PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Memory Management I - Introduction

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 3** due **Sunday March 1**
- **Assignment 3** due **Sunday February 16**
  - Equivalent to two questions in the final
- Introduction to Lab 3

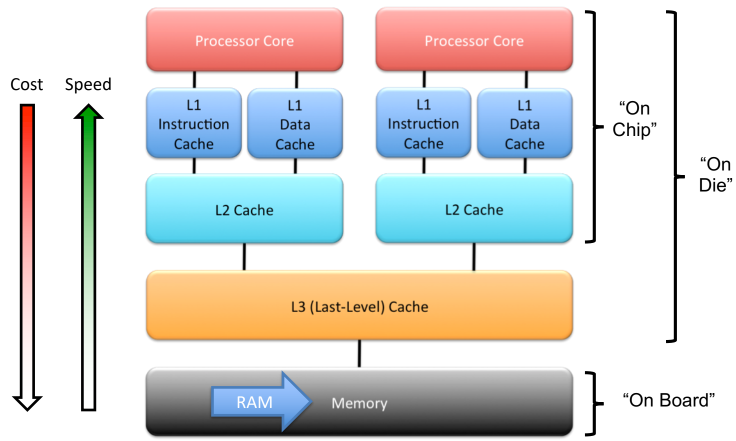
Today's Lecture

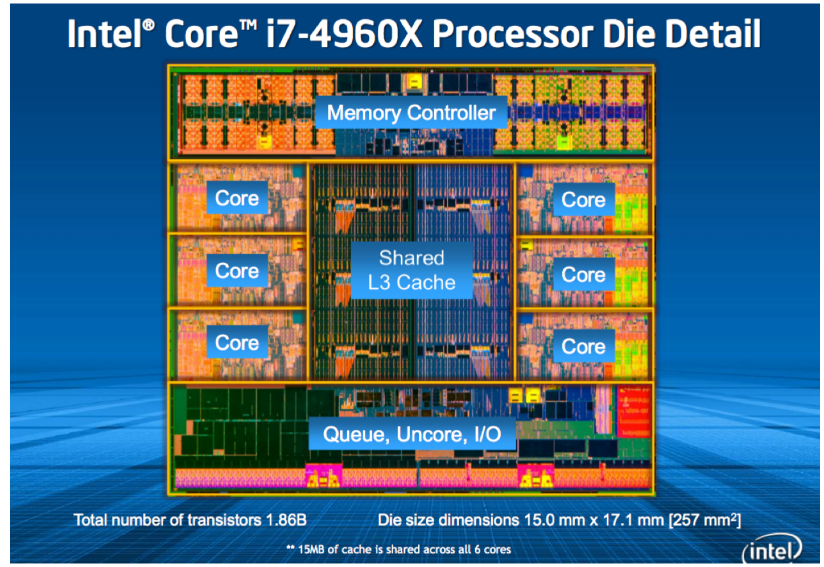
- **Introduction to Memory Management**
  - The User Process View
  - Compilation & Linking
  - Logical & Physical Addresses
  - Partitioning & Protection

Operating Systems & Memory

- Conceptually, computers simply consist of a CPU, some memory, and a number of input/output devices
- **Operating Systems are primarily concerned with CPU scheduling and memory management**
- Memory concepts are interrelated, it's sometimes hard to talk about one concept without considering another ☹
- We've already seen that a CPU has L1, L2, and L3 memory caches on the die/chip
- This series of lectures will largely ignore the L caches and concentrate on the off-chip, but on-board, Random Access Memory (RAM)

Modern CPU Memory Layout





Memory - The User Process View

- User programs go through a number of **initialization steps** before they execute:
  - OS structures initialized
  - Memory must be allocated to the process
  - Executable code must be loaded into that memory
  - Program structures must be created and initialised (stack, variables etc.)
  - Process execution is initiated
  - **YOU DO ALL THIS IN LAB 3 ☺**
- **Fundamental principle:**
  - **Memory is a preemptable resource**
    - No process is allowed to grab memory and refuse to give it up

Memory as a Resource

- Executing programs are processes with machine instructions and variables residing in memory
- Physical memory addresses start from zero but …
  - **Do process addresses need to start from zero?** NO
  - **Do processes need to be stored sequentially in memory?** NO
  - **Does the entire process need to be in memory at one time?** NO
- **Questions:**
  - Can processes in a **time-sharing** operating system ever deadlock over access to I/O devices? YES
  - Can processes in a **multiprogrammed** operating system ever deadlock over memory access? NO
  - Remember:
    - Time-sharing **strictly implies** multiprogrammed
    - Multiprogrammed **does not imply** time-sharing

