Foundations of Programming Languages

The UNIX Run-Time System

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

17. Oktober 2014

Memory and Multi-Processing

Modern computers run multiple processes in parallel

Process 1

```
li $t0, 0xc00000808
li $t1, 1
sd $t1, 0($t0)
```

Process 2

```
li $t0, 0xc00000808
li $t1, 0x1337
sd $t1, 0($t0)
```

Memory and Multi-Processing

Modern computers run multiple processes in parallel

Process 1

```
li $t0, 0xc00000808
li $t1, 1
sd $t1, 0($t0)
```

Process 2

```
li $t0, 0xc00000808
li $t1, 0x1337
sd $t1, 0($t0)
```

- *Virtual memory*: Memory addresses mean different things for different processes
- Processes don't interfere

x86-64 memory addresses

- x86-64 uses 64-bit memory addresses
- Only lowest 48 bits are actually used
- Processes can decide how to use (‘map’) these

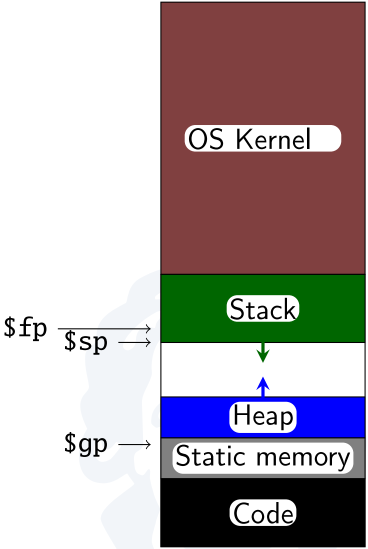
x86-64 memory addresses

- x86-64 uses 64-bit memory addresses
- Only lowest 48 bits are actually used
- Processes can decide how to use (‘map’) these
- At program start:
  - *Loader* allocates some addresses
  - Maps addresses to physical memory
  - Loads code, data into memory
  - Jumps into loaded code to start execution

Conventional memory layout in x86-64/Linux

Default allocation at program start:

- Operating system memory: not accessible to user-space code
- Stack: function calls, some temporary allocation
- Heap: temporary allocation
- Static memory: ‘global’ memory
- **Code** (also known as **text**): machine code



Layout requested by OS loader

Static Memory

- **Used for**:
  - Global variables (e.g., in C)
  - Constants (e.g., literal strings)
- **Size of region**:
  - fixed by loader
- **Access via**:
  - $gp register points to this region

Stack Memory

- **Used for**:
  - Local variables
  - Function calls, parameters

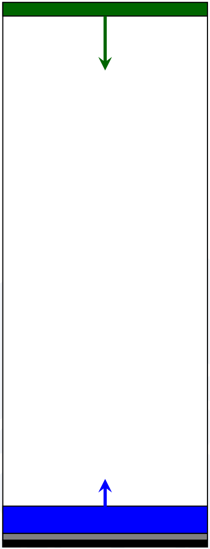
    ...
- **Size of region**:
  - On x86, stack begins at *top* of address space (by convention)
  - Grown automatically by operating system
- **Access via**:
  - `$sp` register points to lowest *allocated* byte on stack
  - `$fp` points into usable portion of stack
- **Usage** (e.g., need b bytes):
  - Lower `$sp` by b
  - Use region from `$sp` to `$sp` $+ b$
  - Increase `$sp` by b when done

Heap Memory

- **Used for**:
  - ‘catch-all’ when static/stack memory don’t suffice
- **Region size**:
  - Arbitrary; grown on demand (explicit requests)
- **Access via**:
  - Keep pointers around after allocation
- **Usage**: Process must *manage* heap:
  - Deallocate unused memory
  - Search for unused space on allocation
  - Grow heap (call operating system) if needed
  - Defragment memory (optional, not in C/C++)

Address Space Conventions

- Conventions simplify interaction with remainder of system
- Address space leaves *substantial* space for custom memory usage
  - Example here: we have mapped about 14 TiB
- Programs can freely allocate addresses for their own purposes
- Address space used e.g. by:
  - File access
  - Dynamic library loader
  - Threads



Summary

- Each process has its own address space
  - No interference with other processes
  - Can allocate ('map') new regions
- Conventional regions (mostly pre-allocated by loader):
  - **Code** ('`.text`'): executable code
  - **Static memory**: fixed-size read/write memory
  - **Stack**: dynamically FILO memory
    - Grows downwards on x86-64
  - **Heap**: catch-all
    - Explicit kernel requests needed to allocate, grow
    - Used by `malloc` (C), `new` (C++)
- Can map additional regions as needed