Modern Operating System Concepts

Benjamin Brewster

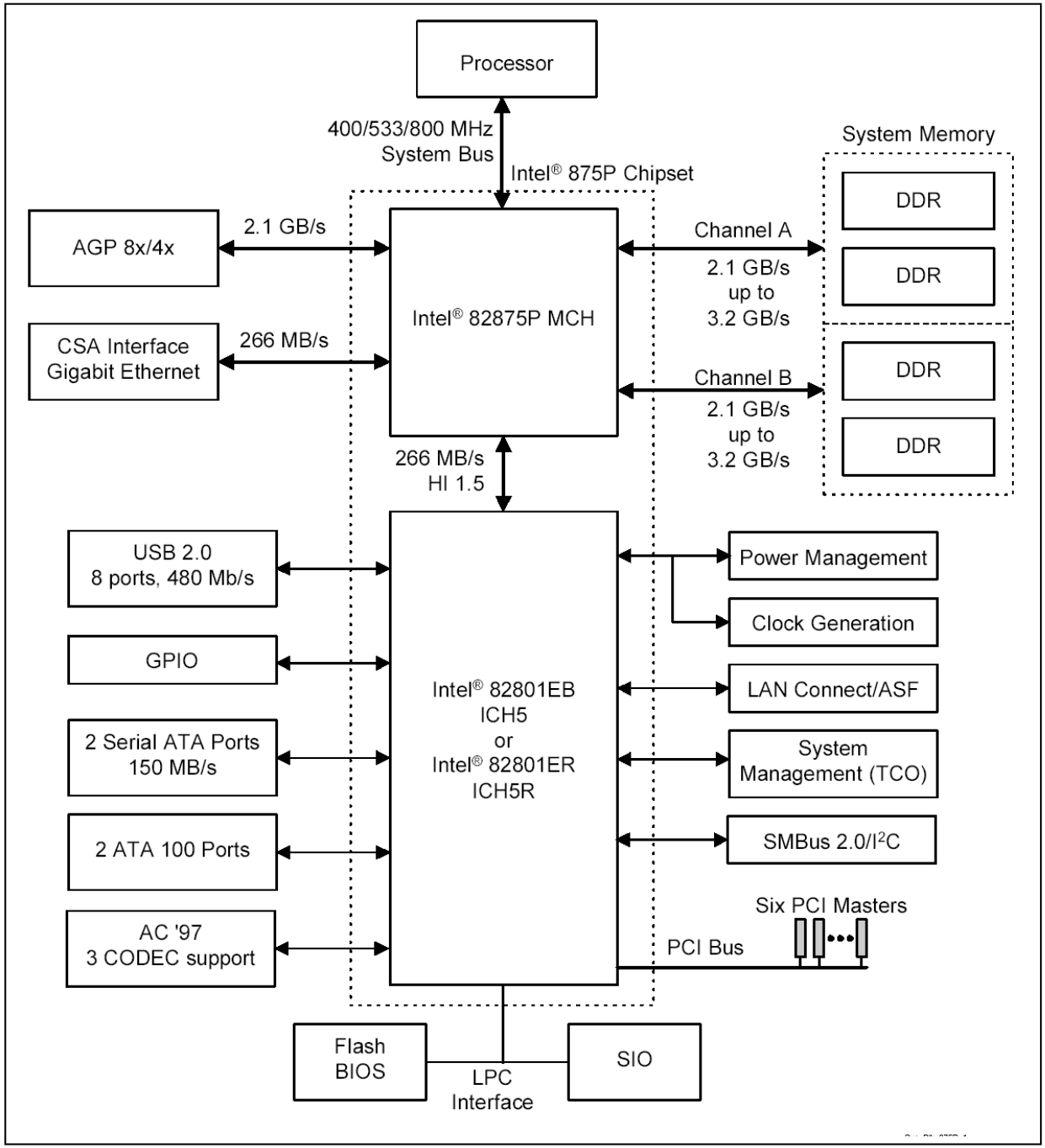
Adapted from slides by Jon Herlocker, OSU

