Operating Systems Concepts

Address Spaces Memory API

CS 4375, Fall 2025

Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu

October 1, 2025

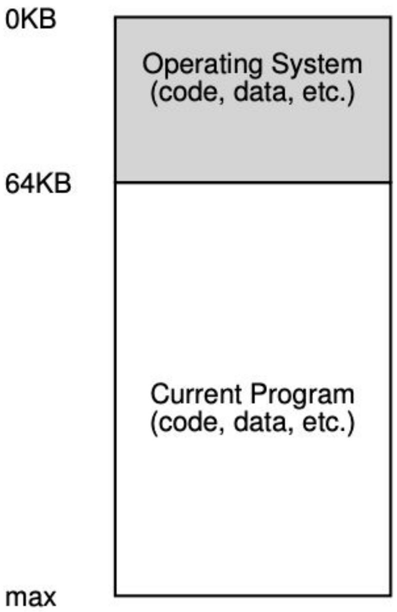
Summary

- CPU Scheduling:
  - Recap
  - Estimating CPU Burst Time
  - Multilevel Feedback Queue Scheduler
  - 4.4BSD Priority Based Scheduler
  - Proportional Share Scheduling

Agenda

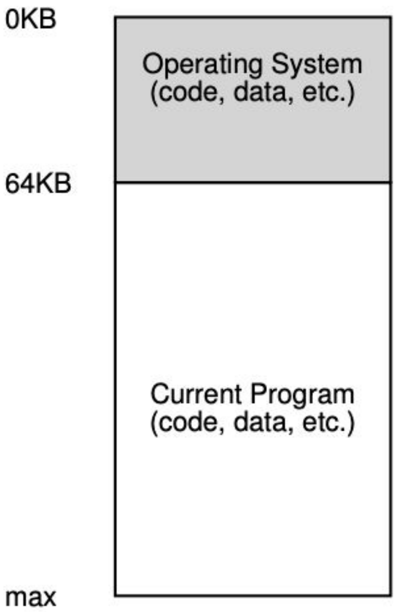
- Address Space
- Virtual memory
- Linux Memory APIs
    - malloc(), free()
- Example Code

Early Systems



More of a Library with a set of routines/function

Early Systems

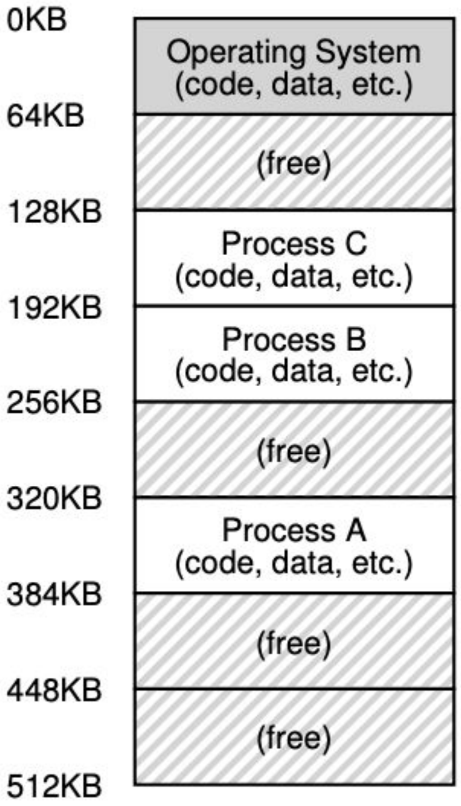


We wanted more from a machine

Multiprogramming

High cost in moving program in and out of memory

Multiprogramming



Keep the programs in memory and switch between the program at runtime

Share the address space
No isolation
Not transparency

Why do we need **virtualization**?

To enable the OS build abstractions of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory.

