Subroutine Linkage
- Save a return address, and return to it
- Pass arguments
- Return results
- Provide local variables / storage / registers

Ex: \( a = f(b, c) \) \( f(a, b) \) \( \text{int } c; \) \( c = a + b; \) \( \text{return } c \)

\[ \sim \text{Asm} \]
\[ \text{Ops: } \]
- push \( c \)
- push \( b \)
- call \( f \)
- \( a = RO \)

\[ f(): \]
- sub \( \text{size(local s), SP} \)
- mov \( * (SP-2 \text{words}), R0 \)
- mov \( * (SP-3 \text{words}), R1 \)
- add \( \text{size(local s), SP} \)
- return

Base ptr = @ enth-a function, usually do:
- add \( R0, R1 \)
- mov \( R1, \text{MSP} \)
- mov \( * (SP), R0 \)

\[ f(\text{?):} \]
- push BP (base point)
- BP = SP
- SP++ = locals

---

Subroutine parameters right before base ptr; locals right after base pointer.

Caller save vs. callee save registers:
- Saved by subroutine if \( f() \) wants to use them again, vs.
- Saved by caller, which knows the regs won't survive subroutine.
Argument Order: last first (C), first first (Pascal)

- Push last arg to subroutine first, so first argument location in stack is close by + well known \( \text{good for variable length args} \)

"Mutual Exclusion"

Critical section problem (Silberschatz 6.2 & 6.3)

\[
\text{deposit( \text{sum} )} \\
\text{balance = balance + sum}
\]

"Race Condition"

Executing 2 threads of \text{deposit()} causes conflicting data

Serializable Memory Transactions:

- The last value for \( x \) written must be next value for \( x \) read; no multiple writes using old read value

User space \( \rightarrow \text{write()} \)

System Space \( \rightarrow \text{disable interrupts} \)

\( \rightarrow \text{read data} \)

\( \rightarrow \text{enable interrupts} \)

Kernel \( \rightarrow \text{IRQ} \)

Hardware

Disabling \text{IRQ}s doesn't work on multiple cores; \text{SMP} doesn't need \text{IRQ}s to run multiple processes.

"Petersen's Algorithm for Mutual Exclusion"

Two threads = 0 + 1

Variables = int flag[2], turn
(A) flag[i,j] = 1
(B) turn = ¬i
(C) while turn == ¬i and flag[¬i]
    Wait
(D) critical section
    flag[i] = 0
(Back to A)

Mutex: (Peterson's)
- Assume Th0 enters first
  turn = 0 or ¬flag[i,j]
  flag[0] or flag[0]
  turn = 0

Progress: (Peterson's)
- Both threads can't spin in wait b/c turn is either 0 or 1

Note: Peterson's not used anymore b/c assumes one operation steps before another stesp more than 2 threads, etc.

"Spin Lock"

Any read before T sees M0; Any read after T sees R

Peterson's Alg only works for 2 threads, but provides:
- Mutual Exclusions
- Progress
- Waiting threads get time
  - after thread exits, there is bounded time before next thread enters
  - Bounded Waiting
    - if thread is waiting # of other threads, that thread is bounded
    - No starvation

Bounded Waiting: (Peterson's)
- Th1 wait; 0 in CS;
  turn = 0
- Th1 wait; 0 in CS;
  turn = 1

* Worst case: Th can complete CS & go once more
- Usually requires HW lock on memory

- Ex Asm: mov 1, eax
  loop: xchg eax, lock
  test eax, lock
  jmp (n3), loop

- Ex Unlock: mov 0, lock

- Spinlocks usually used in OS kernels, not user-level code. B/c 2 threads scheduled on same CPU would sit and spin until switched out.