Memory Binding

Binding of instructions and data to memory addresses can happen at any (or all) of three different stages:

- **Compile time**: If the memory location is known *apriori*, absolute code can be generated; source code must be recompiled if starting location changes
- **Load time**: Relocatable code must be generated if memory location is unknown at compile time
- **Execution time**: Binding delayed until run time, the process (or parts of it) can be moved during its execution from one set of memory addresses to another
  - Need hardware support for address maps
    - e.g., base and limit registers - see later slides

Standard Compilation

- Modern C compilers generate relocatable code:

```
$ gcc -o test test.c
```

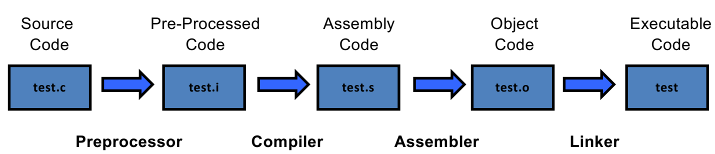
- Utilities exists to convert the relocatable executable to a “flat”, non-relocatable, absolute address format:

```
$ objcopy -O binary test test.bin
```

- **Why might you want to do this?**
  - Performance
  - Especially if cross-compiling for low specification devices
  - Even more especially if planning to run from Read Only Memory (ROM) or Erasable Programmable Read Only Memory (EPROM)

The C "Compilation" Sequence (statically linked)

$ gcc -o test test.c

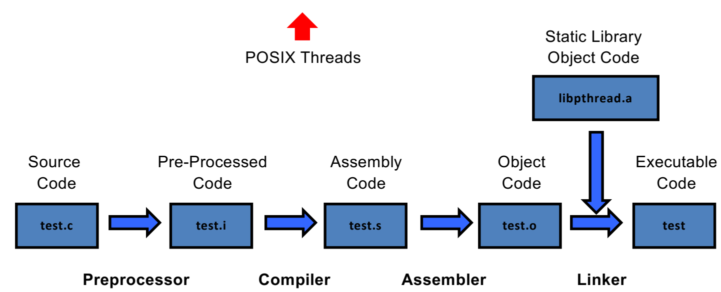


Object and executable files come in several formats:

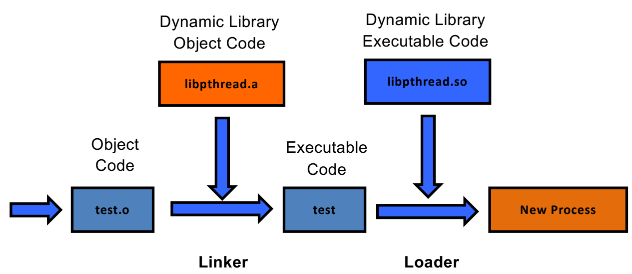
- Common Object File Format (COFF) on Windows
- Executable and Linking Format (ELF) on pretty much everything else, including Unix and Unix-Like Operating Systems (e.g. Linux, macOS, Pintos)

The C “Compilation” Sequence (statically linked)