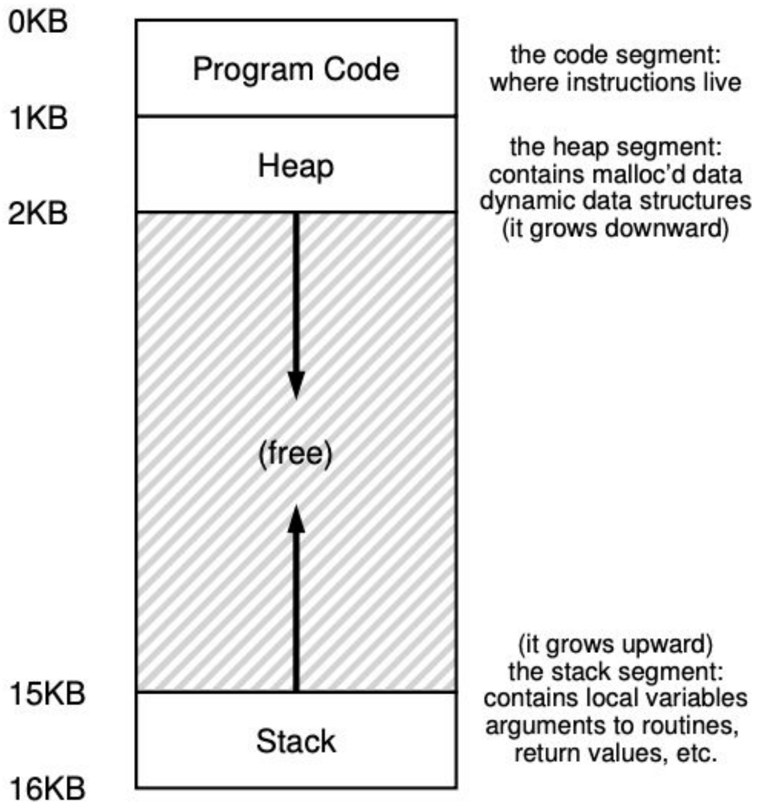
OS virtualization goals

- Transparency
  - Virtualization memory space should invisible to a program
  - Behaves as if it has its own private physical memory
- Efficiency
  - Strive to make virtualization as efficient as possible (time & space)
  - Hardware support required
- Protection/isolation
  - Protect one program from another
  - No access outside its own memory space
  - read/write/execute

Program Address Space

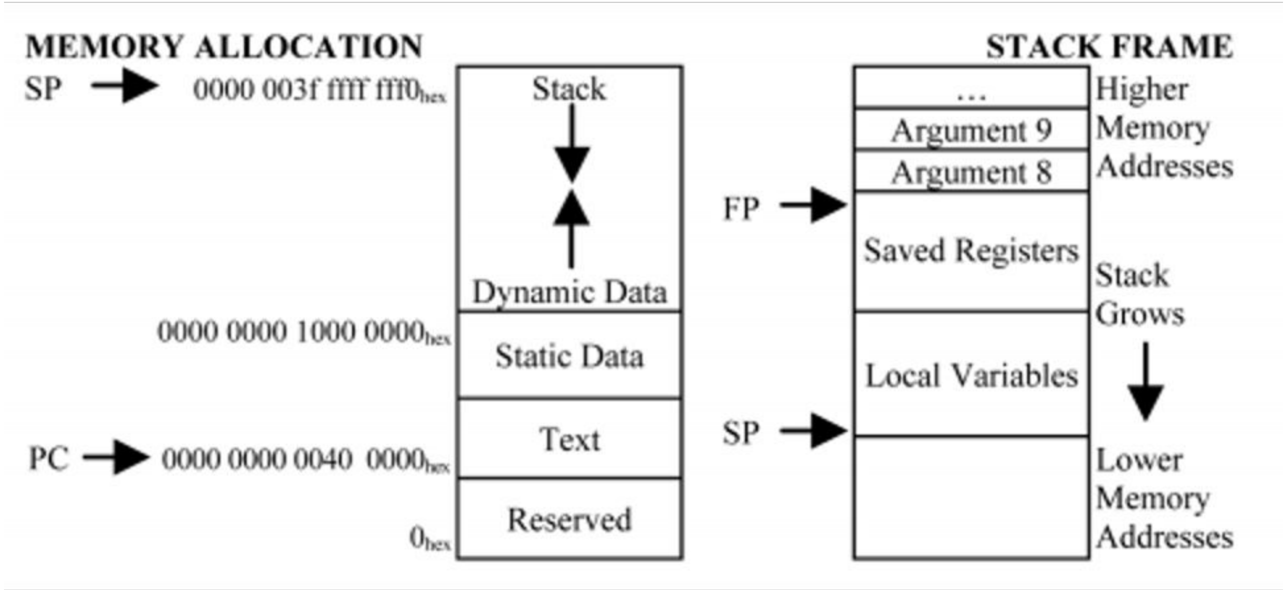
- Program Code
    - A set of instructions
    - Static
- Stack
    - Data of a program
    - Statically defined data
        - *Int a, char array[50]*
- Heap
    - Data of a program
    - Dynamic data request by a user program
        - *char array[] = malloc(50)*

