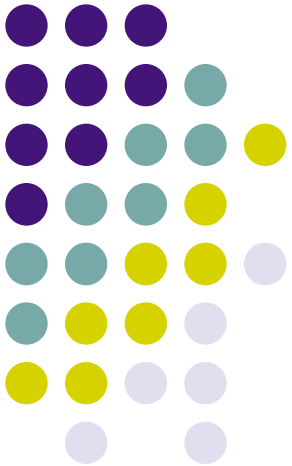
Memory Management Background:
1. Computer System Review
2. Address Binding & Linking



ECE469, Feb 16

Yiying Zhang

19. Everything Becomes Memory

gap among Architecture, Compiler and OS courses

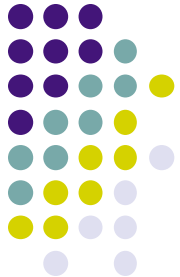
33. cross references — relocation info for linker

Lab3

- Implement mailboxes and mailbox APIs for IPC
- Solve lab2 chemical reactions using mailboxes
- Add CPU running time stats
- Implement BSD-4.4-style priority scheduling
- Add sleep(), yield(), pocessIdle()

2

[lec11] Threads

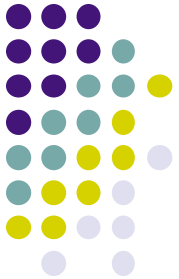


- Separate the concepts of a "thread of control" (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)

- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

3

[lec11] Thread Implementations

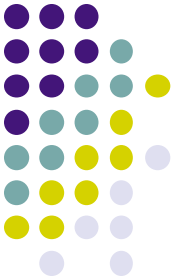


- User-level thread implementation

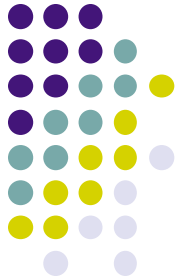
- Kernel-level thread implementation

4

Finally done with processor, Moving to memory!



"Things related to memory" -- you have learned/heard so far



- What is a processor
- What are registers
- What is memory
- How's memory organized

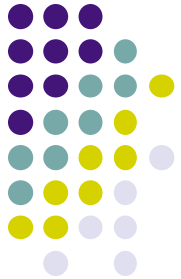
- What's a heap?
- What's a stack?
- Globals, locals, etc.
- PC, SP

- All of above deal with logical memory!

- Hardware memory
- What's a cache and how's it organized

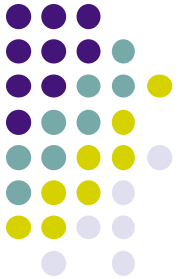
- Physical memory a whole new can of worms!

Warning! You May Be Bored



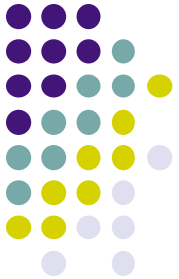
- This material may be redundant if
  - You've already had it (but may have forgotten)
  - You already hacked and found it
  - Your first language was assembly
- Feel free to …
  - I won't be offended
  - You'll still be held responsible for the material

Warning: Approximate Truth



- Some details for general info
- Most details ignored entirely
- Goals
  - Simplicity
  - Coverage
- C, Unix, Uniprocessors, No Threads

What does a Processor Do?



while (1)

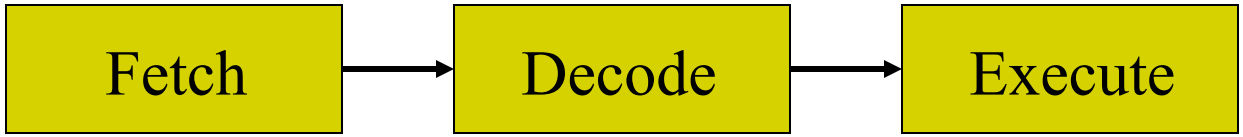
fetch (get instruction)

decode (understand instruction)

execute

Execute: load, store, test, math, branch

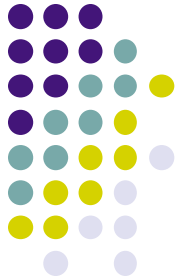
Logical Organization



Logically

F1 D1 E1 F2 E2 D2

Processor Operations



Logically

F1 D1 E1 F2 D2 E2

Pipeline

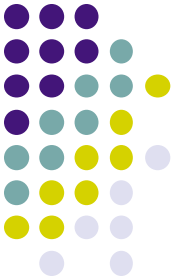
F1 D1 E1

F2 D2 E2

F3 D3 E3

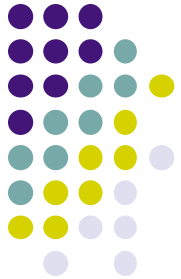
What is the condition for a smooth pipelining?
What can happen to the E part of Load inst: LD R0, _Y

What Is Memory (Address Space)



- “Slots” that hold values
- Slots are “numbered”
- Numbers are called addresses
- Two operations – read or write
  - e.g., LD R1, _X
- What can you put in memory?
  - Anything. No “intrinsic” meaning

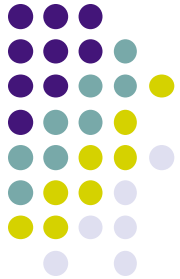
What is Cache?



