LECTURE 3

Translation

PROCESS MEMORY

There are four general areas of memory in a process.

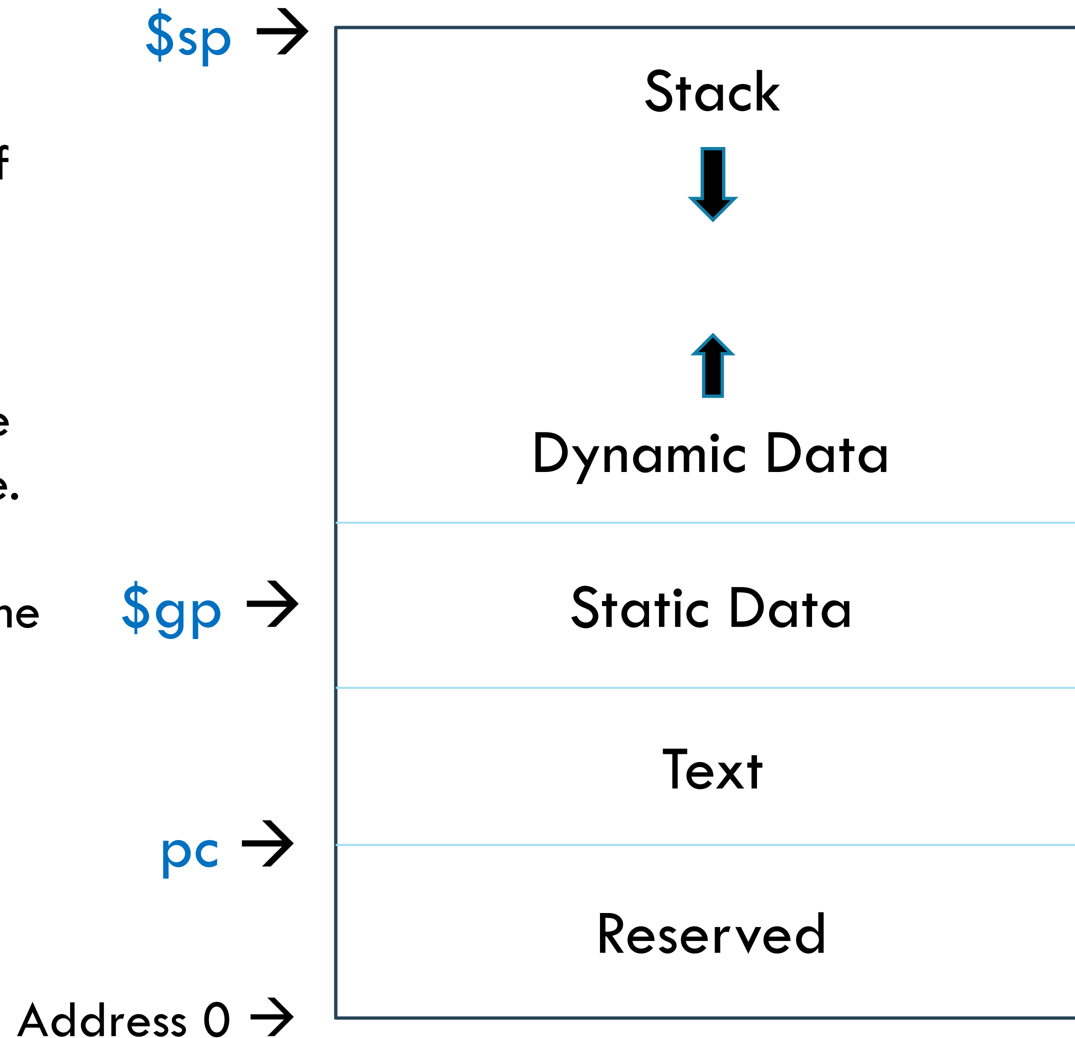
- The **text** area contains the instructions for the application.
  - Text area may contain data (constants), accessed using PC-relative addressing mode
- The **static data** area is also fixed in size and contains:
  - Global variables, static local variables, string and sometimes floating-point constants
- The **run-time stack**
  - Contains *activation records* (information associated with a function invocation).
  - Saved values of callee-saved registers
  - Local variables and arguments not allocated to registers.
  - Space for the maximum words of arguments passed on stack to other functions.
- The **heap** contains dynamically allocated data
  - e.g. data allocated by the new operator in C++ or malloc function call in C.

