

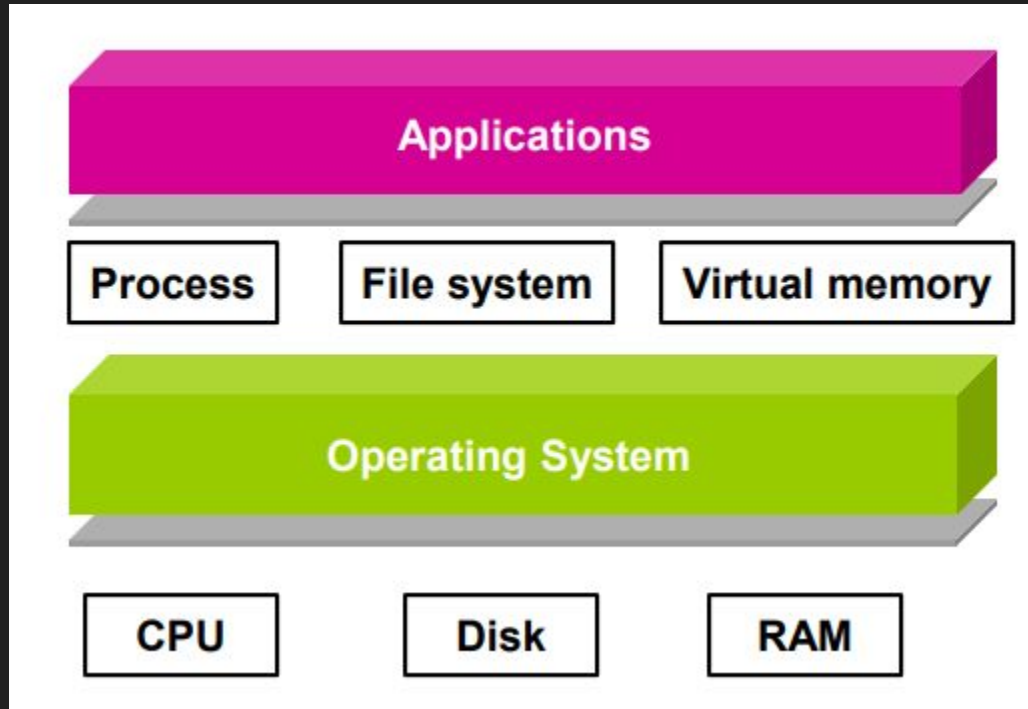
ILLINOIS TECH

College of Computing

CS 450 Operating Systems Memory Management & Midterm Review

Yue Duan

OS Abstractions

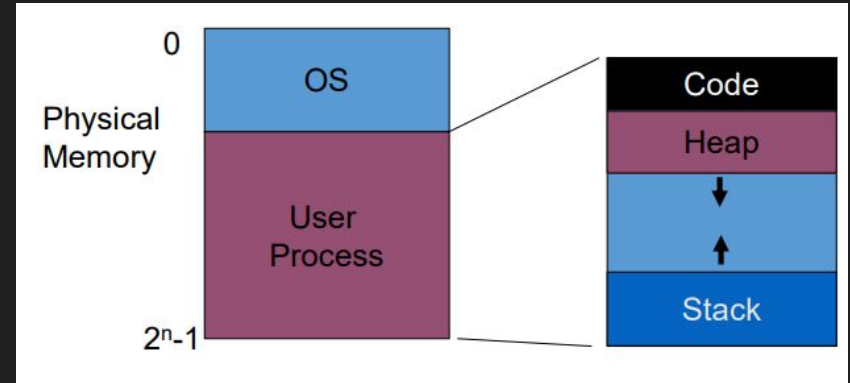


Virtualizing Resources

- Physical Reality: different processes/threads share the same hardware ==> need to multiplex
 - CPU (temporal)
 - Memory (spatial)
 - Disk and devices (later)
- Process: Abstraction to virtualize CPU
 - Give the **illusion** of **private** CPU (registers)
 - Use time-sharing in OS to switch between processes
- Memory?
 - Give the **illusion** of **private** memory

Motivation

- (Very) old days: Uniprogramming
 - Only one process existed at a time
 - OS was little more than a library occupying the beginning of the memory
 - Advantage:
 - Simplicity — No virtualization needed
 - Disadvantages:
 - Only one process runs at a time
 - Process can destroy OS



