```
36
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
53
```

Borrowed from xv6: https://github.com/mit-pdos/xv6-public/blob/eeb7b415dbcb12cc362d0783e41c3d1f44066b17/proc.h

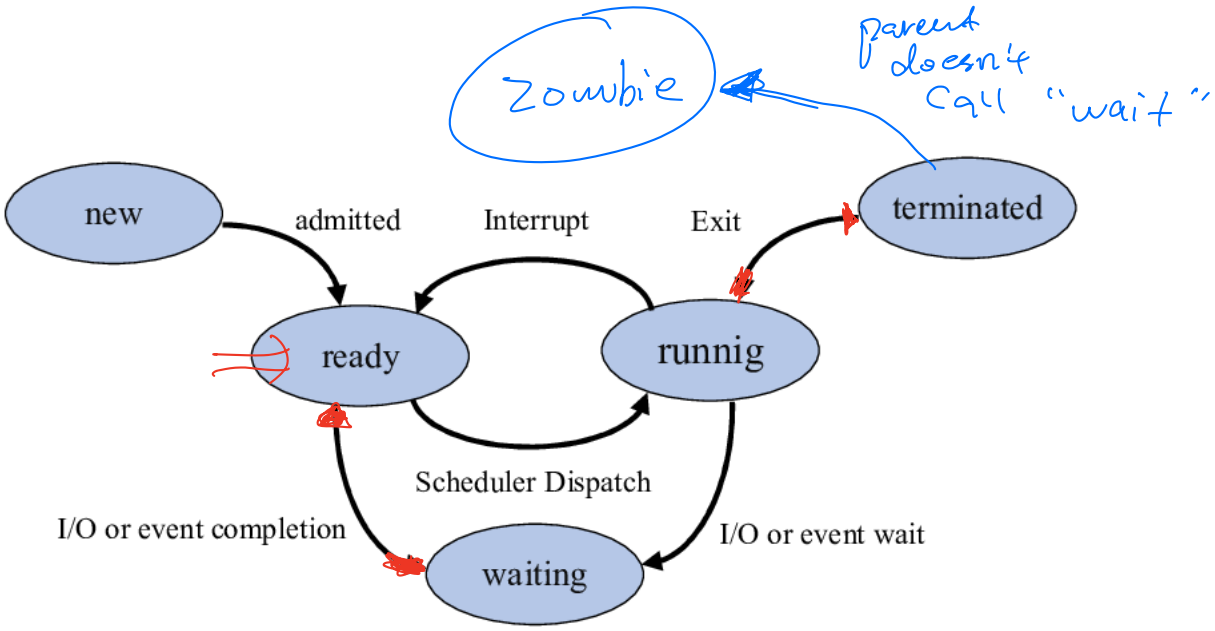
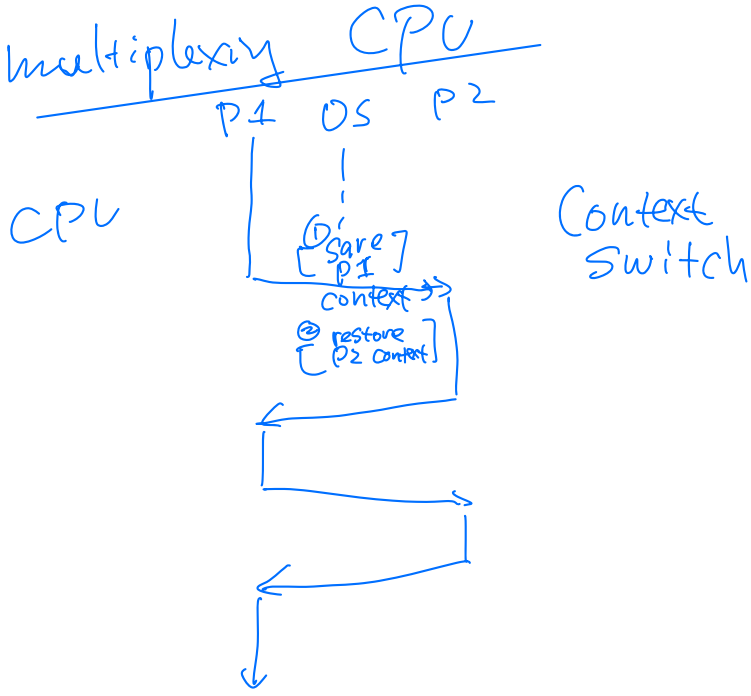
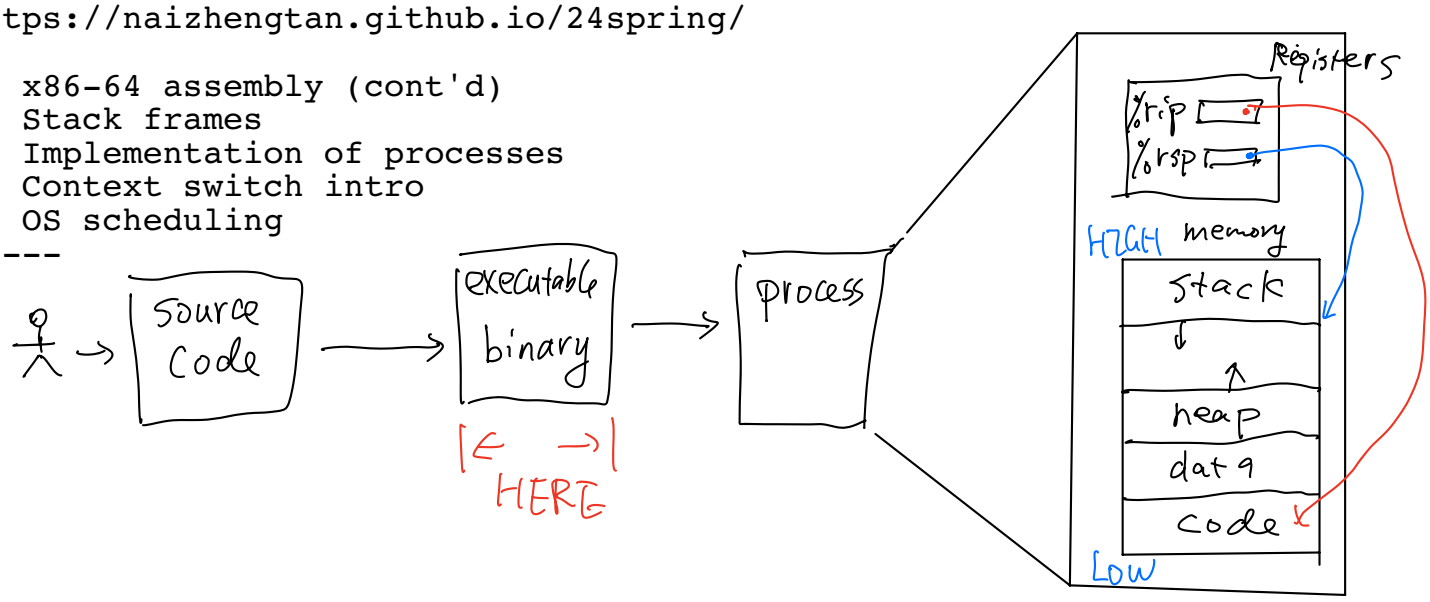