The Underlying Hardware

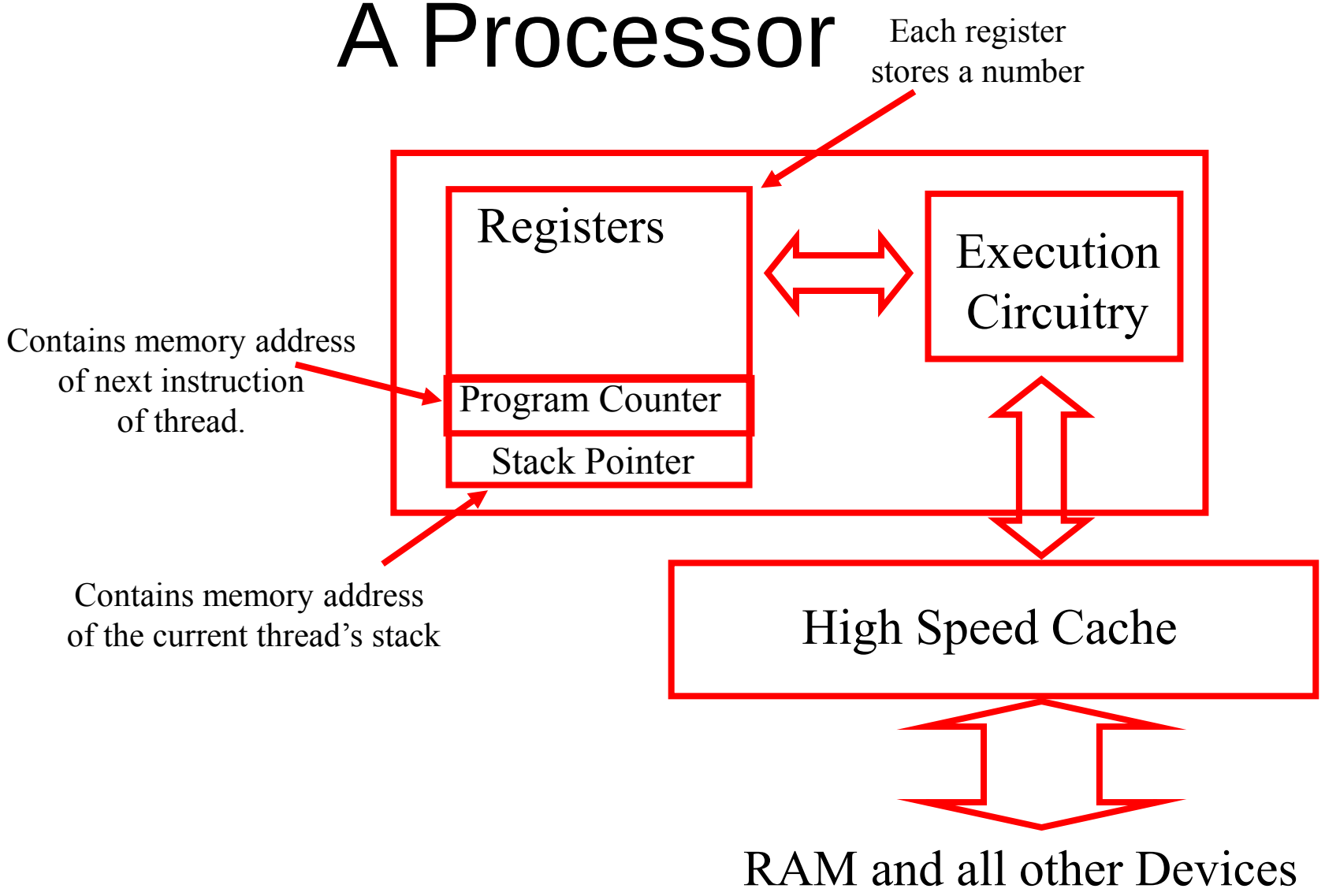
- The OS provides applications access to the hardware in an *abstracted* manner

- What does that hardware actually look like?

Figure 1. Intel® 875P Chipset System Block Diagram

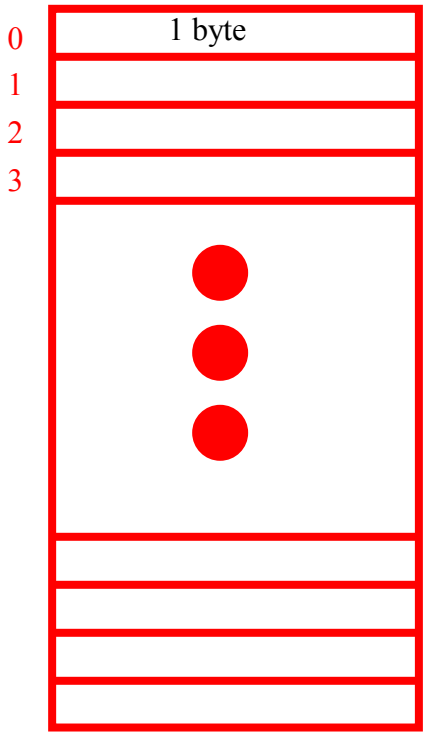


A Processor


Each register stores a number
Registers
Execution Circuitry
Contains memory address of next instruction of thread.
Program Counter
Stack Pointer
Contains memory address of the current thread's stack
High Speed Cache
RAM and all other Devices