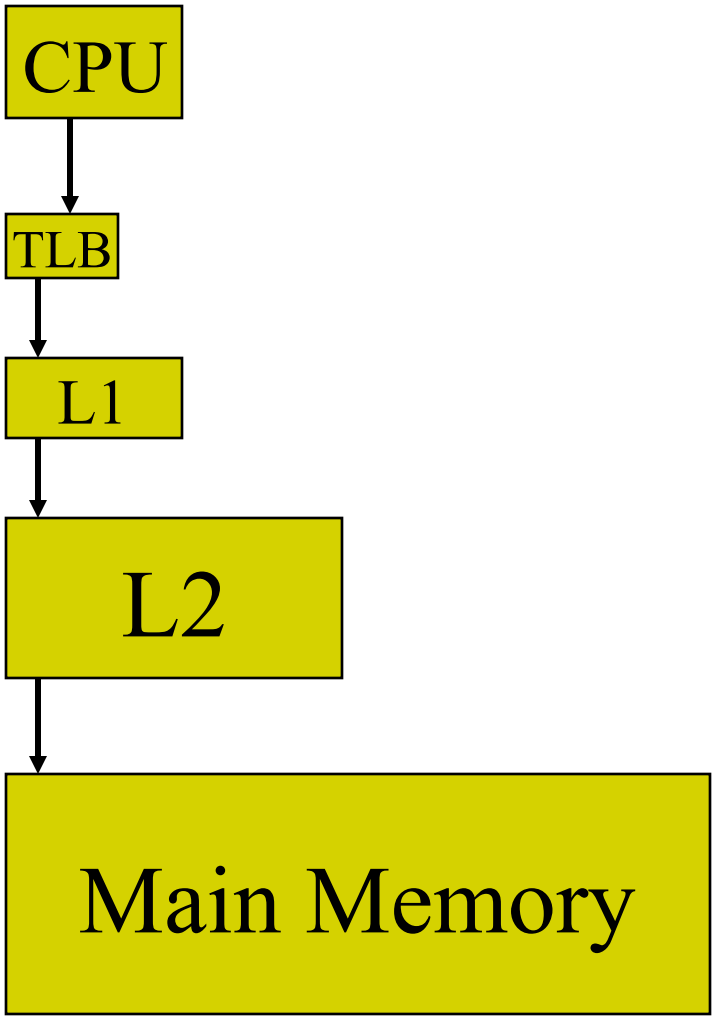
- Another kind of “memory”, physically
- Closer to the processor
- More expensive, smaller, faster

- Operation is logically *transparent*
  - No naming/access by
    - program, compiler, linker, loader
    - OS?
    - CPU?

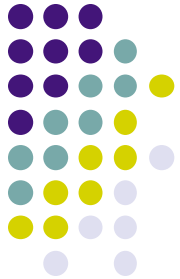
(Hardware) Memory Hierarchy



Where do we hope requests get satisfied?
e.g. LD R1, _X



What Are Registers?



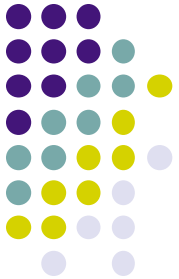
- Places to hold information
- Built into the processor
- “Named” specially

Why?

- Need a place to put operands / temp values
  - e = (a+b) * (c+d) * (a-f)
- Highest level of memory hierarchy
- Register allocation problem – NP-complete
  - who does it?

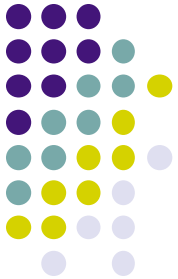
avoid hazard

What Is a Program?



```
int *totalPtr;
Init(void)
{
    totalPtr = calloc(1, sizeof(int));
}
AddToTotal(int y)
{
    int i;
   *totalPtr += y;
}
```

