Low-Level Programming in C and Multithreaded Programming

TDDD56

Self-study material for students not familiar with C, operating systems, or Pthreads programming

Read this before Lab 0 (Wednesday first week)

Christoph Kessler

PELAB / IDA
Linköping University
Sweden

2025

Why *native* programming languages?

(e.g. C/C++, Fortran, Rust, Ada, Assembly languages)?

- Low-level programming in system software
  - Necessary for direct access to hardware devices,
    e.g. in device drivers, embedded systems
  - Early operating systems were implemented in assembly language
    (and some low-level parts of modern OS still are).
- C is the dominating language for system programming since the 1970s.
  - Also used a lot in high-performance and embedded computing
  - No new microprocessor ships without a C compiler.
  - Many “high-level” programming languages are compiled to C.
- Resource-aware programming
  for performance-critical code and energy-efficient computing
  - HPC and machine learning libraries
  - Graphics
  - Realtime systems, Embedded systems
  - …

Example: Java vs. C

Java
(similarly: **Python**)

- For application programming only
- Design goals:
  - Programmer productivity
  - Safety
  - Hardware completely hidden
- Comfortable
  - E.g. automatic memory management
- Protection (to some degree)
  - E.g., array bound checking
- Slow
- Time-unpredictable

C

- For system programming mainly
- Design goals:
  - Direct control of hardware
  - High performance / real-time
  - Minimalistic design
- Less comfortable
- Little protection (by default)
- ”low-level”
- But also the ”greenest” (most energy-efficient) programming language

Programmer Productivity **Program Productivity**

Programming in Python&Co. is Not Sufficient for High-Performance Computing

- Resource-aware programming and use of native programming languages can give orders of magnitude in speedup
- Exploit multiple levels of parallelism and optimizations

Example:
Matrix-Multiply: relative speedup to a Python version (18 core Intel Xeon CPU)

Version	Speedup	Optimization
Python	1	
C	47	Rewrite in a static, compiled ("native") progr. language
C with parallel loops	366	Extract multi-core parallelism (OpenMP)
C with loops and memory optimization	6,727	Loop tiling for data locality
Loop vectorization using Intel AVX SIMD instructions	62,806	Extract SIMD parallelism

Data source: C. Leiserson et al. "There's plenty of room at the top: What will drive computer performance after Moore's law?" *Science* 368(6495): pp. 1-7, June 2020.
Table source: J. Hennessy, D. Patterson: "A New Golden Age for Computer Architecture." *Communications of the ACM* 62(2):48-60, Feb. 2019.

Relative Performance and Energy Usage with Different Programming Languages

Table 4
Normalized global results for Energy, Time, and Memory.

	Total				
	Energy (J)		Time (ms)		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Source:

R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. Fernandes, J. Saraiva: "**Ranking programming languages by energy efficiency**". *Science of Computer Programming* 205, 2021. https://doi.org/10.1016/j.scico.2021.102609

- Based on a set of 10 benchmark applications
- Time and energy are strongly correlated here (fixed clock frequency)
- In both aspects, C outperforms Python by 2 orders of magnitude!

A Short History of C

- C was developed in the early 1970's by Dennis Ritchie at Bell Labs
  - Objective: structured but flexible programming language
    e.g. for writing device drivers or operating systems
  - Used for implementing the Unix OS
  - Book 1978 by Brian Kernighan and Dennis Ritchie ("K&R-C")
- "ANSI-C" 1989 standard by ANSI ("C89")
  - *The* C standard for many programmers (and compilers...)
  - Became the basis for standard C++
  - Java borrowed much of its syntax
  - The GNU C compiler (gcc) implemented a superset ("GNU-C")
- "C99" standard by ISO, only minor changes
- "C11" (ISO) – multithreading support added
- …

A short introduction to system programming in C

Remark: This is not a complete language tutorial. Instead, we focus on the C issues often perceived as *difficult* by programmers used to Python or Java.

