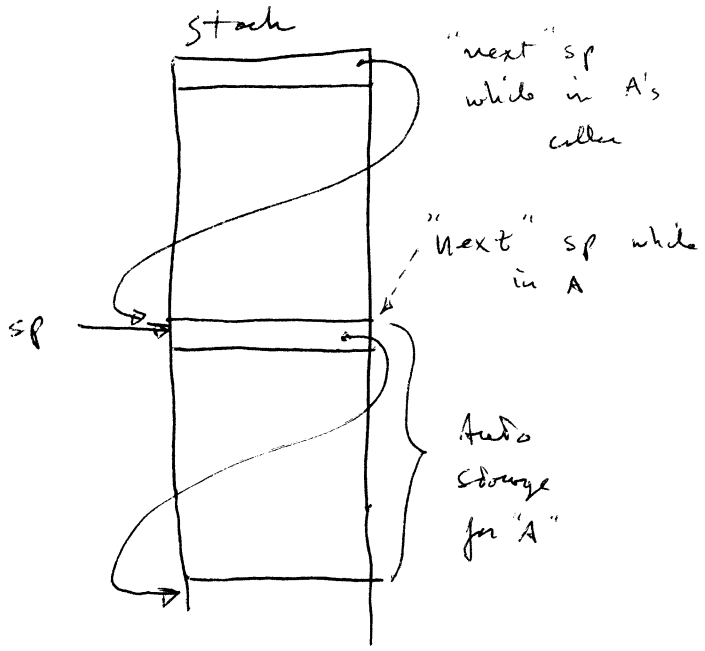
A: ~~save sp~~
push sp = sp + t
.
.
.
restore sp = sp - t
jmp L+1, 4

{ # of automatic local vars

4. Want to add additional automatic storage at execution time; value of "t" is computed on your go.

Strategy: have sp point to beginning of frame rather than the open end, keep next sp around.

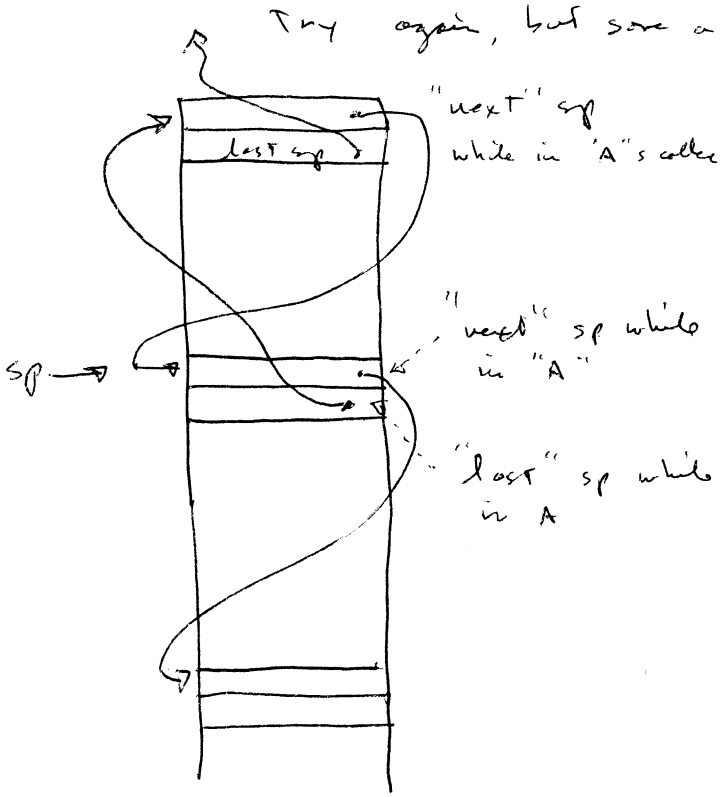
(If sp points to end of frame, when sp moves, old to your frame) reference must be updated also)




A: load bp from next sp (sp|0)
~~add~~ push bp = bp + t
load sp from next sp
~~save~~
store bp in next sp (sp|0)
.
.
.
.
.
.

need to "restore" sp to return!

Try again, but save a back pointer




A: load bp from sp|0
store sp in bp|2
push bp = bp + t
~~load sp~~
store bp in bp|0
load sp from ~~next sp~~ sp|0
.
.
.
.
.
.

} push or pull "next" sp as often as wanted

load sp from sp|2
return nil, 4

5. Want to accept ~~wait~~ an interrupt, fault, or signal at any time, - in midst of call
- on some stack

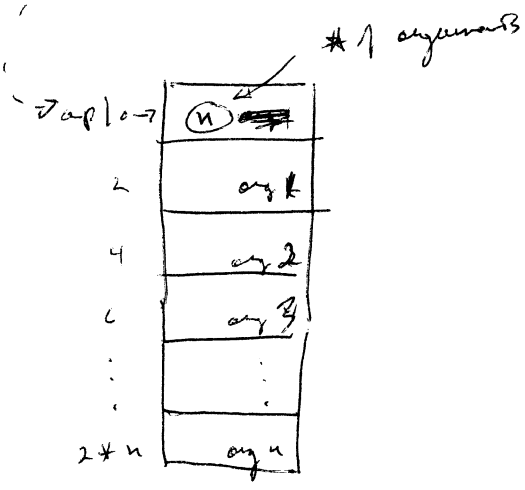
Technique, at interrupt time
look at sp
go to sp[0] ("next" sp)
go to next sp + 32; start new from there.