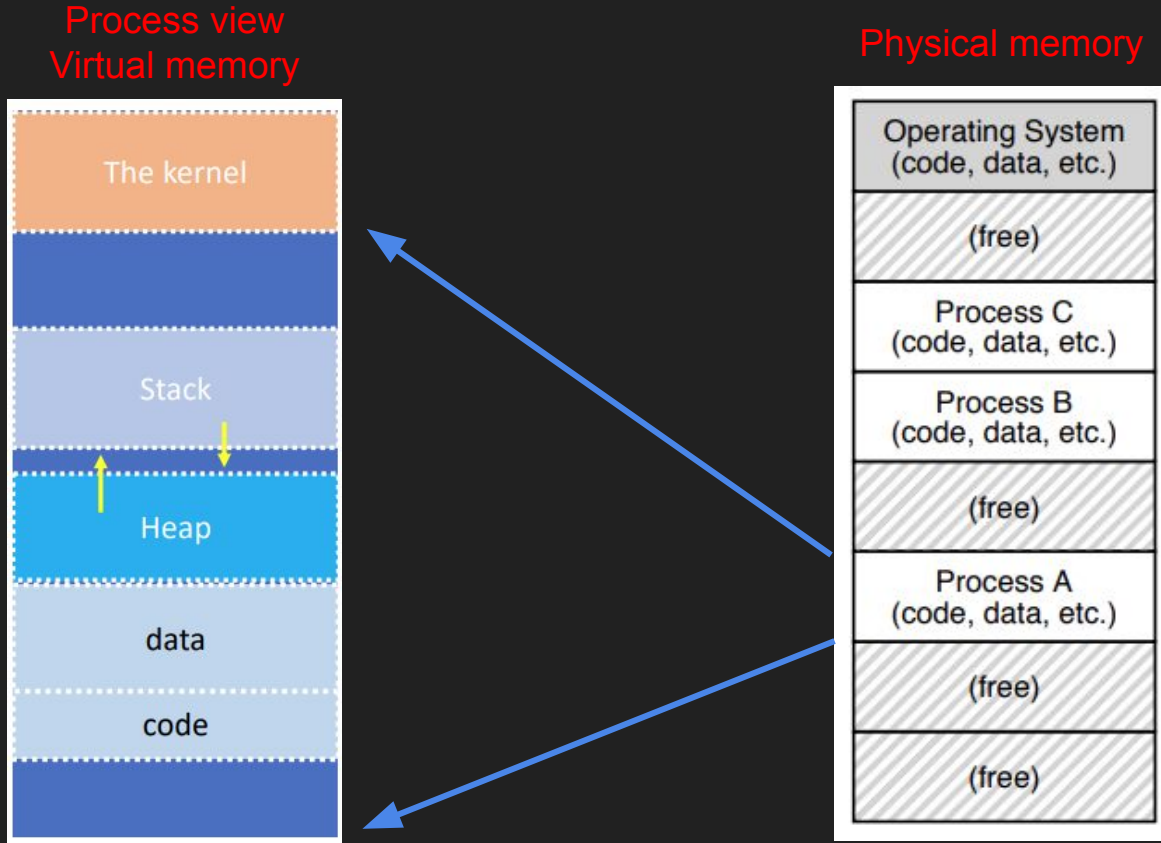
Motivation

- Multiprogramming goals
 - **Transparency**
 - Process is unaware of sharing
 - **Protection**
 - Cannot corrupt OS or other process memory
 - **Efficiency**
 - Do not waste memory or slow down processes
 - **Sharing**
 - Enable sharing between cooperating processes

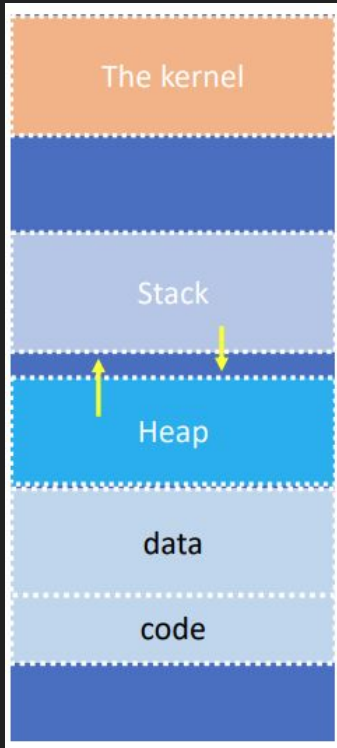
Abstraction: Address Space

- Address space
 - Each process' view of its own memory range
 - Set of addresses that map to bytes
- Problem
 - How can OS provide illusion of private address space to each process?
- Address space has static and dynamic components
 - Static: Code and some global variables
 - Dynamic: Stack and Heap

