CS202-ØØ2: Operating Systems

Two Days Ago ... Ended with History

Today

- Processes
- Process Execution
  - Stack
  - Function Calls — Stack Frames
  - System Calls
- Other Traps

Maybe went through some of this in CS0.
- Review it again
- Agree on terminology
- Right add details

Only Considering Processor + Memory

In Questions
10 QUESTIONS

Q1. WHAT DOES THE PROCESSOR LOOK LIKE TO A PROCESS?
   → TODAY!

Q2. HOW DOES THE KERNEL IMPLEMENT & MANAGE PROCESSES?
   → LEAVE THIS FOR ANOTHER DAY

But why?

- CONVENIENCE

- EFFICIENCY

VSCode → gcc → Music → ...

- Waiting for Keystroke
- Waiting for bits from disk
Q1. a) What does the processor look like to a process?
   b) What does memory look like to a process?

Processor ("core") & Memory: An Abstract View

Registers

Exec Units

CACHE

MMU

Memory

Exec units: Have no state. Must read both instruction & data from elsewhere.

Registers: A set (of known size, determined by processor designer) of fixed-size holders of state (in this class 64 bits).
This one is special

CONSISTENT + FAST ACCESS

MEMORY: More plentiful amount decided when building a computer

But slower access (O(ns – 100ns))

```c
int p = fork();
if (p != 0) {
    printf("I'm yellow");
}
```
Each process gets

- Its own set of registers
- Its own view of memory
- Some other fiddly metadata
**IMPORTANT**: All information used by the processor is either in registers or memory.
Somewhat more explicit in assembly

Simple assembly

% %rax <-- Registers written with %
$ $22 <-- Numbers (immediate) written with $
movq source, destination

subq x, y \Rightarrow y = y - x

pushq \%rax

8 - 8 = 0

movq \%rax, \%rbp
"EPILOG"

Ok! Now let us understand the bug in g'

```c
uint64_t * q;

38 uint64_t g(uint64_t a)
39 {
40     uint64_t x = 2*a;
41     q = &x;  // <-- THIS IS AN ERROR (AKA BUG)
42     return x;
43 }
```

...  

\( x = g(*ptr) ; \)

...  

Calling **Convention**

1. How to pass arguments

2. Where does the return value go
3. Who saves what registers?

**Caller saved / volatile**

%rax, rcx, rdx, r8, r9, r10, r11

**Callee saved**

rbx, rbp, rdi, rsi, rsp, r12–15

```
movq %rax, -8(%rbp)
movq -8(%rbp), %r10
addq $1, %r10
movq %r10, %rax
movq %rbp, %rsp
popq %rbp
ret
```

f epilog

Syscalls

write(...)

? Process

Kernel
/* CS202 -- handout 1
*   compile and run this code with:
*   $ gcc -g -Wall -o example example.c
*   $ ./example
*   examine its assembly with:
*   $ gcc -O0 -S example.c
*   $ [editor] example.s
*/

#include <stdio.h>
#include <stdint.h>

uint64_t f(uint64_t* ptr);
uint64_t g(uint64_t a);
uint64_t* q;

int main(void)
{
    uint64_t x = 0;
    uint64_t arg = 8;
    x = f(&arg);
    printf("x: %lu\n", x);
    printf("dereference q: %lu\n", *q);
    return 0;
}

uint64_t f(uint64_t* ptr)
{
    uint64_t x = 0;
    x = g(*ptr);
    return x + 1;
}

uint64_t g(uint64_t a)
{
    uint64_t x = 2*a;
    q = &x; // <-- THIS IS AN ERROR (AKA BUG)
    return x;
}

2. A look at the assembly...
To see the assembly code that the C compiler (gcc) produces:
$ gcc -O0 -S example.c
(then look at example.s.)
NOTE: what we show below is not exactly what gcc produces. We have
simplified, omitted, and modified certain things.

main:
    pushq %rbp        # prologue: store caller's frame pointer
    movq %rsp, %rbp   # prologue: set frame pointer for new frame
    subq $16, %rsp    # prologue: make stack space
    movq $0, -8(%rbp) # x = 0 (x lives at address rbp - 8)
    movq $8, -16(%rbp) # arg = 8 (arg lives at address rbp - 16)
    leaq -16(%rbp), %rdi  # load the address of (rbp-16) into %rdi
    # this implements "get ready to pass (&arg) to f"
    call f             # invoke f
    movq %rax, -8(%rbp) # x = (return value of f)
    # eliding the rest of main()

f:
    pushq %rbp        # prologue: store caller's frame pointer
    movq %rsp, %rbp   # prologue: set frame pointer for new frame
    subq $16, %rsp    # prologue: make stack space
    movq $0, -8(%rbp) # x = 0 (x lives at address rbp - 8)
    movq $8, -16(%rbp) # arg = 8 (arg lives at address rbp - 16)
    movq -24(%rbp), %r8 # move 'ptr' to %r8
    movq (%r8), %r9   # dereference 'ptr' and save value to %r9
    movq %r9, %rdi    # Move the value of *ptr to rdi,
    # so we can call g
    call g            # invoke g
    movq %rax, -8(%rbp) # x = (return value of g)
    addq $1, %r10     # compute x + 1, part I
    movq %r10, %rax   # Get ready to return x + 1
    movq %rbp, %rsp   # epilogue: undo stack frame
    ret                # return

g:
    pushq %rbp        # prologue: store caller's frame pointer
    movq %rsp, %rbp   # prologue: set frame pointer for new frame
    subq $0x8, %rsp   # prologue: make stack space
    ....
    movq %rbp, %rsp   # epilogue: undo stack frame
    popq %rbp         # epilogue: restore frame pointer from caller
    ret                # return