N.B. some proposed call/return sequences did not want this degree of ~~in~~

6.8. Want to allow variable # of arguments with count and return instruction

ST strategy: load a register to point to arg list, which is with data rather than in instruction pointer (will make pure procedures easier, later)

```
                                    A:  save
load ap = arglist
TSX     0,4                         load A      ap|2, *
always return →
to 1,4
                                    return
                                    load sp     sp|2
                                    TRA         1,4
```



7. Want to allow subroutine to be in a different segment
(Need to save status word of caller)

Two ways: both in hardware on instruction

(1) STCD X — store status word of the X
TRA A
RTCD X — restore IC to (status word of X) + 1

(2) TSBcp A

return: TRA cp|1

In 645, there were only 4 registers available, so method (1) was chosen. Probably method (2) is conceptually simpler; also form of subroutine does not need to reserve (and therefore save) cp.

N.B.: assumption of 645 – all registers are saved + restored

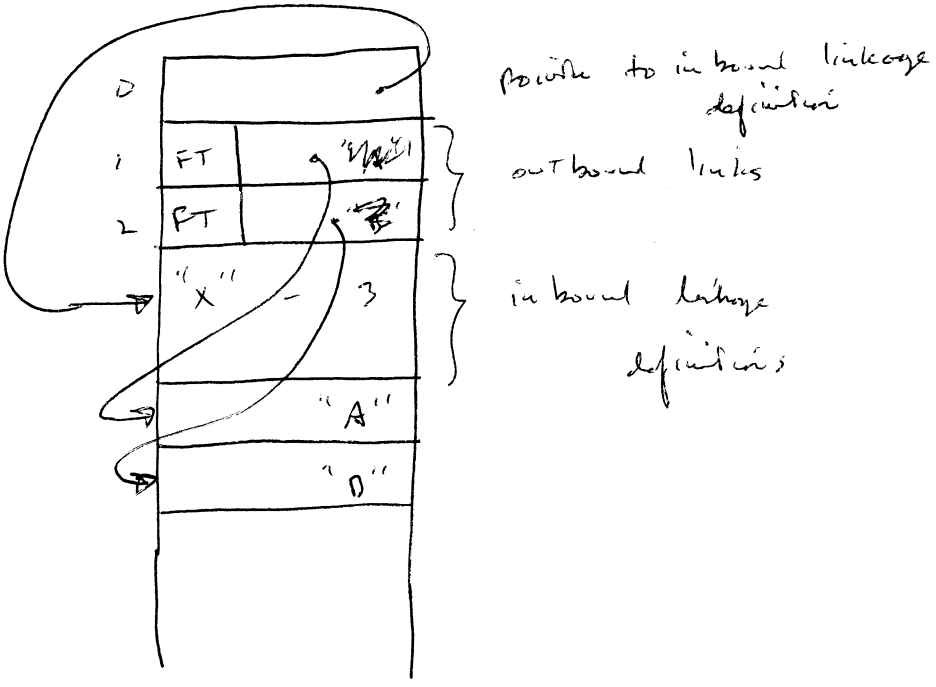
another assumption – subroutine only saves those it uses.

Also: must figure out meaning of "A"

7a. Want multiple entry points in a segment
(Solution is sticky)
Use name of segment + name of entry point