ORGANIZATION OF PROCESS MEMORY

Here is MIPS convention for allocation of memory.

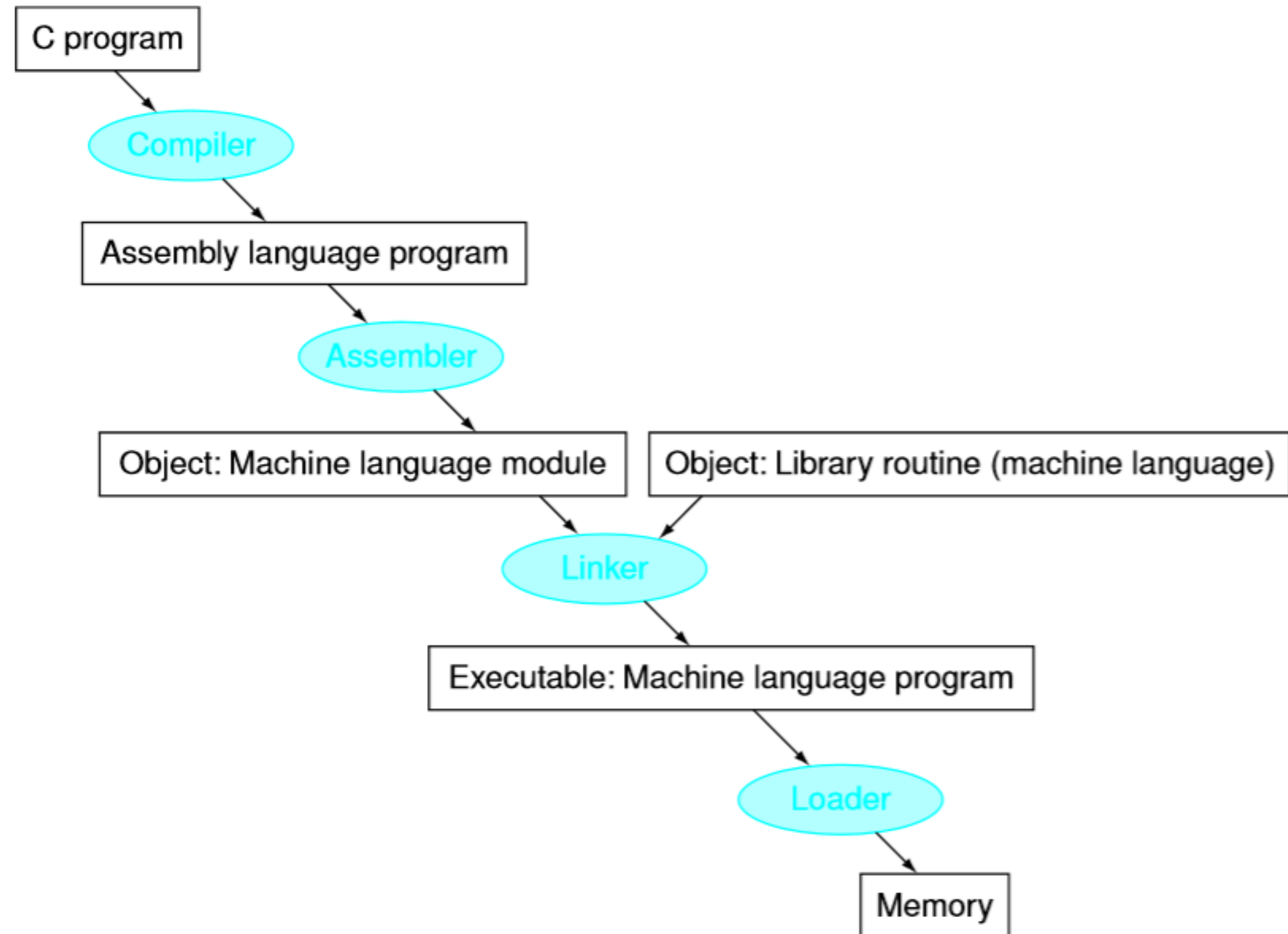
The stack starts at the higher-end of memory and grows downward, while the heap grows upward into the same space.