Program Address Space



Program Address Space (RISC-V)

- RISC-V Linux 39-bit Address Space



Example code

Compile and run on linux/macOS
Code & Makefile in class website

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf("Location of code: %p\n", (void *) main);
    printf("Location of heap: %p\n", (void *) malloc(1));
    int stack_var = 0x42;
    printf("Location of stack: %p\n", (void *) &stack_var);
    printf("Value of stack variable: 0x%x\n", stack_var);
    return 0;
}
```

address_map.c

Example code

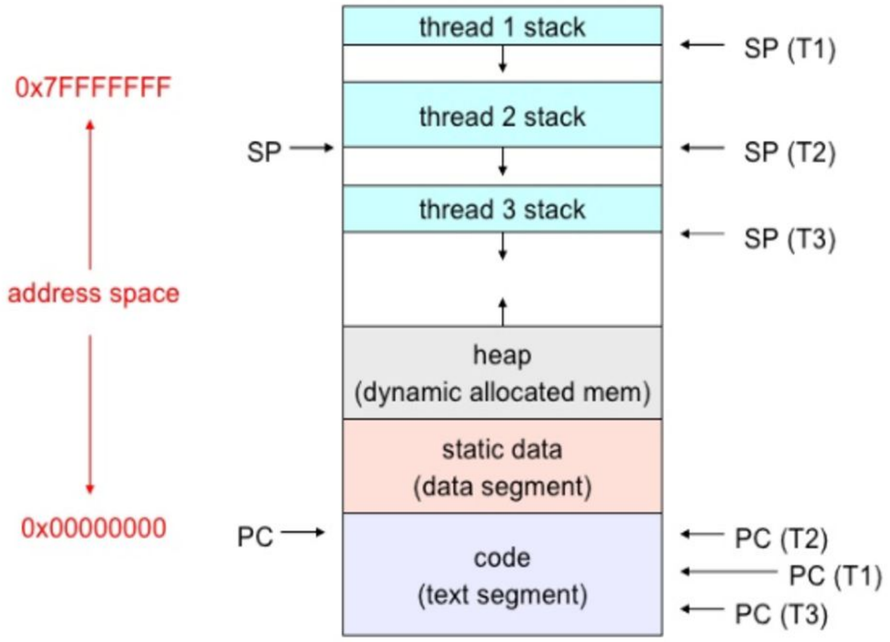
- Code > heap > stack
- At every run the addresses are different
    - Why?
    - Read about what is ASLR
- All memory is virtual memory not physical
    - Only the OS/kernel knows which physical memory is mapped to these addresses