Abstraction: Address Space



Abstraction: Address Space



stack segment: grows downward; contains local variables, arguments to functions, return values, etc

heap segment: grows upward, contains malloc'd data, dynamic data structures

data segment: where global variables live

text segment: where instructions live

Stack Structure

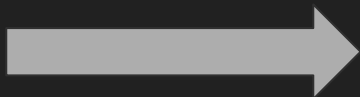
- What goes on stack?

```
void
bar(int a, int b)
{
    int x, y;

    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111, 222);
}
```

gcc -S -m32 try.c



- 32-bit machine:
 - ebp: base pointer
 - esp: stack pointer

```
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $555, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    addl     %edx, %eax
    movl     %eax, -8(%ebp)
    leave
    ret

foo:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $222, 4(%esp)
    movl     $111, (%esp)
    call     bar
    leave
    ret
```

Stack Structure

```
bar:      # ----- start of the function bar()
    pushl  %ebp      # save the incoming frame pointer
    movl   %esp, %ebp # set the frame pointer to the current top of stack
    subl   $16, %esp  # increase the stack by 16 bytes (stacks grow down)
    movl   $555, -4(%ebp) # x=555 a is located at [ebp-4]
    movl   12(%ebp), %eax # 12(%ebp) is [ebp+12], which is the second parameter
    movl   8(%ebp), %edx  # 8(%ebp) is [ebp+8], which is the first parameter
    addl   %edx, %eax    # add them
    movl   %eax, -8(%ebp) # store the result in y
    leave  #
    ret     #

foo:      # ----- start of the function foo()
    pushl  %ebp      # save the current frame pointer
    movl   %esp, %ebp # set the frame pointer to the current top of the stack
    subl   $8, %esp   # increase the stack by 8 bytes (stacks grow down)
    movl   $222, 4(%esp) # this is effectively pushing 222 on the stack
    movl   $111, (%esp) # this is effectively pushing 111 on the stack
    call   bar        # call = push the instruction pointer on the stack and branch to foo
    leave  # done
    ret     #
```

Stack Structure

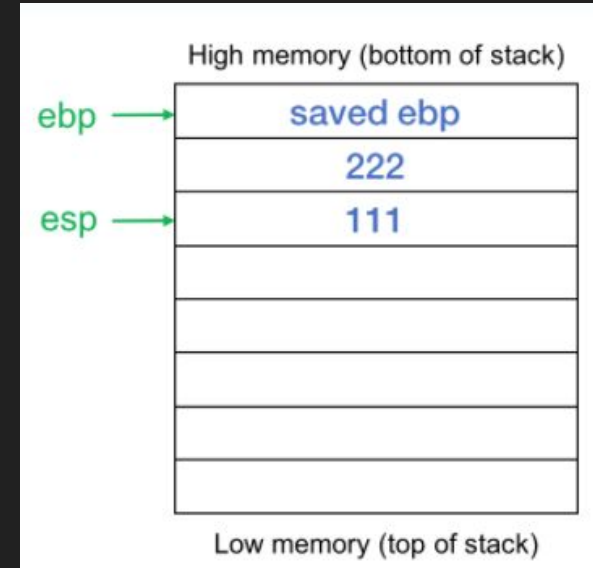
```
void
bar(int a, int b)
{
    int x, y;

    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111,222);
}
```

```
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $555, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    addl     %edx, %eax
    movl     %eax, -8(%ebp)
    leave
    ret

foo:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $222, 4(%esp)
    movl     $111, (%esp)
    call     bar
    leave
    ret
```



