Chapter 19 – Parallel Processing Part IV

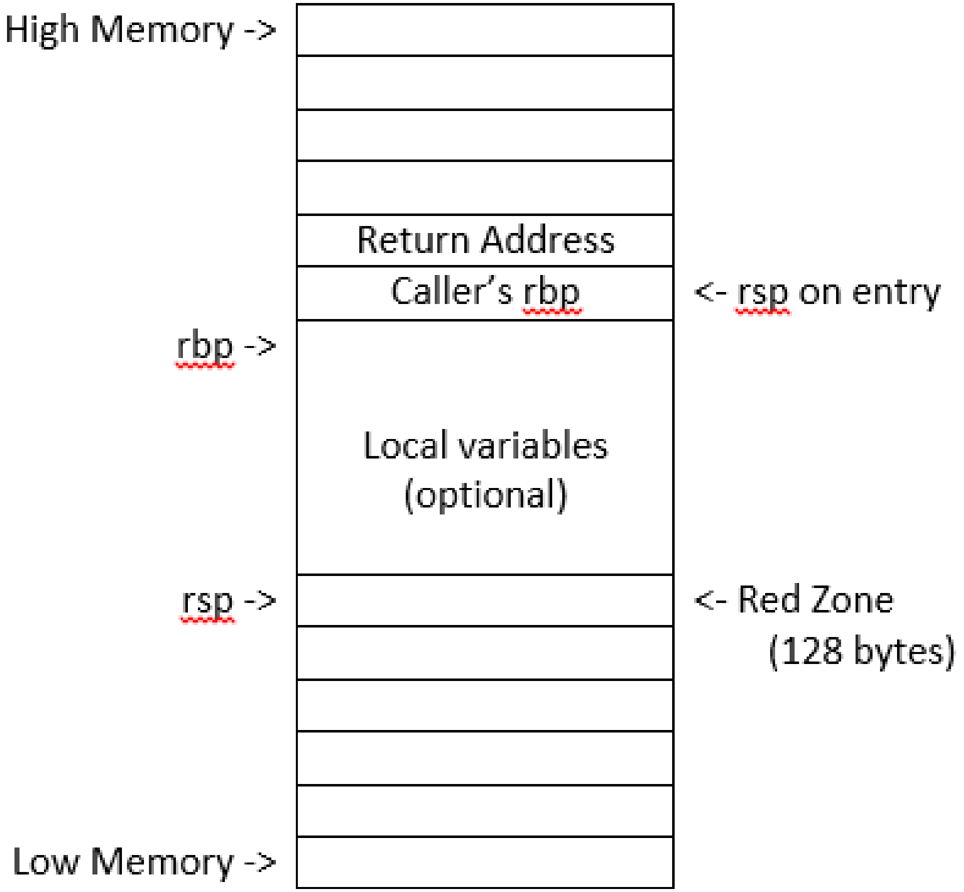
CPSC 240-09

John Overton

Review

More Parallel Processing…

- The Red Zone: An area of 128 bytes that is before the rsp pointer (remember, the rsp grows down)




```
_start:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 0x10 ; allocate local space
        …
```

```
;  Allocating local variables: 2 qword sized variables
var1        equu        -0x8
var2        equ         -0x10

; using local variables:
            mov         qword [rbp+var1], rax  ; save rax
            mov         rbx, qword [rbp+var2]
```

Parallel Processing – Atomic instructions in x86_64

Atomicity in x86_64

- The lock prefix can be used in front of several different instructions to cause the instruction to be an atomic operation. Can be used in front of:
  - `ADD, SUB, INC, DEC, NEG, NOT, OR, AND, CMPXCHG, 10 others`
- We could use lock in front of an add instruction in our Assignment #9 where each thread added its nbrFactors to a grandTotal variable (instead of having an array) –

```
;-- r11 is nbrFactors
;mov [thread_total+(r8*8)], r11
lock add qword [grand_total], r11
```

- This is where I demo some code that uses lock in front of an inc instruction

Compare-and-Swap Method of Atomicity

Using the cmpxchg instruction with the lock prefix is a true compare-and-swap atomic operation

And example of cmpxchg:

```
lock cmpxchg  qword [lockword], r8
```

- Compares the contents of rax with the contents of lockword
  - If equal, then the contents of r8 is copied to lockword.
  - If not equal, then the contents of lockword are copied to rax to get ready to retry the cmpxchg instruction
- The lock prefix makes the instruction atomic

Compare-and-Swap Method of Atomicity – Example: spinlock

```
mutex   dq      0
        ...
        mov     rdi, mutex
        call    spinlock
        ...

;-------------------------------------------------------------
; Lock a mutex.  On entry, rdi will point to qword lock word.
; This routine will not return until it "owns" the mutex.
; The mutex will only contain 0 (unlocked) or 1 (locked)
spinlock:
        mov     rax, 0          ; What we expect in the mutex
        mov     r8, 1           ; What we want in the mutex
lockloop:
        lock                     ; lock prefix to cmpxchg
        cmpxchg  qword [rdi], r8  ; if equal, r8->[rdi]

        jne     lockloop         ; if not equal, [rdi]->rax
        ret

;-------------------------------------------------------------
; Unlock a mutex.  Assume we were the thread that locked it.
; On entry, rdi will point to qword lock word.
spinunlk:
        mov     r8, 0
        xchg    r8, qword [rdi]  ; no need to use lock - implied.
        ret
```

Phase 3 - Results running with NUMBER_TO_FACTOR = 2 Billion

Total factors of 2000000000 = 110

With 1 thread:

real 0m14.700s

user 0m14.656s

real = Elapsed time
User = time processor(s) spend working

With 4 threads:

real 0m4.373s

user 0m17.188s ← Increase in overhead

With 8 threads:

real 0m2.745s ← dramatic decrease in elapsed time

user 0m20.234s

With 16 threads:

real 0m2.708s ← no decrease in elapsed time because I only have 8 processors

user 0m20.234s

Assignment – Phase 3 – with "lock add"

In Phase 3, you will replace your Hello World part of your threads with an equation to factor a number. That is, given a number, count the number of numbers that can be wholly divided into that number:

```
NUMBER_TO_FACTOR    equ  10000000000
RANGE_EACH_THREAD   equ  NUMBER_TO_FACTOR / MAX_THREADS

from = (threadIndex * RANGE_EACH_THREAD) + 1
to = from + RANGE_EACH_THREAD

for( i = from; i < to; i++ ) {
     if ( number % i == 0 )
          nbrFactors++
}
```

Assignment – Phase 3 – with "lock add"

In your thread function, it's easier to use registers for all of the variables since each thread gets its own registers.

On Tuesday, we talked about a solution that involved each thread saving nbrFactors in its own element in an array.

```
threadTotals[threadIndex] = nbrFactors
```

We can now use the "lock add" solution to add nbrFactors to a grand total without worrying about each thread causing a race condition with the grand total variable