```
marmanuzzaman@CCS31022-184542 memoryAPI % ./a.out
Location of code: 0x102b68460
Location of heap: 0x121e06000
Location of stack: 0x16d296dcc
Value of stack variable: 0x42
```

Output

Multithreaded Address Space

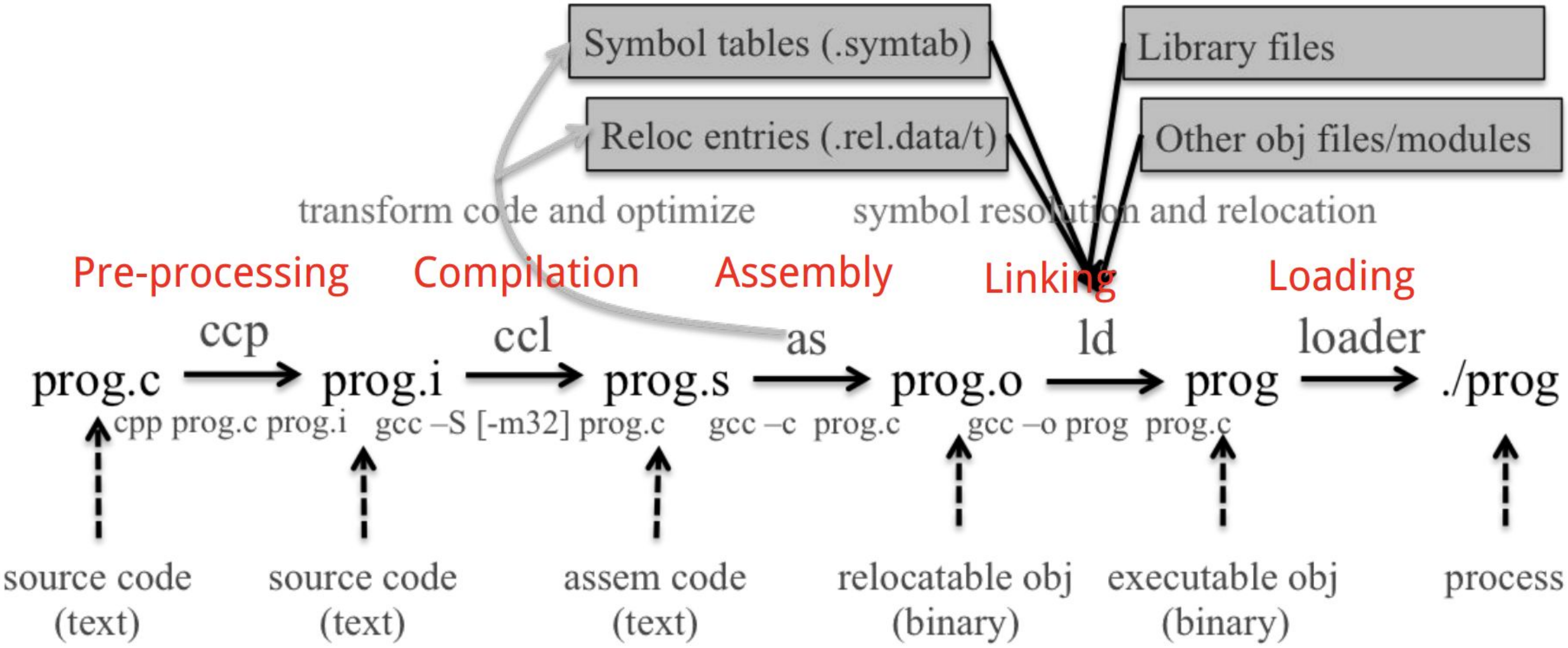
- Threads in Linux share the same address space but each thread has its own stack
- Each thread has its own register values.
- xv6 does not have threads in this sense



Linux C Memory API

- malloc() and free()
  - These are library calls, not system calls.
  - They manipulate memory in the user's heap space.
  - They may call a system call (e.g., brk(), sbrk(), mmap()) to grow or shrink heap space.
- See malloc and free man page
  - `man malloc`

From a C program to a process




```
gcc -save-temps file_name.c
```

Investigate program binary

- `objdump`
- `gdb`
- `otool`
- `strace`
- `ltrace`
- ...

Investigate address space

- `address_map.c`
- `memory-user.c`
- `memory-user2.c`
- **`pmap (Linux)`**
- **`vmmap (macOS)`**

Code and Makefile from class website

Question

- Compile C file to generate temporary files
    - Do you find anything interesting?
- How does the program memory change during sleep of the programs?
- Do you see any different when the program is allocating different sized memory in heap?

Announcement

- Homework 2
    - Due on Monday October 6th, 11.59 PM