Fig13 from "Priority based round robin (PBRR) CPU scheduling algorithm"



Week 5.a
CS3650
02/05 2024
https://naizhengtan.github.io/24spring/

1. x86-64 assembly (cont'd)
2. Stack frames
3. Implementation of processes
4. Context switch intro
5. OS scheduling

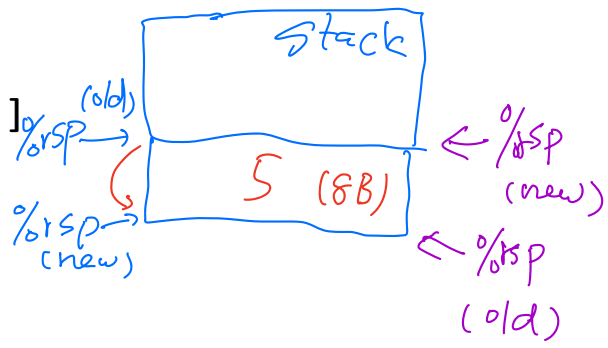


Crash course of x86-64 assembly

int x = 5;

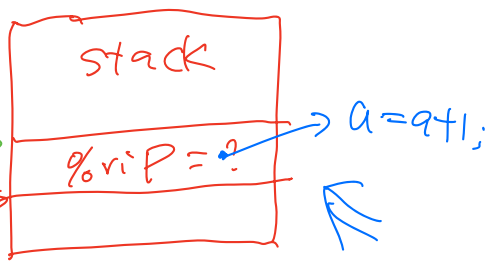
* movq PLACE1, PLACE2
  1. Registers
  2. addresses
  3. immediate

* pushq %rax [subq $8, %rsp
  movq %rax, (%rsp)]



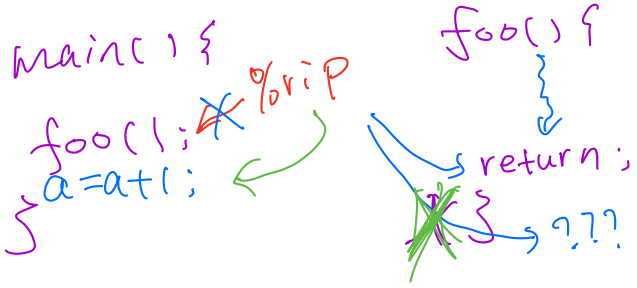
* popq %rax [movq (%rsp), %rax
  addq $8, %rsp]

* call 0x12345 [pushq %rip
  movq $0x12345, %rip]



foo

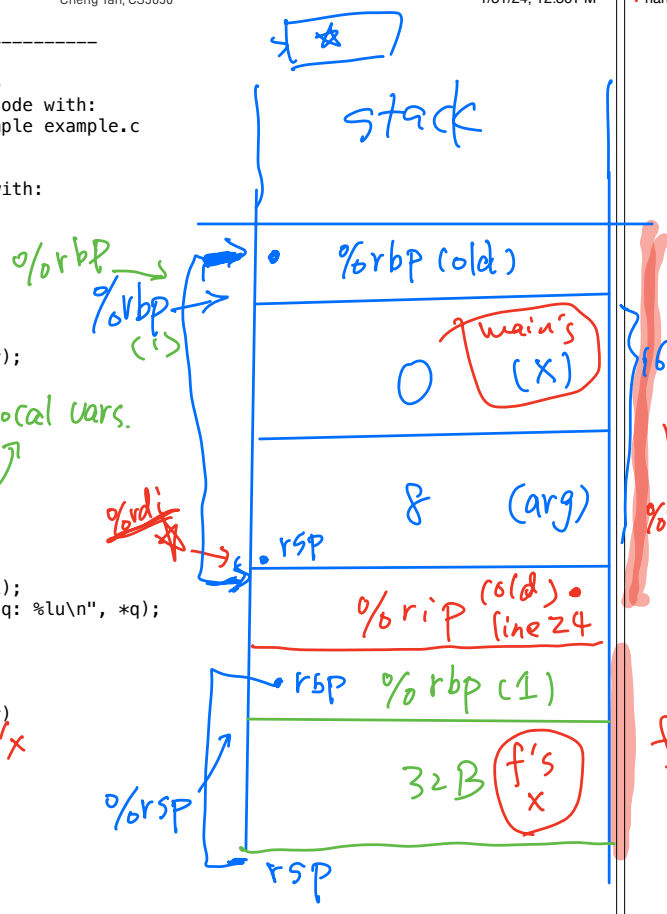
* ret [popq %rip]