$ gcc -o test test.c **-lpthread**



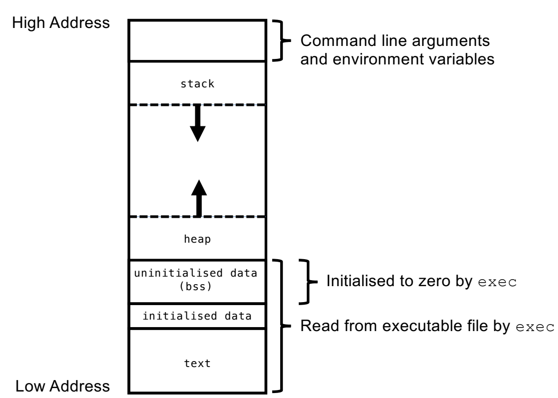
Dynamic Library Binding



Dynamic Linking

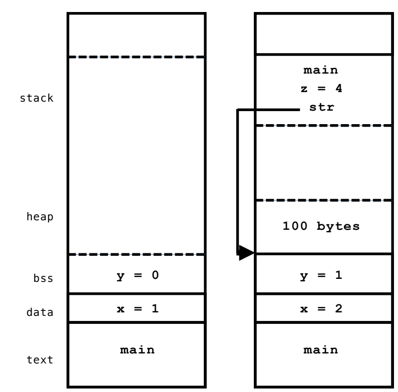
- Linking is postponed until execution time
- A small piece of code, a **"stub"**, is used to locate the appropriate memory-resident library routine
- On the first invocation, the stub is replaces itself with the address of the routine, and calls the routine
  - **Question: Why wait until the first invocation?**
- Operating system help is needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for (large) libraries
- Also allows for library updates (new ones get loaded, no recompilation required)
  - .dll (dynamic link library) on Windows
  - .so (shared object) on Unix and Unix-Like
  - .dynlib (dynamic library) on macOS

C Program Memory Layout



Stack or Heap? (initialised or uninitialised?)

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

int x = 1;
int y;

int main(void)
{
  int z = 4;
  char *str;

  x = 2;

  str = malloc(100*sizeof(char));

  sprintf(str,
    "x = %d, y = %d, z = %d",
    x, y, z);

  printf("%s\n", str);

  free(str);

  return (EXIT_SUCCESS);
}
```



```
$ gcc -o test test.c

$ ls -l test
-rwxrwxr-x 1 david users 7492 Nov  7 12:14 test

$ size --format=Berkely test
   text    data    bss    dec     hex       filename
   1357    292     8      1657    679       test
```

```
$ size --format=SysV test
test  :
section                   size        addr
.interp                     19   134512980
.note.ABI-tag               32   134513000
.note.gnu.build-id          36   134513032
.gnu.hash                   32   134513068
.dynsym                    128   134513100
.dynstr                     94   134513228
.gnu.version                16   134513322
.gnu.version_r              32   134513340
.rel.dyn                     8   134513372
.rel.plt                    40   134513380
.init                       35   134513420
.plt                        96   134513456
.plt.got                     8   134513552
.text                      482   134513568
.fini                       20   134514052
.rodata                     31   134514072
.eh_frame_hdr               44   134514104
.eh_frame                  204   134514148
.init_array                  4   134520584
.fini_array                  4   134520588
.jcr                         4   134520592
.dynamic                   232   134520596
.got                         4   134520828
.got.plt                    32   134520832
.data                       12   134520864
.bss                         8   134520876
.comment                    52           0
Total                     1709
```



```
$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80483a0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6252 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```



```
$ readelf -e test
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000080 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804824c 00024c 00005e 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          080482aa 0002aa 000010 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080482bc 0002bc 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482dc 0002dc 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482e4 0002e4 000028 08  AI  5  24  4
  [11] .init             PROGBITS        0804830c 00030c 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        08048330 000330 000060 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048390 000390 000008 00  AX  0   0  8
  [14] .text             PROGBITS        080483a0 0003a0 0001e2 00  AX  0   0 16
  [15] .fini             PROGBITS        08048584 000584 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048598 000598 00001f 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        080485b8 0005b8 00002c 00   A  0   0  4
  [18] .eh_frame         PROGBITS        080485e4 0005e4 0000cc 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        0804a000 001000 000020 04  WA  0   0  4
  [25] .data             PROGBITS        0804a020 001020 00000c 00  WA  0   0  4
  [26] .bss              NOBITS          0804a02c 00102c 000008 00  WA  0   0  4
  [27] .comment          PROGBITS        00000000 00102c 000034 01  MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 001761 00010a 00      0   0  1
  [29] .symtab           SYMTAB          00000000 001060 0004a0 10     30  47  4
  [30] .strtab           STRTAB          00000000 001500 000261 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```



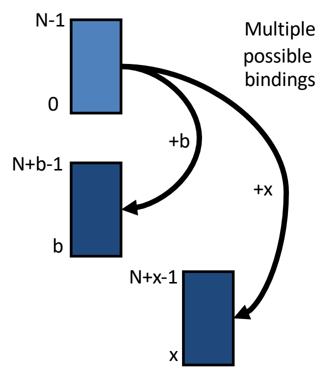
```
$ ls -l test
-rwxrwxr-x 1 david users 7492 Nov  7 12:14 test
$ strip tests
$ ls -l test
-rwxrwxr-x 1 david users 5604 Nov  7 12:16 test
```

```
$ strings test
/lib/ld-linux.so.2
lY5!
3d_b
libc.so.6
_IO_stdin_used
sprintf
puts
malloc
__libc_start_main
free
__gmon_start__
GLIBC_2.0
PTRh
UWVS
t$,U
[^_]
x = %d, y = %d, z = %d
;*2$"(
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0
20160609
```