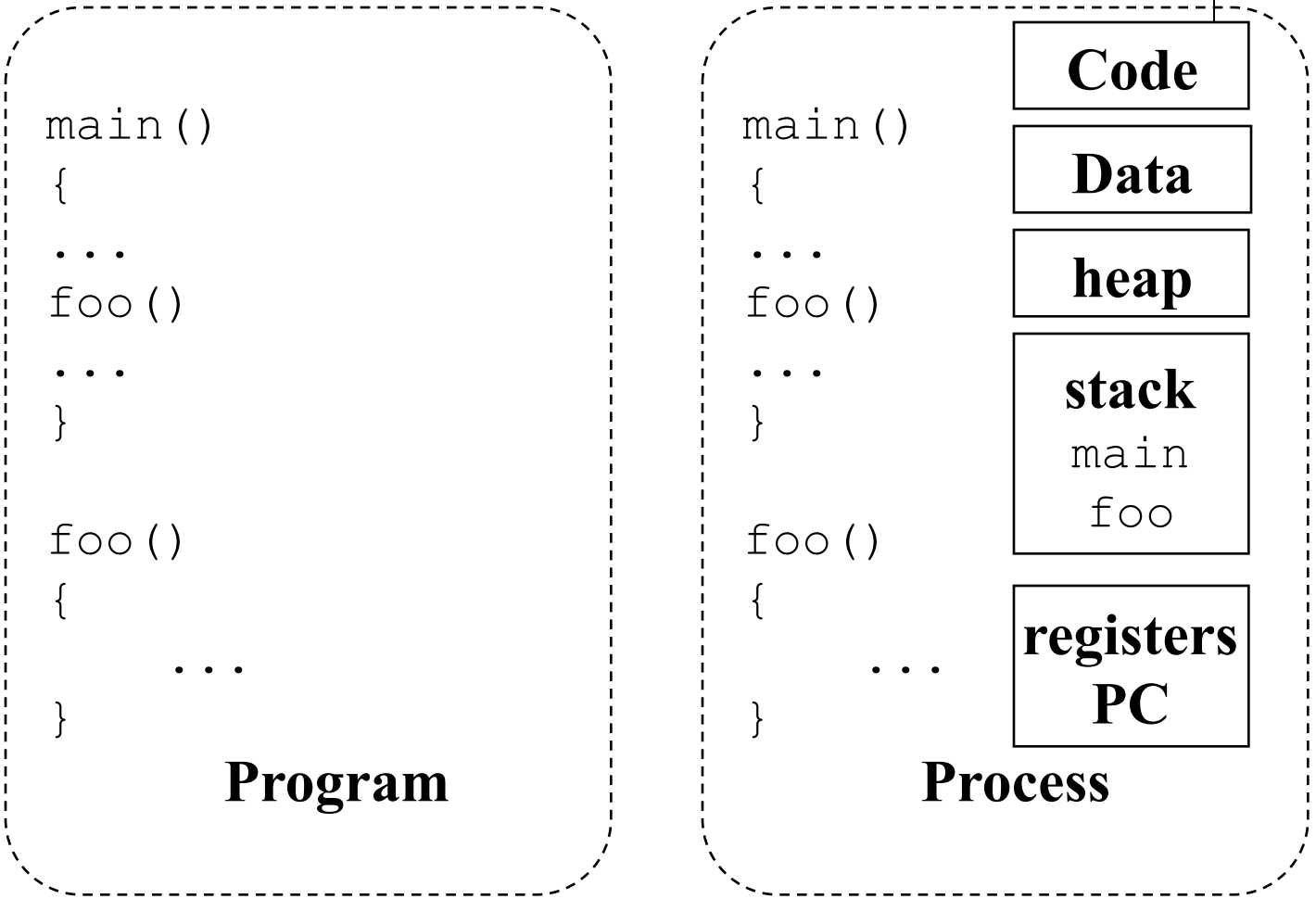
What Is a Program?



- Code
  - Main, subroutines (lib functions)
- Program accessed data
  - Static, global variables
  - Dynamically-allocated data *(e.g. ,malloc())*
  - Parameters, local variables
- What is a process? process: execuation of program

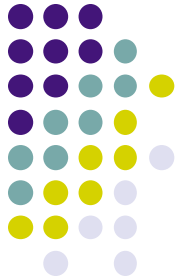
```
int *totalPtr;
Init(void)
{
    totalPtr = calloc(1,
    sizeof(int));
}
AddToTotal(int y)
{
    int i;
   *totalPtr += y;
}
```

[Ice3] Program vs. Process



18

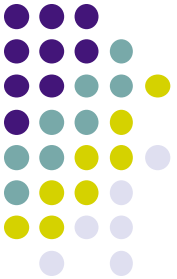
Everything Becomes Memory



- Various ranges of memory (addr space) are ***used*** for different purposes
  - Text/Code (program instructions)
  - Data (global variables)
  - Stack (local variables, parameters, etc)
  - Heap (dynamically allocated memory)

each process has its own virtual memory in OS -> flexible

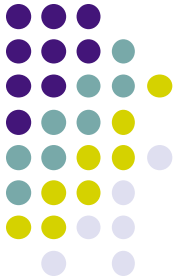
What Is a Stack?



- Data structure that supports push/pop
- Uses?
  - Anything w/ LIFO (last-in first-out behavior)
  - Only care about recent behavior