Stack Structure

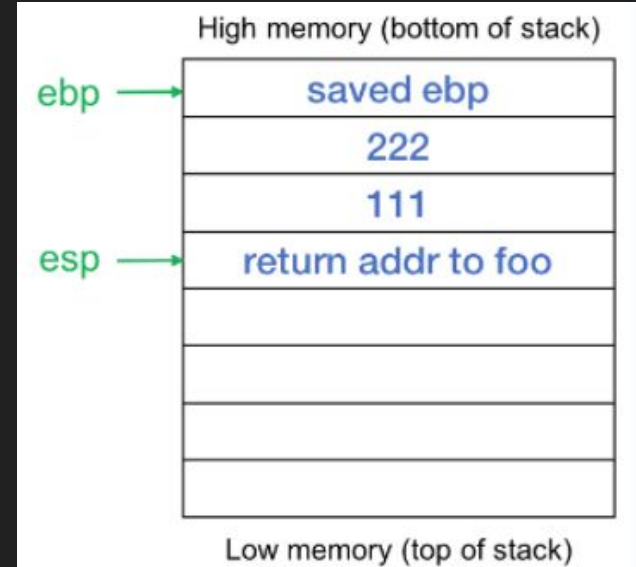
```
void
bar(int a, int b)
{
    int x, y;

    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111,222);
}
```

```
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $555, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    addl     %edx, %eax
    movl     %eax, -8(%ebp)
    leave
    ret

foo:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $222, 4(%esp)
    movl     $111, (%esp)
    call     bar
    leave
    ret
```




Stack Structure

```
void
bar(int a, int b)
{
    int x, y;

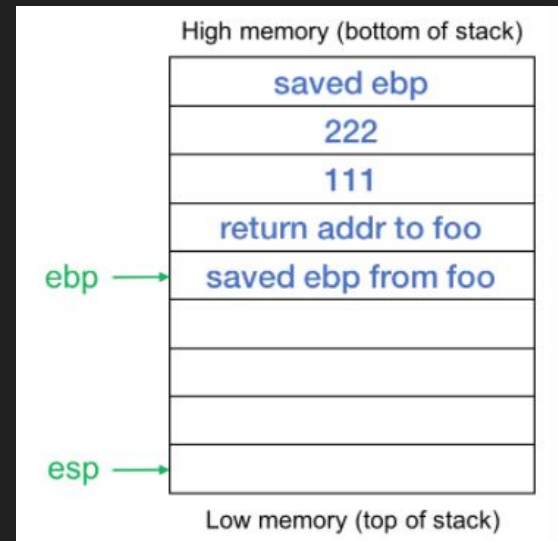
    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111,222);
}
```



```
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $555, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    addl     %edx, %eax
    movl     %eax, -8(%ebp)
    leave
    ret

foo:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $222, 4(%esp)
    movl     $111, (%esp)
    call     bar
    leave
    ret
```



Stack Structure

```
void
bar(int a, int b)
{
    int x, y;

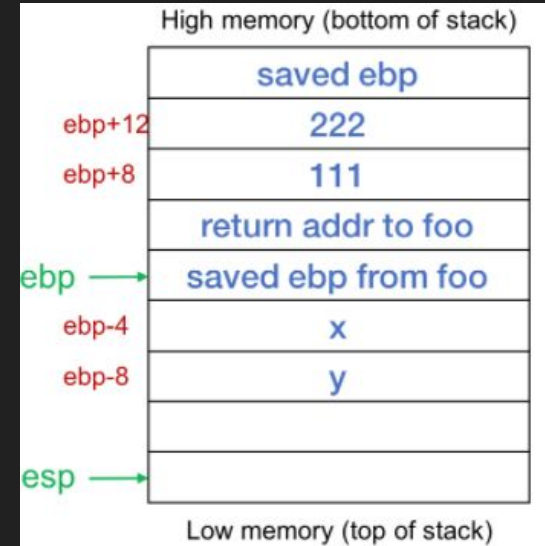
    x = 555;
    y = a+b;
}

void
foo(void) {
    bar(111,222);
}
```



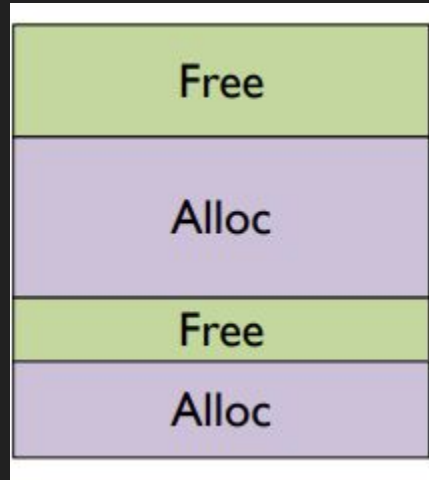
```
bar:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $16, %esp
    movl     $555, -4(%ebp)
    movl     12(%ebp), %eax
    movl     8(%ebp), %edx
    addl     %edx, %eax
    movl     %eax, -8(%ebp)
    leave
    ret

foo:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $222, 4(%esp)
    movl     $111, (%esp)
    call     bar
    leave
    ret
```



Heap Organization

- Allocate from any random location: malloc(), new() etc.
- Heap memory consists of allocated and free areas
- Order of allocation and free is unpredictable



Midterm Reviews

- Process
 - What is a process?
 - What resource does it virtualize?
 - What is the difference between a process and a program?
 - What is contained in a process?
- Process Control Blocks (PCBs)
 - What information does it contain?
 - How is it used in a context switch?