- %rip : instruction ptr (next instr
- %rsp : stack ptr
- %rbp : base ptr


```
-------------[example.c]---------------

1    /* CS3650 -- handout w04b
2     * compile and run this code with:
3     * $ gcc -g -Wall -o example example.c
4     * $ ./example
5     *
6     * examine its assembly with:
7     * $ gcc -O0 -S example.c
8     * $ [editor] example.s
9     */
10
11    #include <stdio.h>
12    #include <stdint.h>
13
14    uint64_t f(uint64_t* ptr);
15    uint64_t g(uint64_t a);
16    uint64_t* q;
17
18    int main(void)
19    {
20        uint64_t x = 0;
21        uint64_t arg = 8;
22
23        x = f(&arg);
24
25        printf("x: %lu\n", x);
26        printf("dereference q: %lu\n", *q);
27
28        return 0;
29    }
30
31    uint64_t f(uint64_t* ptr)
32    {
33        uint64_t x = 0;
34        x = g(*ptr);
35        return x + 1;
36    }
37
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    }
```




```
-------------[as.txt (x86)]-------

1   2. A look at the assembly...
2
3   To see the assembly code that the C compiler (gcc) produces:
4   $ gcc -O0 -S example.c
5   (then look at example.s.)
6   NOTE: what we show below is not exactly what gcc produces. We have
7   simplified, omitted, and modified certain things.
8
9   main:
10      pushq %rbp              # prologue: store caller's frame pointer
11      movq %rsp, %rbp         # prologue: set frame pointer for new frame
12
13      subq $16, %rsp          # make stack space
14
15      movq $0, -8(%rbp)       # x = 0 (x lives at address rbp - 8)
16      movq $8, -16(%rbp)      # arg = 8 (arg lives at address rbp - 16)
17
18      leaq -16(%rbp), %rdi    # load the address of (rbp-16) into %rdi
19                              # this implements "get ready to pass (&arg)
20                              # to f"
21
22      call f                  # invoke f
23
24      movq %rax, -8(%rbp)     # x = (return value of f)
25
26      # eliding the rest of main()
27
28  f:
29      pushq %rbp              # prologue: store caller's frame pointer
30      movq %rsp, %rbp         # prologue: set frame pointer for new frame
31
32      subq $32, %rsp          # make stack space
33      movq %rdi, -24(%rbp)    # Move ptr to the stack
34                              # (ptr now lives at rbp - 24)
35      movq $0, -8(%rbp)       # x = 0 (x's address is rbp - 8)
36
37      movq -24(%rbp), %r8     # move 'ptr' to %r8
38      movq (%r8), %r9         # dereference 'ptr' and save value to %r9
39      movq %r9, %rdi          # Move the value of *ptr to rdi,
40                              # so we can call g
41
42      call g                  # invoke g
43
44      movq %rax, -8(%rbp)     # x = (return value of g)
45      movq -8(%rbp), %r10     # compute x + 1, part I
46      addq $1, %r10           # compute x + 1, part II
47      movq %r10, %rax         # Get ready to return x + 1
48
49      movq %rbp, %rsp         # epilogue: undo stack frame
50      popq %rbp               # epilogue: restore frame pointer from caller
51      ret                     # return
52
53  g:
54      pushq %rbp              # prologue: store caller's frame pointer
55      movq %rsp, %rbp         # prologue: set frame pointer for new frame
56
57      ....
58
59      movq %rbp, %rsp         # epilogue: undo stack frame
60      popq %rbp               # epilogue: restore frame pointer from caller
61      ret                     # return
```

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes

Borrowed from
https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf
Figure 3.4