8. Want to allow Dynamic Linking

Must a. not demand in bound linkage
b. come first on out bound frames



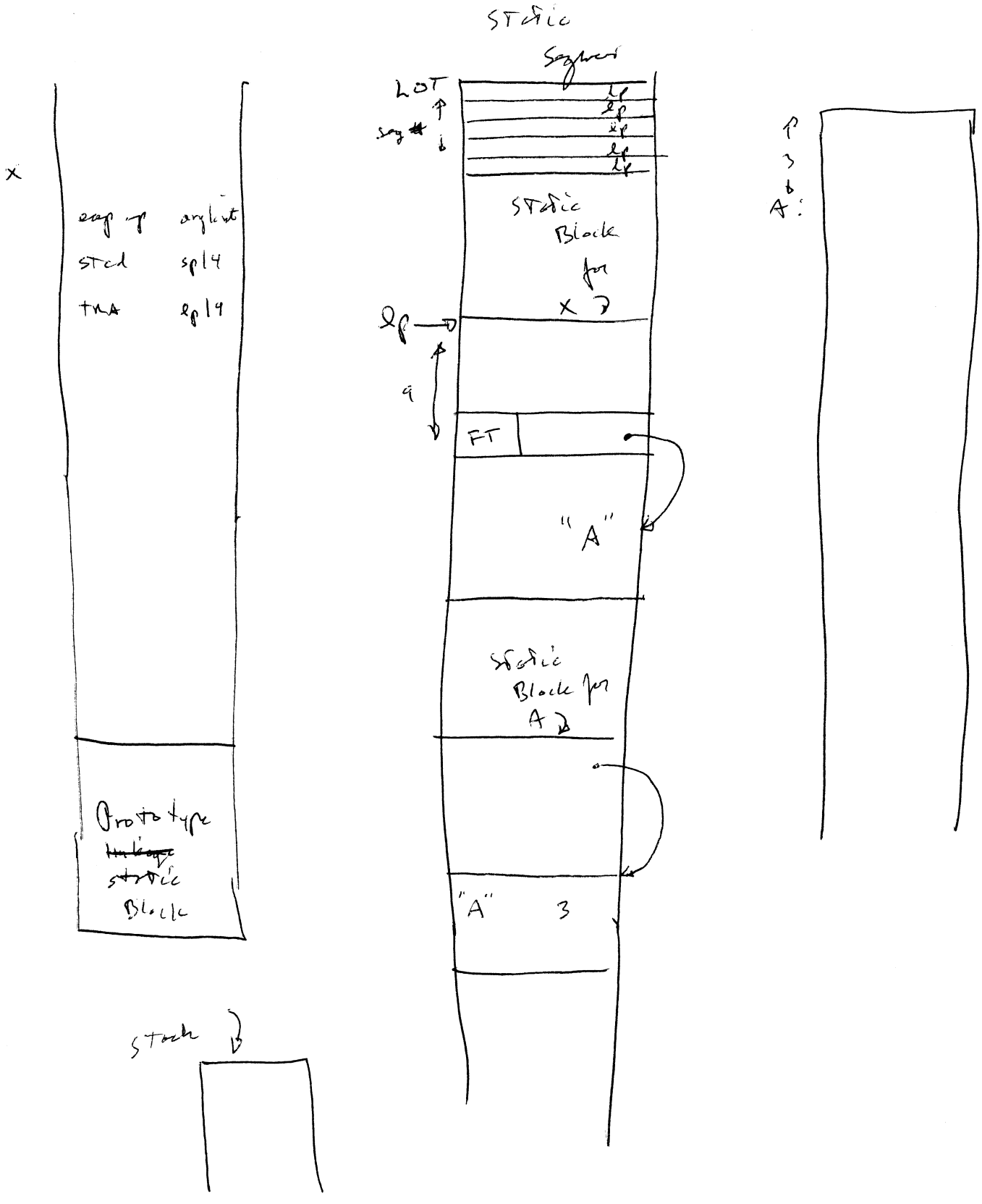
9. Want to allow shared procedures.

Tactic: make procedure body pure,
give each process a copy of the variables
give each process a stack.

Problems: linkage section
argument pointers
static variables

- Hidden objective!
  10. 9. Want to allow subroutines to be different in different processes.
- Hidden consequence!
  10a. 9a. Seg #'s are different in different processes.

Solution: Take linkage section + static variables,
create a "static block"
- prototype at end of procedure
- when running, copy prototype to a per process static segment
- maintain a pointer ("lp") to current static area
- must move "lp" around on call and return


static
Segment
LOT
Static Block for X
qp
q
FT
"A"
Static Block for A
"A" 3
Prototype
static Block
stack
A:

Dynamic linking faults:

1. get name of procedure wanted ("A")
2. Search file system for a segment named "A"
3. Get a segment # for A (= A#)
4. look in LOT for A#. If there, go to 7.
5. Allocate a core in which for A# ~~linkage section~~ static block.
6. Copy A# ~~linkage segment~~ prototype static block, place A's # in LOT of A#
7. look in A's static block for entry point offset.
8. construct ITS pointer to A#| offset, replace FT
9. continue as if no fault occurred.

---

How is # set on call + return?

Old Multics: Through through linkage section which leads one s# by EAPn *

New Multics: look up owner s# in LOT, load # from there.

10. Want to allow cross ring calls to be the same.

a. Stack ★ = ring ★ (Hardware knows this)

b. Pointer get ring ★'s to allow argument validation

c. ~~Call instruction does STCD + TRA~~
~~switching rings if necessary, and storing~~
~~return in the new stack!~~

c. Call instruction adjusts sp if necessary (knows PR2 = ring ★)

d. sb|0 always contains "next" sp, so callee can always find out where to go

call:

| | |
|---|---|
| eppp sp | arglist |
| stcd | sp\|20 |
| callsp | entry point |

save

| | | |
|---|---|---|
| epp bp | pr7\|0,* | get "next" sp |
| spp sp | bp\|16 | save "last" sp |
| epp sp | bp | new sp |
| epp bp | sp\|4 | |
| spp bp | pr7\|0 | set new "next" sp. |

ret

Pointer to LOT is in STack header

return

| | | |
|---|---|---|
| spp sp | pr7\|0 | |
| epp sp | sp\|16,* | |
| rtcd | sp\|20 | |