The lower end of memory is reserved. The text segment follows, housing the MIPS machine code.



TRANSLATION PROCESS

- Preprocessing
- Compiling
- Assembling
- Linking
- Loading



PREPROCESSING

Some preliminary processing is performed on a C or C++ file.

- Definitions and macros
- File inclusion
- Conditional compilation

Try g++ with the -E option!

COMPILING

Compiling is referred to as both the entire translation process from source file to executable or the step that translates a source file in a high-level language (sometimes already preprocessed) and produces an assembly file.

Compilers are also responsible for checking for correct syntax, making semantic checks (e.g., to locate bugs and vulnerabilities), and performing optimizations to improve performance, code size, and energy usage.

ASSEMBLING

Assemblers take an assembly language file as input and produce an object file as output.

Assembling is typically accomplished in two passes.

- First pass: stores all of the identifiers representing addresses or values in a table as there can be forward references (e.g., jump to the end of the loop).
- Second pass: translates the instructions and data into bits for the object file.

THE OBJECT FILE

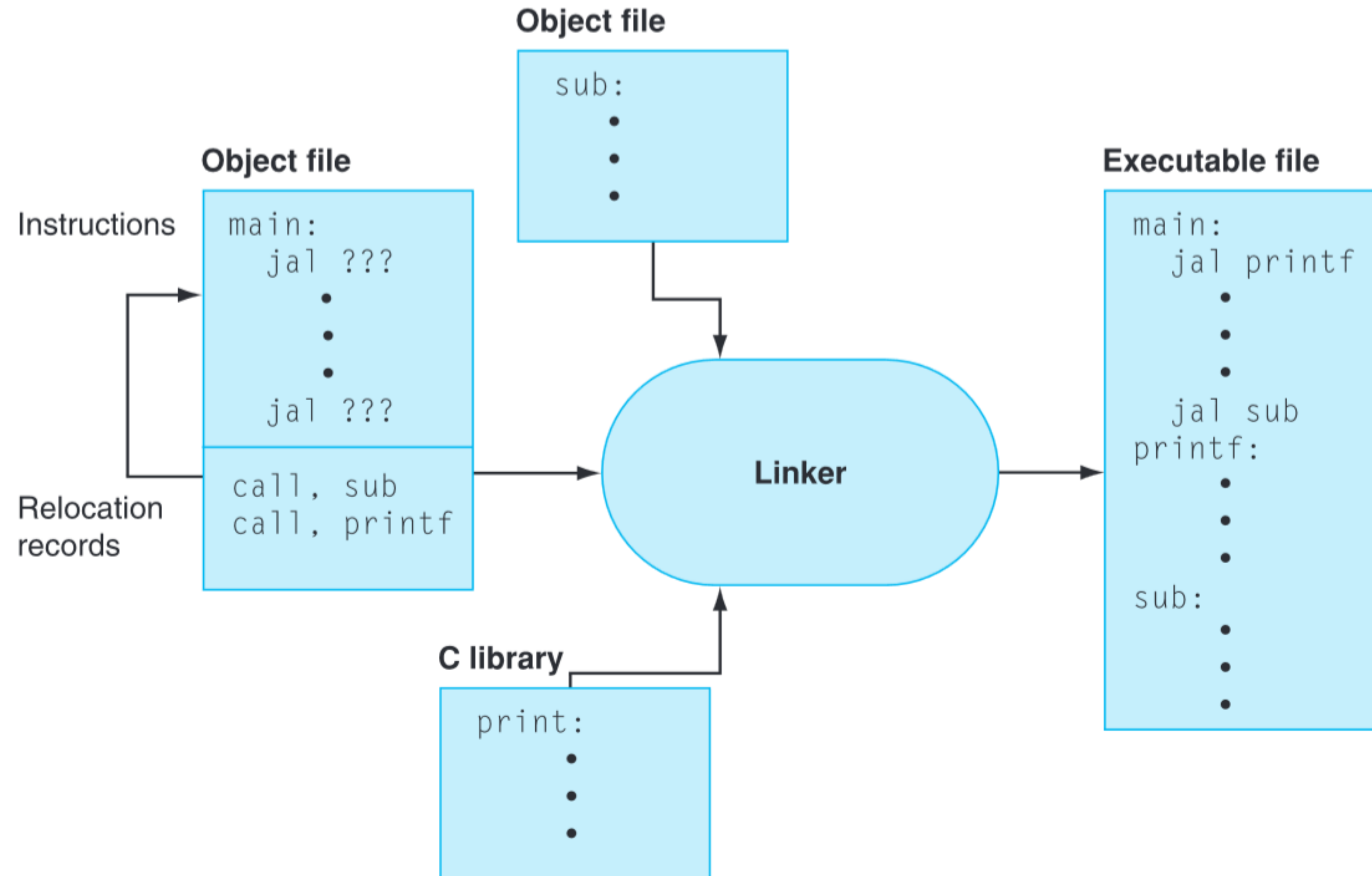
The object file (in the format of ELF, PE, etc) contains:

- An *object file header* describing the size and position of the other portions of the object file.
- The *text segments* containing the machine instructions.
- The *data segments* containing the data values.
- Relocation information identifying the list of instructions and data words that depend on absolute addresses (e.g., call printf function from your code).
- A *symbol table* containing global labels and associated addresses in object file and the list of unresolved references.
- Debugging information to allow a symbolic debugger to associate machine instruction addresses with source line statements and data addresses with variable names.

LINKING

- Linkers take object files and object libraries as input and produce an executable file as output.
- Linkers also resolve external references by either finding the symbols in another object file or in a library.
- The linker aborts if any external references cannot be resolved.
- A generated executable is either loadable at a fixed address or many addresses (position-independent executable)
  - PIE binaries are essential for address-space layout randomization, a security defense

LINKING



LOADING

The loader copies the executable file (or a portion of it) from disk into memory so it can start executing.

- Reads the executable file's header to determine segment sizes.
- Allocates the address space for the process (text, data, heap, and stack segments).
- Copies the code into the text segment and data into the static data segment.
  - It also inserts the code to load a library/resolve a symbol if necessary (e.g., glibc)
- Copies arguments passed to the program onto the stack.
- Initializes the machine registers and stack pointer.
- Jumps to a start-up routine that will call the main function.
  - Main function is NOT the first piece of code executed

GCC EXAMPLE

Let's say I have three files – a class declared in exp.h and defined in exp.c, as well as a main.c file which uses the class.

```
$ ls
exp.c exp.h main.c
$ gcc -c exp.c
$ gcc -c main.c
$ ls
exp.c exp.h exp.o main.c main.o
$ gcc main.o exp.o -o exp_prog
exp.c exp.h exp.o exp_prog main.c main.o
```

`$ gcc -c exp.c` / `$ gcc -c main.c`: Preprocessing, Compiling, and Assembling the source code individually. Result is an object file.

`$ gcc main.o exp.o -o exp_prog`: Linking the object files together into an executable file.

You can check out the pre-processed version of your code with the –E option. Prints to stdout.
You can check out the assembly version of your code with the –S option. Check the *filename.s* file.

STORED PROGRAM CONCEPT

- Memory can contain both instructions and data and the computer is instructed to start executing at a specific location.
- Different programs can be loaded in different locations and the processor can switch between processes very quickly.