```
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rel.dyn
.rel.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.jcr
.dynamic
.got.plt
.data
.bss
.comment
```

Logical and Physical Addresses

- **Logical** address space
  - Program / Process view
  - 0 to N-1
- **Physical** address space
  - System view
  - Memory-address register
- "Binding"
  - (one name for) scheme for translating logical to physical addresses

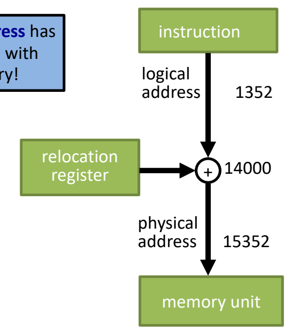


Execution Time Binding

- **Logical** address
  generated by instruction
- **Physical** address
  passed to memory unit
- **Virtual** address
  a logical address in systems where logical and physical memory addresses are different

A **virtual address** has ***nothing*** to do with virtual memory!

A user program always uses **logical** *or* **virtual** addresses, never both - it ***does not*** get to see the physical addresses



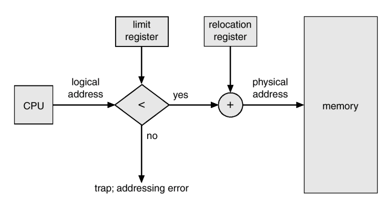
Memory Partitioning

- Main memory must hold the code and data for both the OS and all user processes
- Main memory needs to be divided into partitions:
  - The resident operating system, usually held in low memory
  - User processes held in high memory
- Typically we want several user processes simultaneously in memory
  (multiprogramming / time-sharing)
- Need to partition memory to protect processes from each other

Memory Protection

The relocation-register scheme:

- **Relocation Register**
  - contains the smallest permitted *physical* address
- **Limit Register**
  - contains the maximum *logical* address



Ensures processes only accesses their own address space

Lab 3 - Introduction

CSE130 Winter 2020 : Lab 3

In this lab you will implement user processes and system calls.

As supplied, Pintos is incapable of running user processes and only implements two systems calls. Pintos does, however, have the ability to load ELF binary executable, and has a fully functioning page-based, non-virtual memory management system.

There are three parts to this lab; each depends on the previous one.

- Allow simple user process to run.
- Support argument passing to user processes.
- Implement seven new systems calls.

This lab is worth 15% of your final grade.

Submissions are due NO LATER than 23:59, Sunday March 1, 20202 (three weeks)


```
$ pintos -v -k -T 60 --qemu --filesys-size=2  -p
build/tests/userprog/args-none -a args-none -- -q -f
run args-none
PiLo hda1
Loading............
Kernel command line: -q -f extract run args-none
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  209,510,400 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial
"QEMU HARDDISK"
hda1: 194 sectors (97 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 94 sectors (47 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file
system...
Putting 'args-none' into the file system...
Erasing ustar archive...
Executing 'args-none':
Execution of 'args-none' complete.
Timer: 95 ticks
Thread: 30 idle ticks, 65 kernel ticks, 0 user ticks
hda2 (filesys): 188 reads, 187 writes
hda3 (scratch): 93 reads, 2 writes
Console: 832 characters output
Keyboard: 0 keys pressed
Exception: 53337 page faults
Powering off...
FAIL build/tests/userprog/args-none
Run didn't produce any output
```

```
$ pintos -v -k -T 60 --qemu --filesys-size=2  -p
build/tests/userprog/args-none -a args-none -- -q -f
run args-none
PiLo hda1
Loading............
Kernel command line: -q -f extract run args-none
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  503,808,000 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial
"QEMU HARDDISK"
hda1: 194 sectors (97 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 94 sectors (47 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file
system...
Putting 'args-none' into the file system...
Erasing ustar archive...
Executing 'args-none':
(args) begin
(args) argc = 1
(args) argv[0] = 'args-none'
(args) argv[1] = null
(args) end
args-none: exit(0)
Execution of 'args-none' complete.
Timer: 99 ticks
Thread: 30 idle ticks, 67 kernel ticks, 3 user ticks
hda2 (filesys): 214 reads, 187 writes
hda3 (scratch): 93 reads, 2 writes
Console: 942 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
pass build/tests/userprog/args-none
```


Next Lecture

- Memory Allocation
- A little Lab 3 Secret Sauce