- Example?
  - DFS
  - Procedure calls! function call

Procedure calls

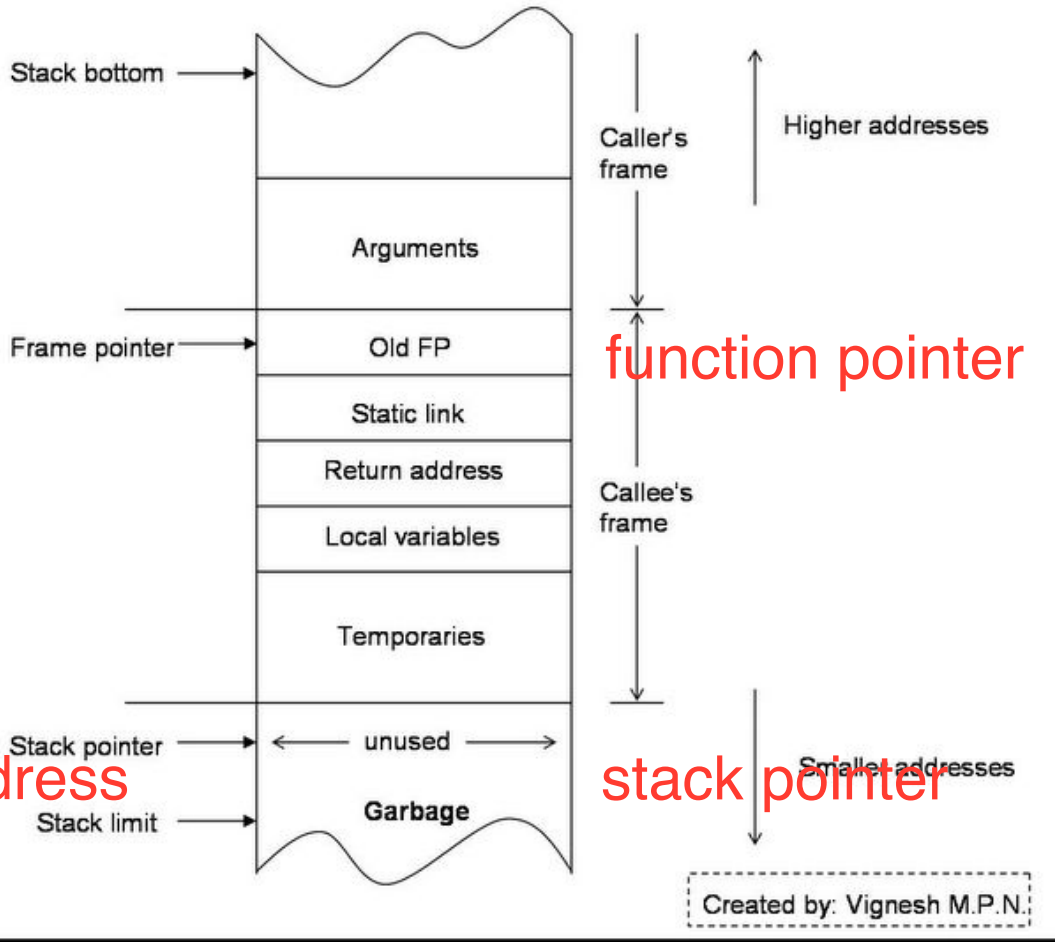


- Incoming parameters from caller
  - Don't even know who caller is
- Local variables survive only when in use
- Temporary variables (a+b) * (c+d)

```
void Loop(int N)
{
    int a,b,c,d,e,f,g;
    …
    g = (a+b)*(c+d);
}
```

Stack Frames

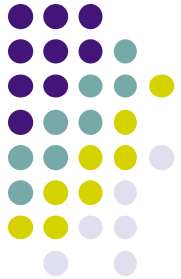
- Frame = info for one procedure call
- Incoming parameters
- Return address for caller
- New local variables
- New temporary variables
- Size of frame



local variables address

stack pointer

Stack Is Just Memory

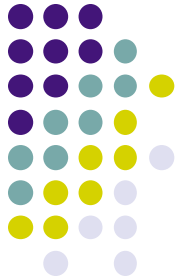


- Defined, used by convention
  - Agreement among OS, compiler, programmer
- How does OS manage stack (and heap)?
  - Allocate chunk of memory
  - Have pointer into chunk
- Problems?
  - Must know maximum size of stack?

does not know max size of the stack

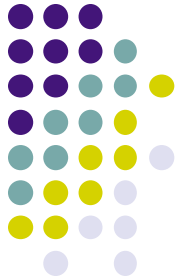
- How to stay efficient despite uncertainty?

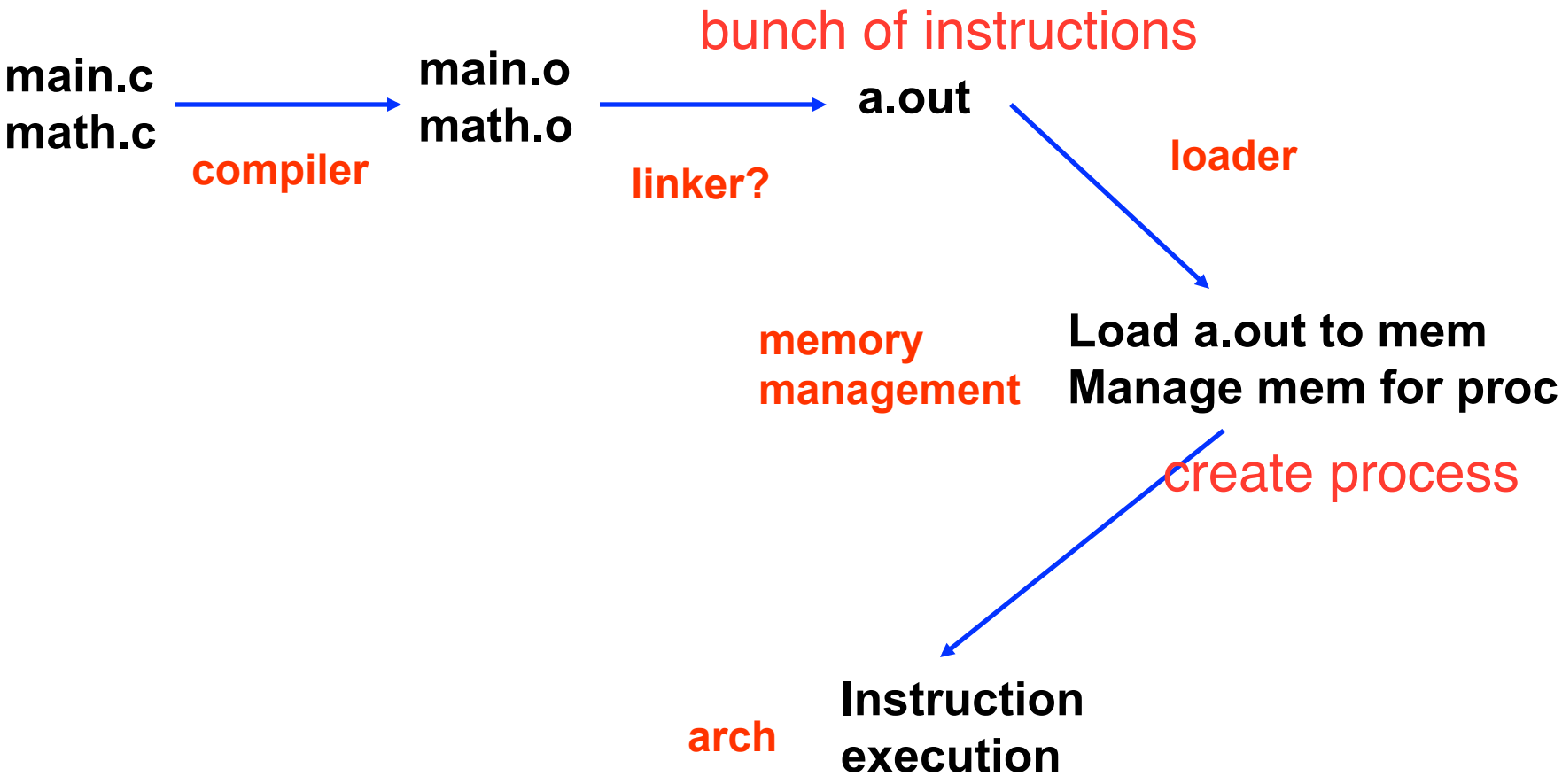
What Does Memory Look Like?