Memory

Memory is an array of *bytes*



Each byte has an address (i.e. the index into the array)

Memory is temporary storage, Much slower than the processor and cache but much faster than a disk

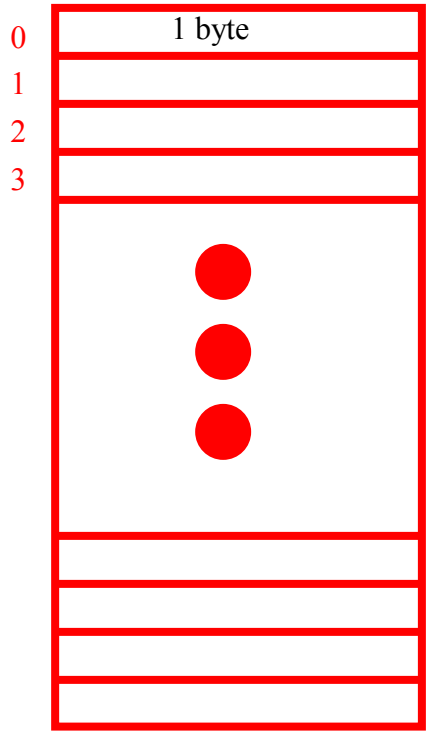
Other storage devices

- Persistent storage
  - Magnetic disk
  - Non-volatile memory (flash, etc.)
  - Magnetic tapes
  - Optical (CD-ROM, DVDs, BD, HD-DVD, etc.)
  - Slow, but persist without power

Virtual Memory

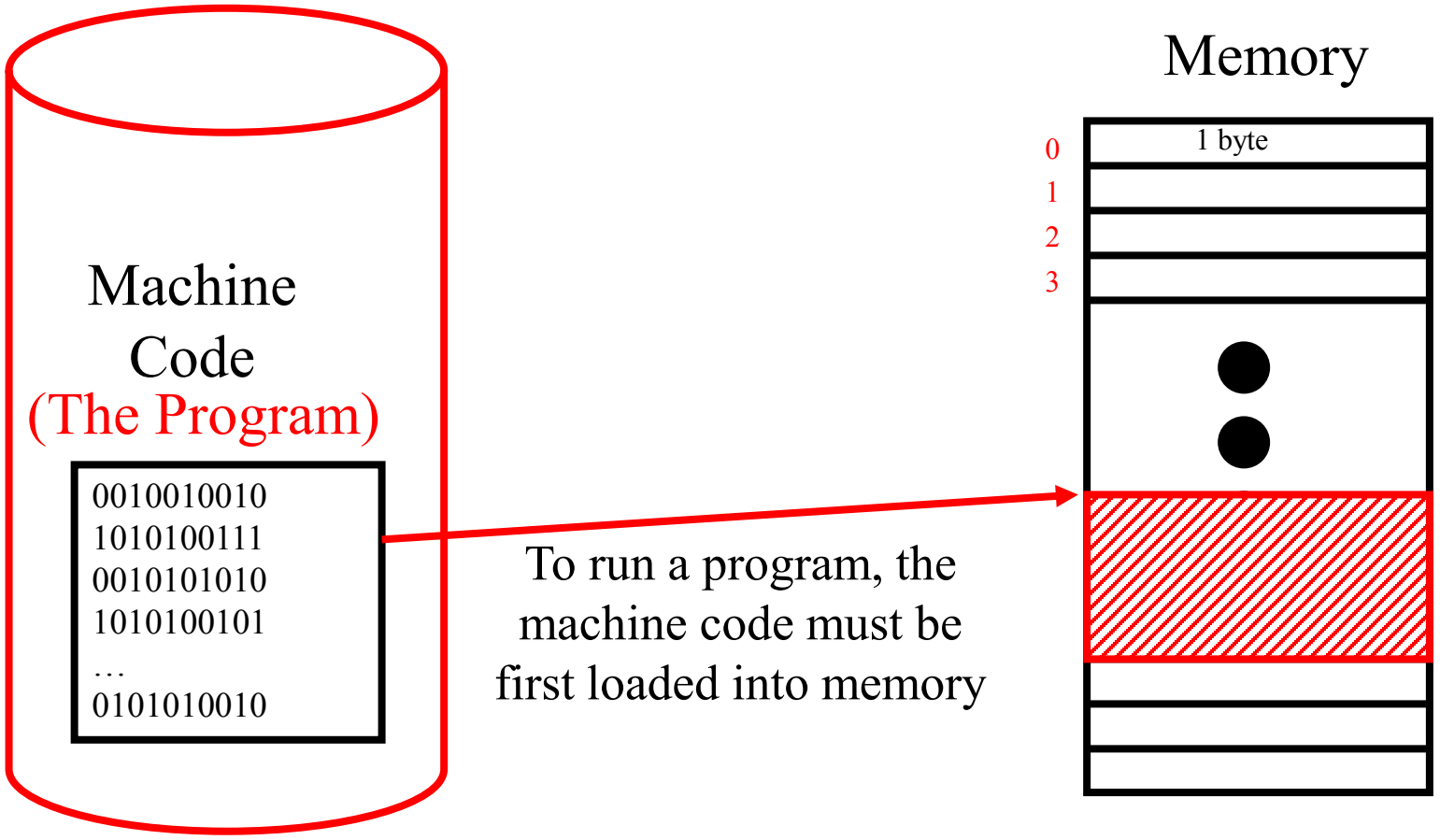
- Virtual memory hardware creates the illusion of
  - Exclusive memory
    - Not shared
  - Unlimited memory
    - (up to the maximum address size)