Midterm Reviews

- Process life cycles
 - What are process states?
 - What is the process life cycle graph?
 - When does a process change state?
 - How does the OS use queues to keep track of processes?
- Process manipulation
 - What does `fork()` on Unix do?
 - What does it mean for it to “return twice”?
 - What does `exec()` on Unix do?
 - How is it different from `fork`?

Midterm Reviews

- Threads
 - What is a thread?
 - What is the difference between a thread and a process?
 - How are they related?
 - Why are threads useful?
 - What is the difference between user-level and kernel-level threads?
 - What are the advantages/disadvantages of one over another?
 - Thread control blocks
 - What is the difference between TCB and PCB?

Midterm Reviews

- CPU scheduling
 - What is context switch?
 - What is the difference between non-preemptive scheduling and preemptive thread scheduling?
 - Different scheduling algorithms
 - FIFO, SJF, RR, Priority-based scheduling

Midterm Reviews

- Synchronization
 - Why do we need synchronization?
 - Coordinate access to shared data structures
 - Coordinate thread/process execution
- Mutual exclusion
 - What is mutual exclusion?
 - What is a critical section?
 - What guarantees do critical sections provide?
 - What are the requirements of critical sections?
 - How does mutual exclusion relate to critical sections?
 - What are the mechanisms for building critical sections?
 - Locks, semaphores

Midterm Reviews

- Locks
 - What does Acquire do?
 - What does Release do?
 - What does it mean for Acquire/Release to be atomic?
 - How can locks be implemented?
 - Spinlocks
 - Disable/enable interrupts
 - How does test-and-set work?
 - What kind of lock does it implement?
 - What are the limitations of using spinlocks, interrupts?
 - Inefficient, interrupts turned off too long

Midterm Reviews

- Condition variables
 - What is condition variable?
 - What are the APIs?
 - What are CV rules of thumb?
 - Producer/Consumer problem

Midterm Reviews

- Semaphores and condition variables
 - What is a semaphore?
 - What does Wait/P/Decrement do?
 - What does Signal/V/Increment do?
 - How does a semaphore differ from a lock or a condition variable?
 - What is the difference between a binary semaphore and a general semaphore?
 - Using semaphores to solve synchronization problems
 - Readers/Writers problem
 - Bounded Buffers problem

Midterm Reviews

- Deadlock
 - What is deadlock?
 - Deadlock happens when processes are waiting on each other and cannot make progress
 - What are the conditions for deadlock?
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
 - What is livelock?

Midterm Reviews

- Describe what happens in the computer when the timer raises an interrupt.
 - CPU receives the interrupt, pauses the current execution, switch to the kernel mode, save the CPU registers.
 - It further reads interrupt vector table, branches into the timer interrupt handler (lots of things might happen here, like scheduling, waiting up sleeping jobs,etc.)
 - After completion, restores saved state from stack and resume the paused execution.

Midterm Reviews

- Two players play table tennis. The first player serves, the second player hits the ball back to first player, and first player hits back, and with a 20% probability, one player may drop the ball.
- Write a program to simulate this situation. Each player runs in one thread. A possible output looks like this:
 - Player 1 serves.
 - Player 2 hits it back to Player 1.
 - Player 1 hits it back to Player 2.
 - Player 2 drops the ball.

```
// semaphore initializations
sem_init(&sem[0], 0, 1);
sem_init(&sem[1], 0, 0);

void Player(void *p) {
    int i = (int)p;
    while(1) {
        sem_wait(sem[(i+1)%2]);
        r = get_random(0,1);
        if (r < 0.2) {
            printf("Player %d drops the ball.\n", i+1);
            break;
        }
        printf("Player %d hits back to Player %d.\n", i+1, ((i+1)%2)+1);
        sem_post(sem[i]);
    }
}
```

THANK YOU!