- Logical memory
  - Code+data, stack, heap
  - Which ones grow?
  - How do you give them the most flexibility
- Physical memory?
  - Another can of worms, entirely
- We will move on to Memory Management

A gap among Architecture, Compiler and OS courses





Example

Main.c:

```
main( )
{
  static float x, val;

  extern float sin( );
  extern printf( ), scanf( )

  printf("Type number: ");
  scanf("%f", &x);
  val = sin(x);
  printf("Sine is %f", val);
}
```

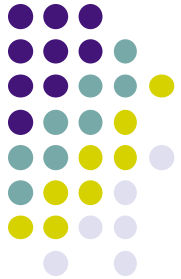
Math.c:

```
float sin(float x)
{
  static float temp1, temp2,
    result;

  – Calculate Sine –

  return result;
}
```

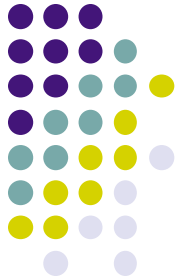
Example (cont)



- Main.c uses externally defined sin() and C library function calls
  - printf()
  - scanf()

- How does this program get compiled and linked?

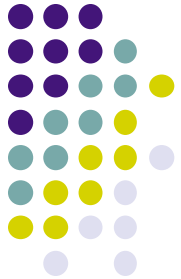
Tasks of a Linker



- Read in object files produced by the compiler
- Produce a self-sufficient object file (a.out)
  - Involves
    - segment relocation 找到各个调用函数的位置
    - address translation 找到之后汇集到一起 汇集到a.out

      combine the .o from library C and .o from main into a.out

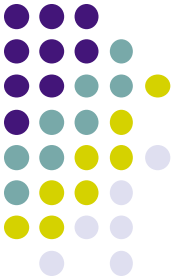
Back to the Example



- Do the main/sin example with the following segment sizes
  - Main: code 420, data 42
  - Math: code 1600, data 12
  - Library: code 1230, data 148

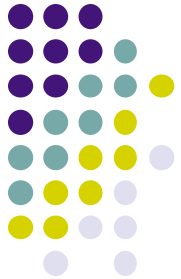
- Output: code 3250, data 202
- In reality segment starts on a page (4 Kbytes) boundary

Memory Layout – Division of Responsibility



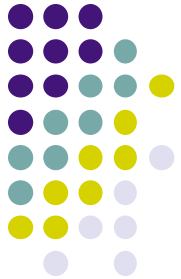
- Compiler: generates object file
  - Information is incomplete
  - Each file may refer to symbols defined in other files
- Linker: puts everything together
  - Creates one object file that is complete
  - No references outside this file (usually)

Division of Responsibility (cont)



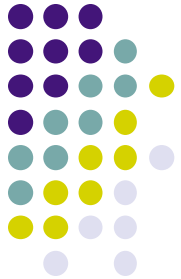
- OS
  - Allow several different processes to share physical memory
  - Provide ways of dynamically allocating more physical memory

What could the compiler not do?



- Compiler does not know final memory layout
  - It assumes everything in .o starts at address zero
  - For each .o file, compiler puts information in the symbol table to tell the linker how to rearrange outside references safely/efficiently
    - For exported functions, absolute jumps, etc
  - Linker needs to rearrange segments
  - What makes rearrangement tricky?
    - Addresses!