Thus a program's ***address space*** begins at 0

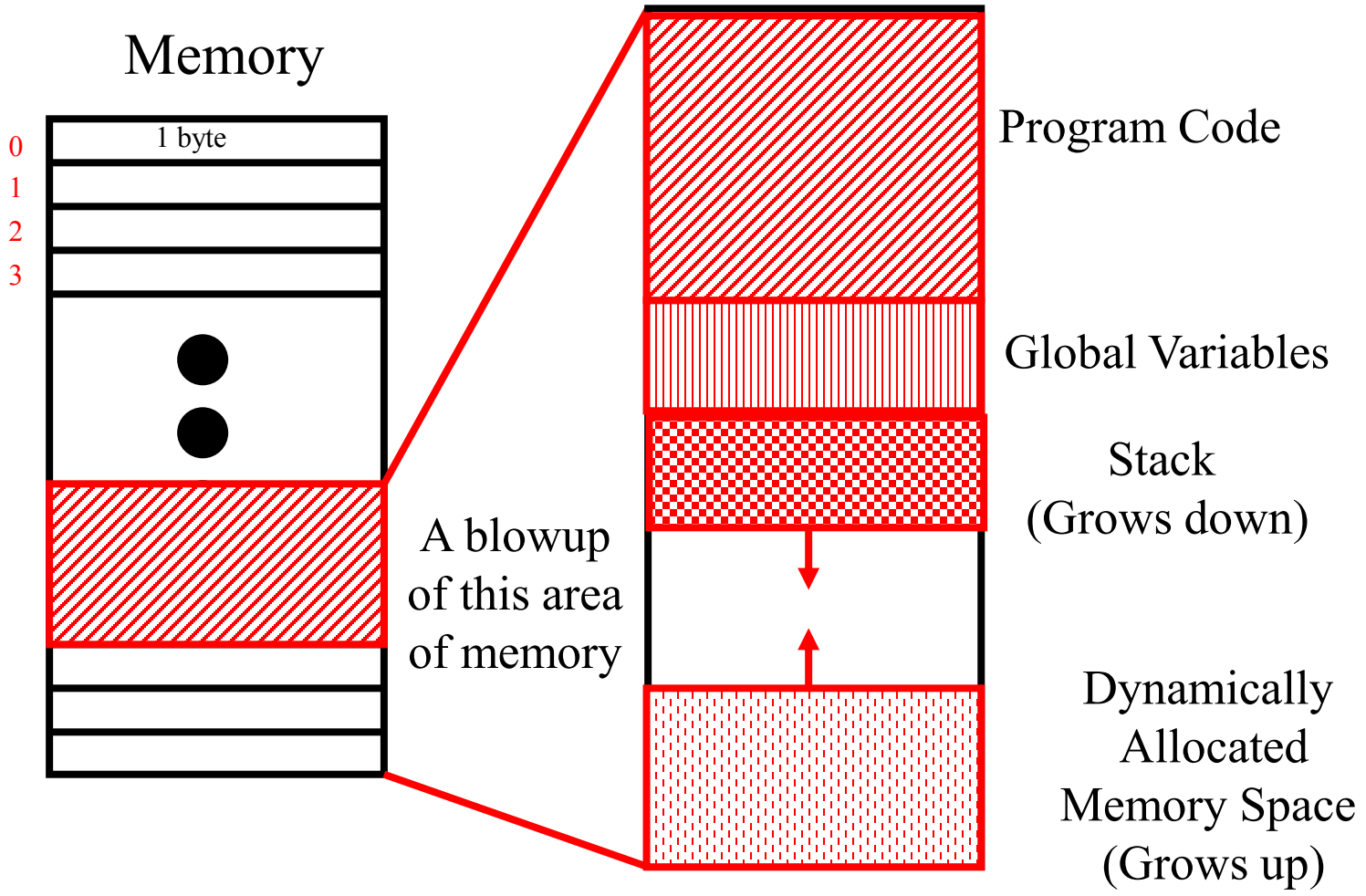


And ends at the largest possible address processor can handle

Running a Program



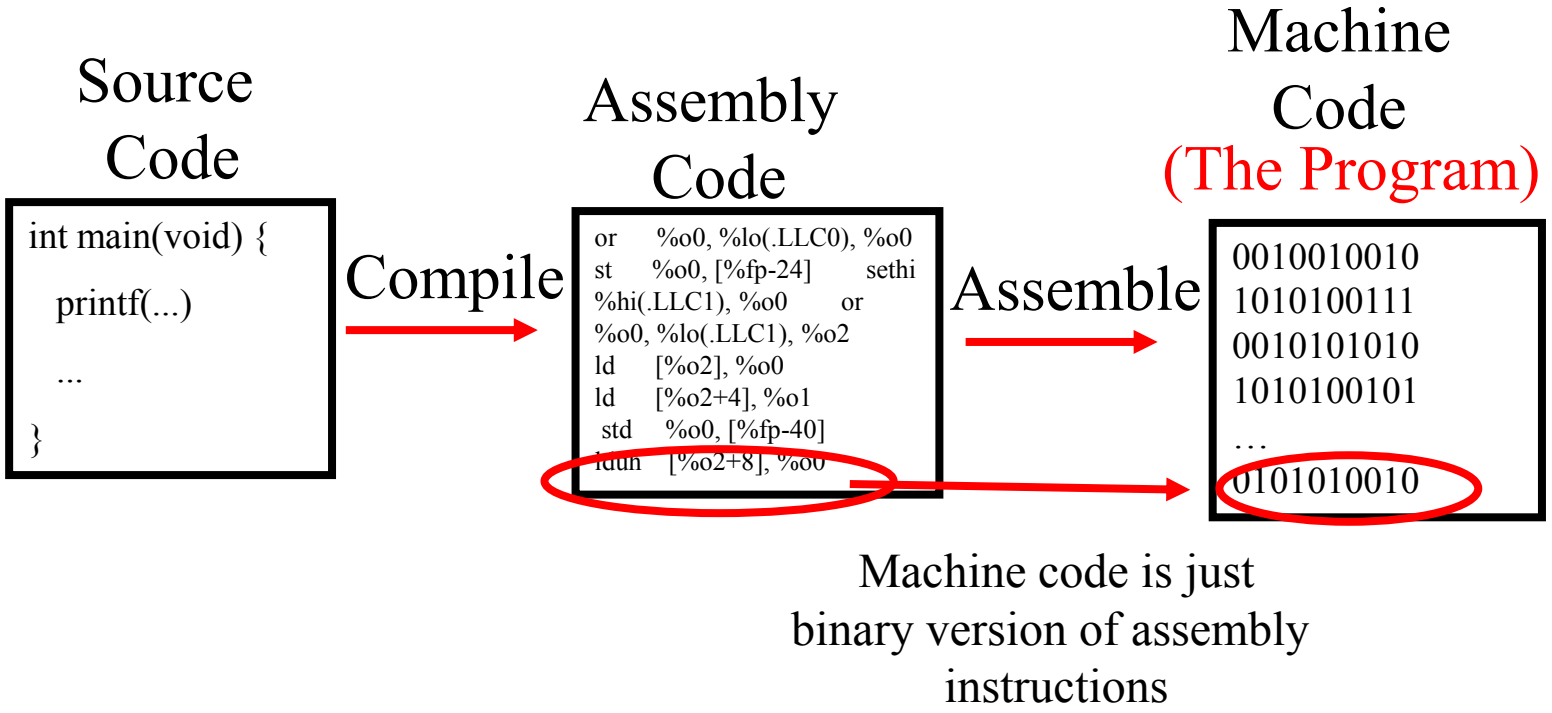
Organization of Program in Memory



Creating The Program Code

- How do we turn a high-level program (C++, Java) into something that the computer can run?

Creating The Program Code – High Level



- By default, most compilers will both compile and assemble your source code.
- `gcc -S yourfile.c` will put the assembly code your program becomes into yourfile.s

The compile/link process

- The C pre-processor expands macros
  - #include, #define, #ifdef, etc.
- The compiler parses your source, checks for errors and generates assembly code
- The compiler automatically calls the assembler, which converts assembly code to machine code
- If you are compiling an executable, the linker step tries to match function calls to function code (they might be in different files!)

GCC

- The options you need to understand
  - `-g` Compile with debugging info for GDB
  - `-c` Compiles only, without linking (more later)
  - `-O3` Optimizes as much as possible
  - `-o` specifies the *name* of the output file
  - These should work with any Unix C/C++ compiler

- `-Wall`: a useful option for gcc
  - Turns on all warnings - which may point out potentially flawed code