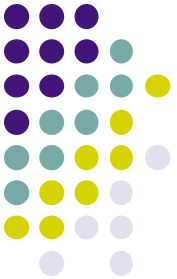
What couldn't the compiler do? (cont)



- Compiler does not know all the references
  - e.g. addresses of functions / variables defined in other files
  - Where it does not know, it just puts a zero, and leaves a comment (relocation info) for the linker to fix things up

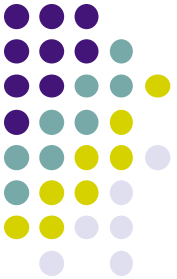
- These are called *cross references*

Components of Object File



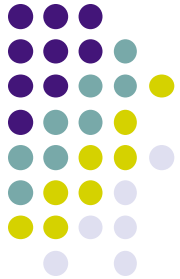
- Header
- Two segments
  - Code segment and data segment
  - OS adds empty heap/stack segment while loading
- Size and address of each segment
  - Address of a segment is the address where the segment begins

Components of Object File (cont)



- Symbol table
  - Information about stuff defined in this module
  - Used for getting from the name of a thing (subroutine/variable) to the thing itself
- Relocation information
  - Information about addresses in this module linker should fix
    - External references (e.g. lib call)
    - Internal references (e.g. absolute jumps)
- Additional information for debugger

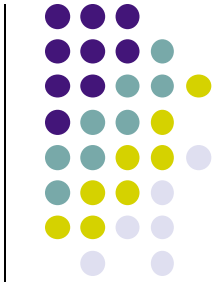
Linker functionality



- Three functions of a linker
  - Collect all the pieces of a program
  - Figure out new memory organization
    - Combine like segments
    - Does the ordering matter? (spatial locality for cache)
  - Touch-up addresses

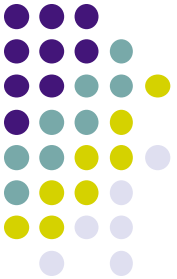
- The result is a runnable object file (e.g. a.out)

Linker – a closer look



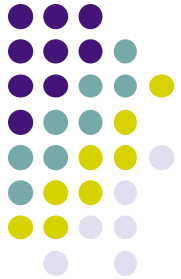
- Linker can shuffle segments around at will, but cannot rearrange information within a segment

Linker requires at least two passes



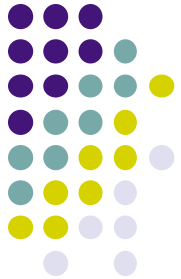
- Pass 1: decide how to arrange memory
- Pass 2: address touch-up

Pass 1 – Segment Relocation



- Pass 1 assigns input segment locations to fill-up output segments
  - Read and adjust symbol table information
  - Read relocation info to see what additional stuff from libraries is required

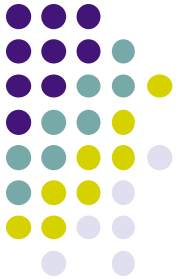
Pass 2 – Address translation



- In pass 2, linker reads segment and relocation information from files, fixes up addresses, and writes a new object file

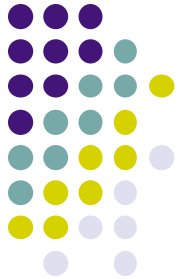
- Relocation information is crucial for this part

Putting It Together



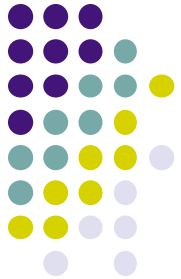
- Pass 1:
  - Read symbol table, relocation table
  - Rearrange segments, adjust symbol table
- Pass 2:
  - Read segments and relocation information
  - Touch-up addresses
  - Write new object file

Dynamic linking



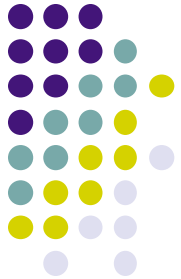
- Static linking – each lib copied into each binary
- Dynamic linking:
  - Instead of system call wrapper code, a stub that finds lib code in memory, or loads it if it is not present
- Pros:
  - all procs can share copy (shared libraries)
    - Standard C library
  - live updates

Dynamic loading



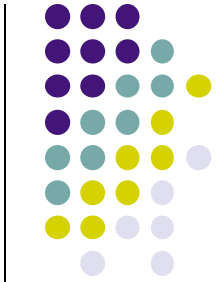
- Program can call dynamic linker via
  - dlopen()
  - library is loaded at running time
- Pros:
  - More flexibility -- A running program can
    - create a new program
    - invoke the compiler
    - invoke the linker
    - load it!

Memory Usage Classification



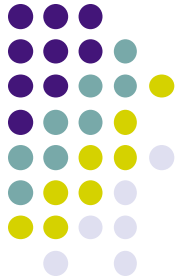
- Memory used to store information that can be used in various ways
- Some possible classifications
  - Role in programming language
  - Changeability
  - Address vs. data
  - Binding time

Role in Programming Language



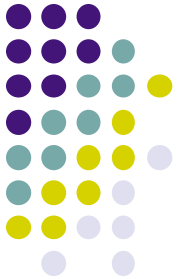
- Instructions
  - Specify the operations to be performed on the operands
- Variables
  - Store the information that changes as program runs
- Constants
  - Used as operands but never change

Changeability



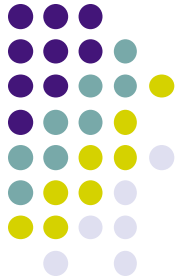
- Read-only
  - Example: code, constants
- Read and write
  - Example: Variables

Address vs. Data



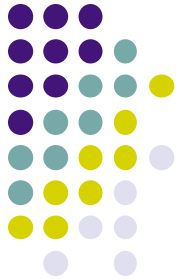
- Need to distinguish between addresses and data
- Why?
  - Addresses need to be modified if the memory is re-arranged

Binding Time



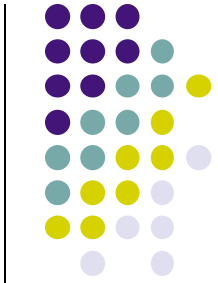
- When is the space allocated?
  - Compile-time, link-time, or load-time
  - Static: arrangement determined once and for all
  - Dynamic: arrangement cannot be determined until runtime, and may change
    - malloc( ), free( )

Classification – summary



- Classifications overlap
  - Variables may be static or dynamic
  - Code may be read-only or read and write
    - Read-only: Solaris
    - Read and write: DOS
- So what is this all about?
- What does memory look like when a process is running?

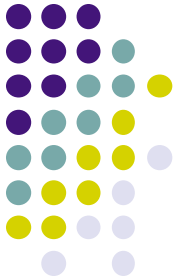
Memory Layout



- Memory divided into segments
  - Code (called text in Unix terminology)
  - Data also have cache for code only and cache for data only
  - Stack

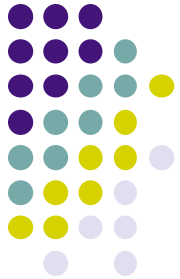
- Why different segments?
  - To enforce classification
  - e.g. code and data treated differently at hardware level

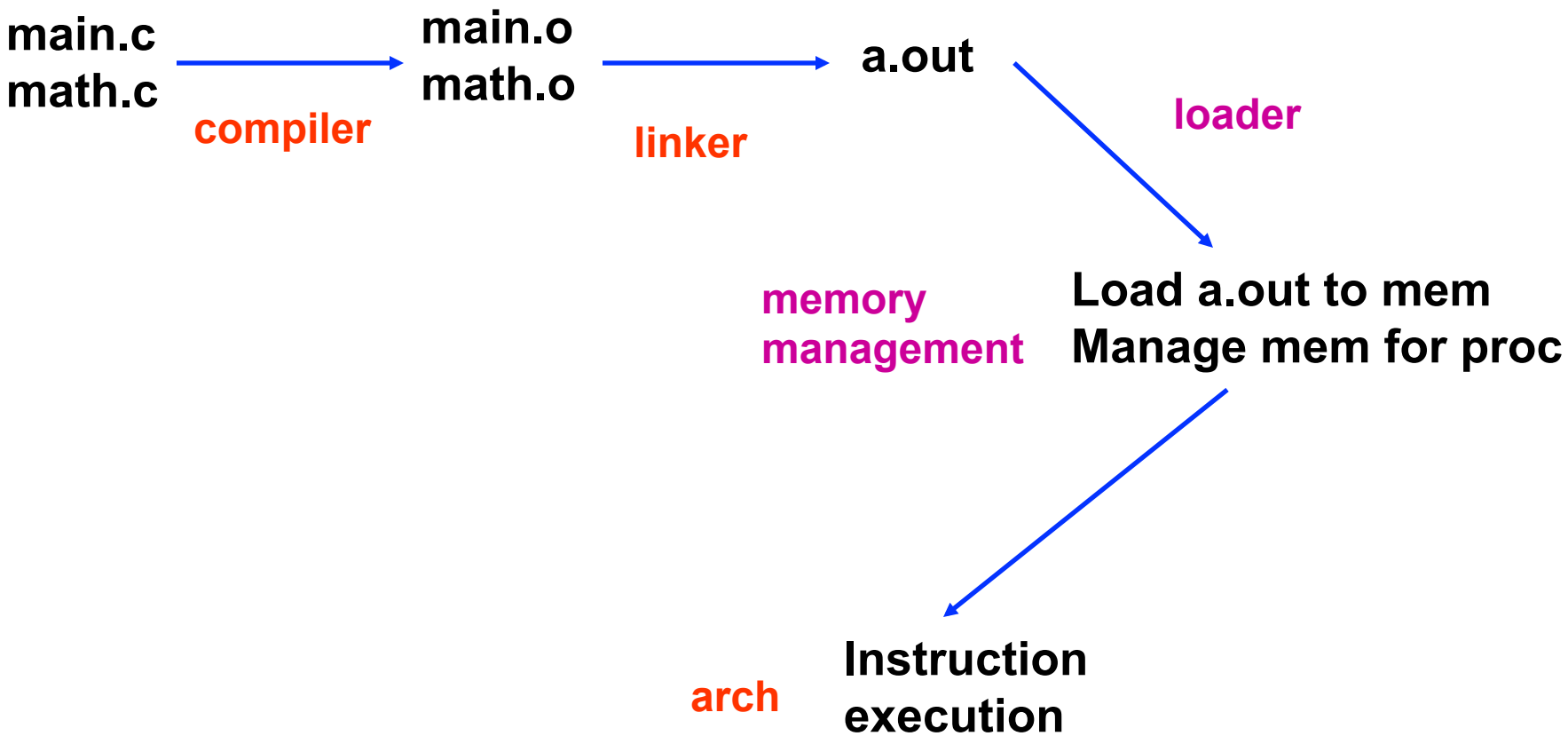
What Is "Systems"