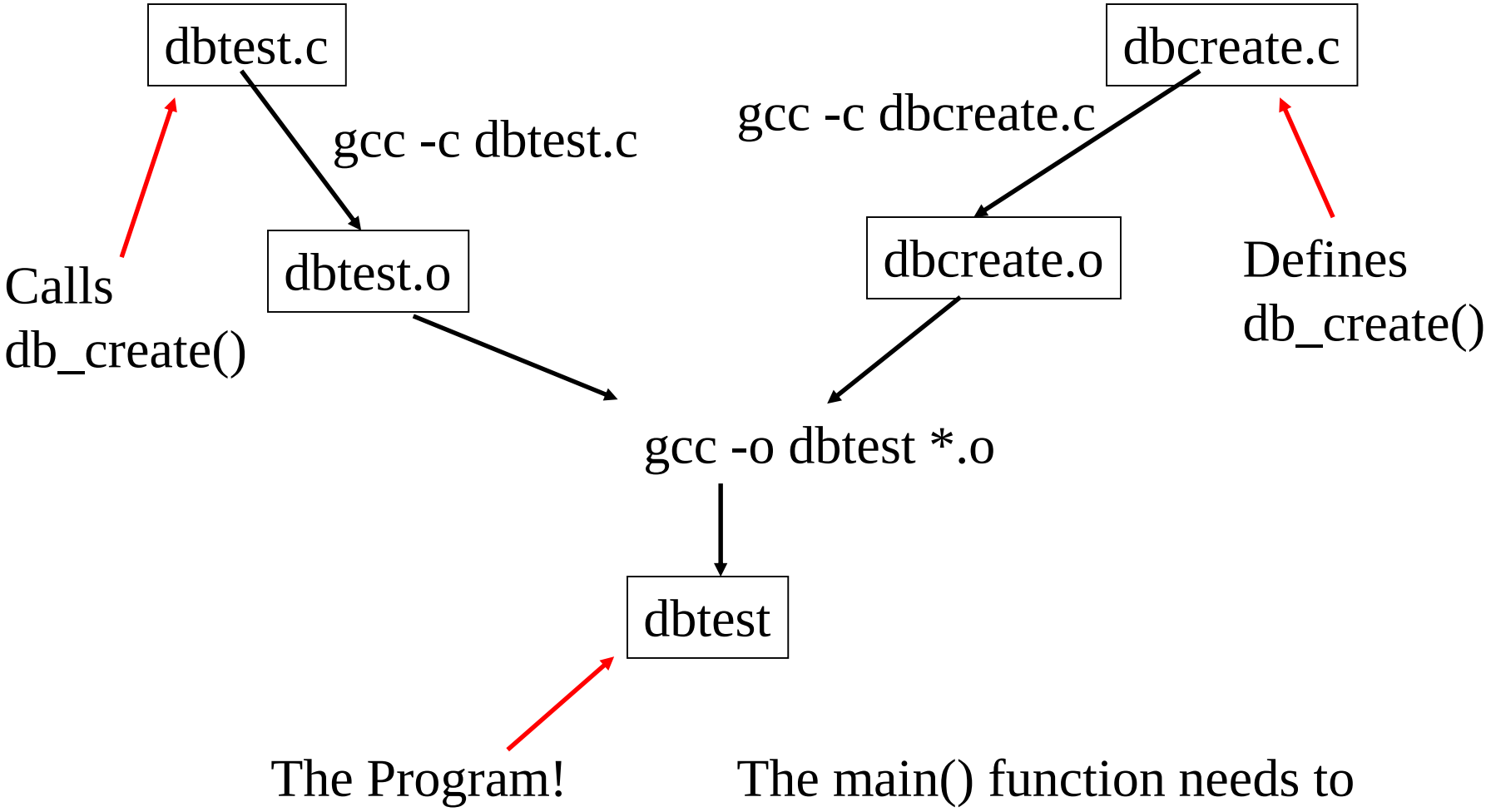
Compiling an executable

- If you have one source (.c) file
  - `gcc -o dbtest dbtest.c`
    - Don't call your executable program `test`... why?
- If you have multiple source (.c) files
  - Option A: compile them all at once
    - `gcc -o dbtest dbtest.c dbcreate.c dbopen.c`
  - Option B: (more efficient) compile them one at a time without linking, then link them all together at the end

Separate Compile and Link

- First compile all source files separately into object files (.o)
  - gcc -c dbtest.c
  - gcc -c dbcreate.c
  - gcc -c dbopen.c
  - gcc -c dbread.c
  - …
- Once all the object (.o) files have been created, link them together to create an executable
  - gcc -o dbtest dbtest.o dbcreate.o dbopen.o

Compile/Link



Library Archives

- Library archives are collections of object files (.o) gathered into a single large file, with indexes to make accessing them fast
  - usually faster than having to read every .o file
  - easier to link with if you aren't changing the library object files frequently

- To create a library
  - First create all the object files (see previous slide)
  - Then use the `ar` command:
    - `ar -r libdb.a dbcreate.o dbopen.o dbread.o`

Using Library Archives

- Include the library anywhere you can use an object file:
  - `gcc -o dbtest dbtest.o libdb.a`

Concurrency

- Concurrency - multiple processes executing at the same time
- On UNIX concurrency is easy
  - Multiple processes can be running simultaneously
  - Multiple copies of the same program can be running
- Concurrency is very powerful
  - Greatly increases the efficiency of an OS:
    - While one process is waiting for I/O, another process can use the CPU

Multi-user, multi-process system?

- Multiprogramming
  - More than one process can be ready to execute…
  - System calls trigger “context switches”: let the next process run
  - The process will not execute again until its system call returns
- Timesharing
  - CPU time split between multiple processes
    - Gives illusion that many processes are running at once

Can processes communicate? Yes! Stay tuned…

Multi-user, multi-process system?

- What about multi-process**or** systems?
  - Each process can do multiprogramming AND timesharing