- Everything that is "not something else"
  - Well-defined non-systems areas
    - Theory (and algorithms, formal security)
    - Languages (functional side)
    - Graphics
- So what's left?
  - Architecture, OS, Compilers, Networking, etc.
  - Applications

Connecting the dots

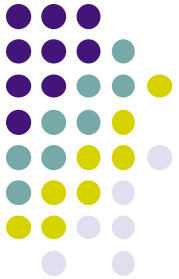




The big picture

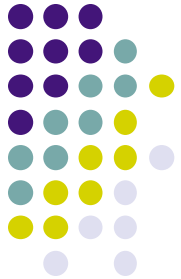
- a.out needs address space for
  - text seg, data seg, and (hypothetical) heap, stack
- A running process needs phy. memory for
  - text seg, data seg, heap, stack
- But no way of knowing where in phy mem at
  - Programming time, compile time, linking time
- Best way out?
  - Make agreement to divide responsibility
    - Assume address starts at 0 at prog/compile/link time
    - OS needs to work hard at loading/runing time
      mapping the virtual to physical

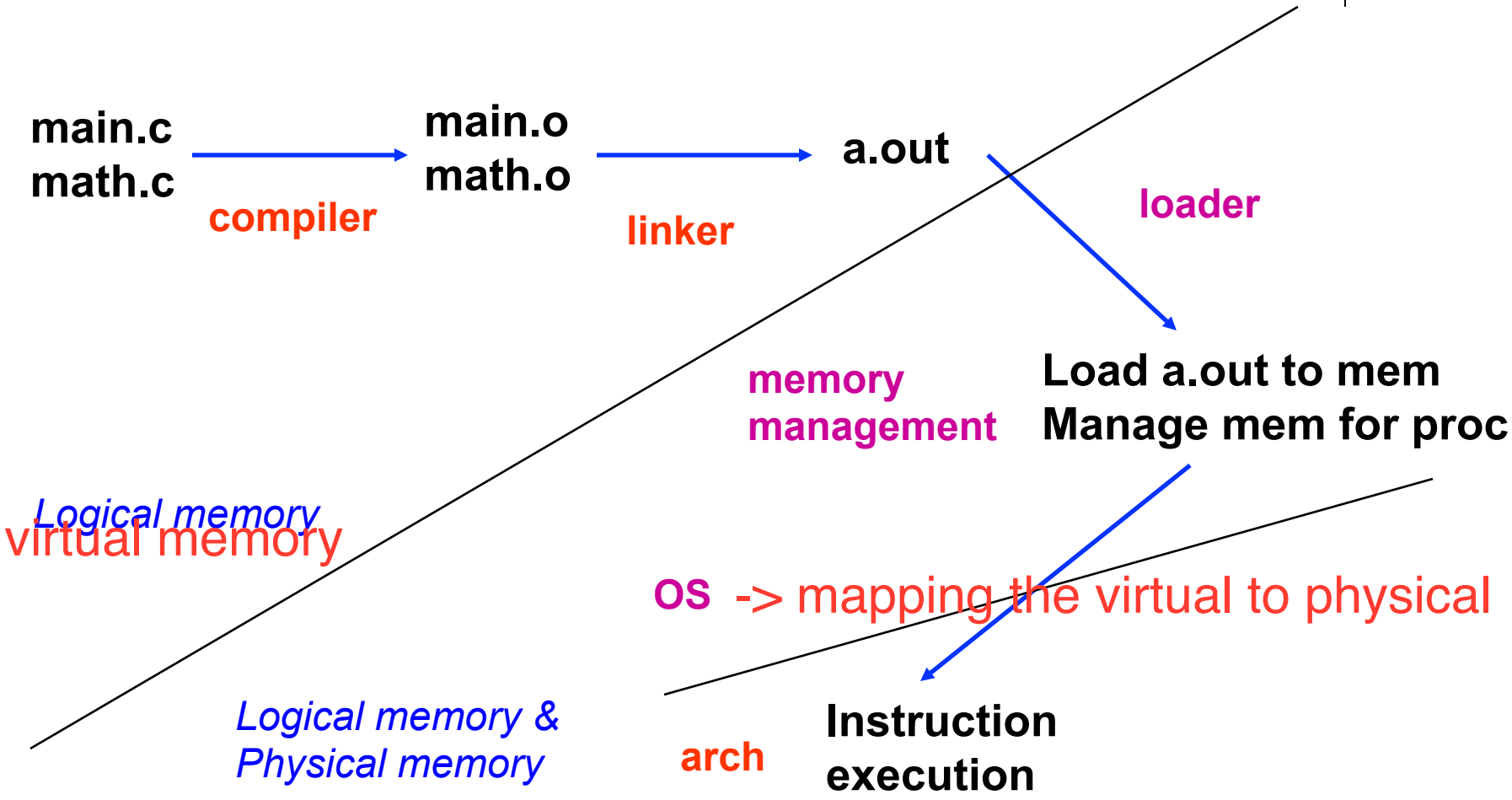
Big picture (cont)



- OS deals with physical memory
  - Loading
  - Sharing physical memory between processes
  - Dynamic memory allocation

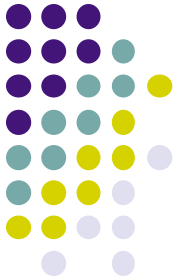
Connecting the dots





55

Questions?



- Will start memory management next time