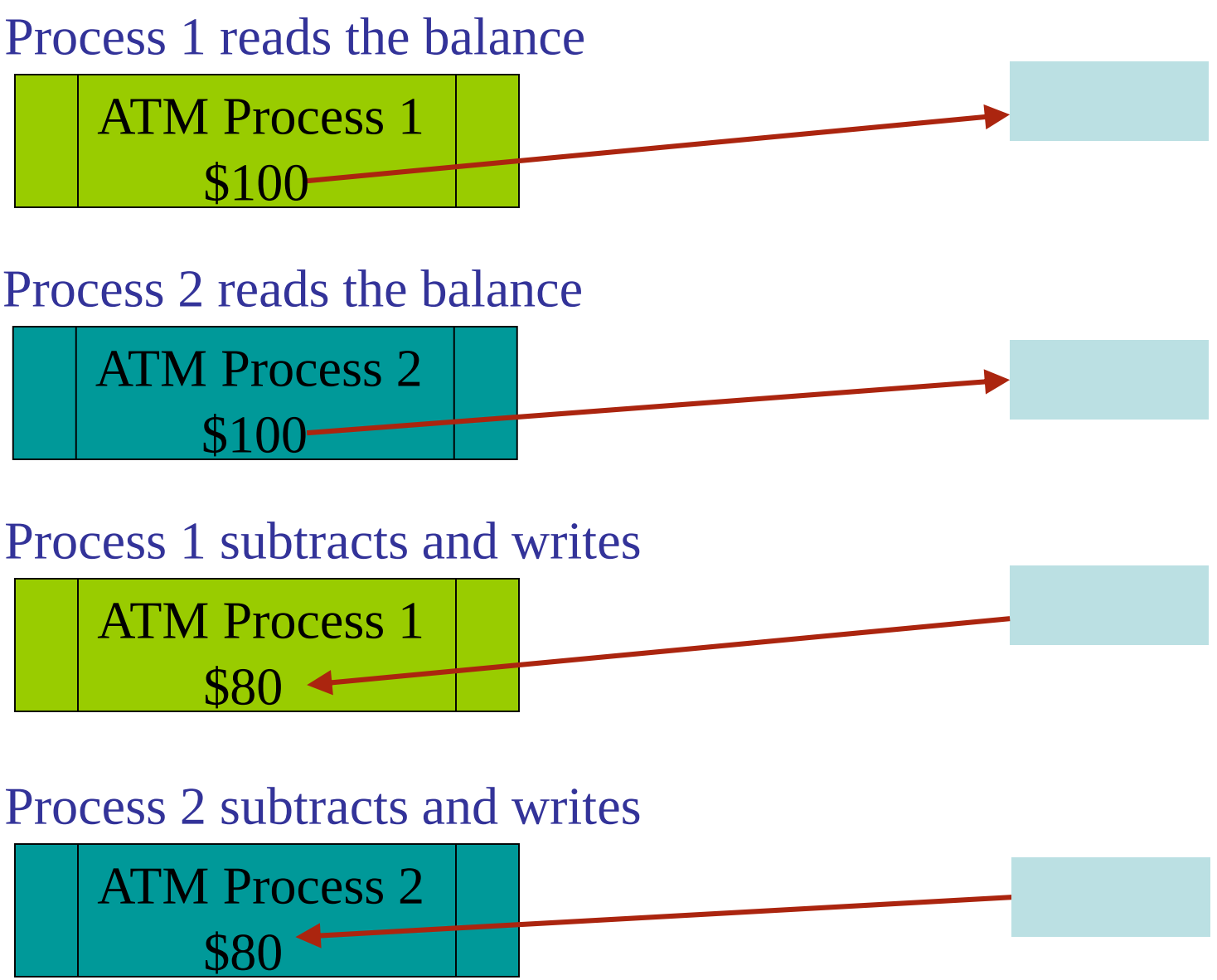
Possible Complications

- Concurrently running processes can share data and/or resources
- What if multiple processes access the same resource at the same time?
- This is most likely a disaster
  - aka, “Race Condition”, “Oops”, or “aw carp”

The Classic Example

- Two ATM machines
  - Each withdrawing $20 from same account

- To update bank account balance
  - Read current balance into memory
  - Subtract $20
  - Write new balance to the bank account


Bank Balance
Process 1 reads the balance
ATM Process 1
$100
Process 2 reads the balance
ATM Process 2
$100
Process 1 subtracts and writes
ATM Process 1
$80
Process 2 subtracts and writes
ATM Process 2
$80

Race conditions?

- Why are race conditions hard to detect?
  - They may only ever show up once

- Another way of saying it: “A race hazard (or race condition) is a flaw in a system or process where the output exhibits unexpected critical **dependence** on the relative timing of events.”

Wikipedia

Aw carp

- Most infamous race condition:
  - http://en.wikipedia.org/wiki/Therac_25
- People could outrun the system
  - the system was counting on a normally slow human, and didn't take into account people learning to use the system faster

Lesson

- Concurrent update situation
  - 2+ processes accessing resource concurrently
  - At least one process might write
- **Must provide access control**
  - If one process is writing, no other process should access (read OR write) the resource
- "Locks" solve these problems
  - Only process owning lock may access (r/w) resource
  - Many ways to do locking (taught in other courses)
  - Locking usually requires support from the OS
    - But you can do it in software, too

Lesson – use a lock file

```
do
{
   lock_fd = open(lock_file_path, O_WRONLY | O_CREAT | O_EXCL, 0644);

   if (lock_fd == -1)
   {
      if (errno == EEXIST)
      {
         // File already exists - wait a while, then try again
         sleep(1);
      }
      else
      {
         // An unexpected error – bail out
         perror("Couldn't open lock file\n");
         return(-1);
      }
   }
}
while (lock_fd == -1);
```

End