Pointer programming, Storage Classes, Memory; Compiling, Linking, Loading

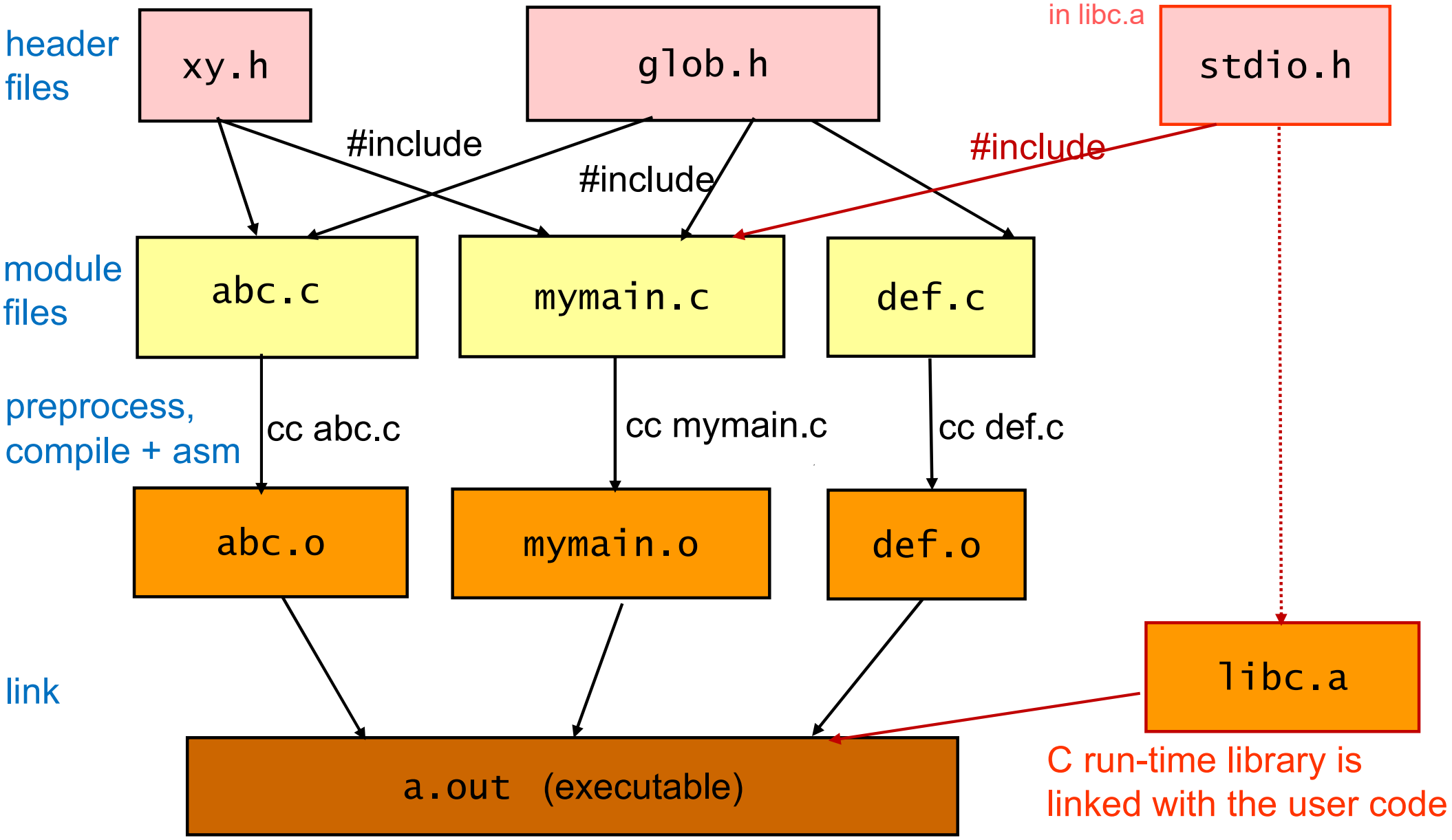
Overview

- Common organization of a C program
- Data and function definitions
- Pointers
- Static and automatic variables
- Dynamically allocated memory
- Linking
- Debugging (as time permits)

Common organization of a C program

- A C program is a collection of C source files
  - ***Module files*** (extension `.c`)
    - contain *implementations* of functionality (similar .java)
  - ***Header files*** (`.h`)
    - contain *declarations* of *global* variables, functions, types
    - #include'd from all module files that need to see them, using the ***C preprocessor*** (missing in Java)
- When a module file is compiled successfully, an ***object code file*** (*object module, binary module*) is created (`.o`, corresponds to Java .class files)
- An ***executable*** (program) is a single binary file (`a.out`) built by the ***linker*** by merging all needed object code files
- There must be exactly one function called `main()`

Compiling and Linking for Multi-Module C Programs



Common organization of a C program (cont.)

Example

glob.h

```
// a comment: header

/*comment: declaration
  of globally visible
  functions
  and variables: */

extern int incr(int);
extern int initval;
```

abc.c

```
#include ”glob.h”

// definition of var. initval:
int initval = 17;

// definition of func. incr:
int incr( int k )
{
  return k+1;
}
```

mymain.c

```
#include <stdio.h>
#include ”glob.h”

int counter; // locally def.

void main( void )
{
  counter = initval;
  printf(“result: %d”,
         incr(counter)
         );
}
```

Note the difference:

Declarations announce signatures (names+types).

Definitions declare (if not already done) AND allocate memory.

Header files should never contain definitions!

Side note for Python programmers:
In C, the code formatting (indentation, whitespace, linebreaks etc.) does (almost) not matter at all for the compiler. But it can help the human reader …

Data types in C

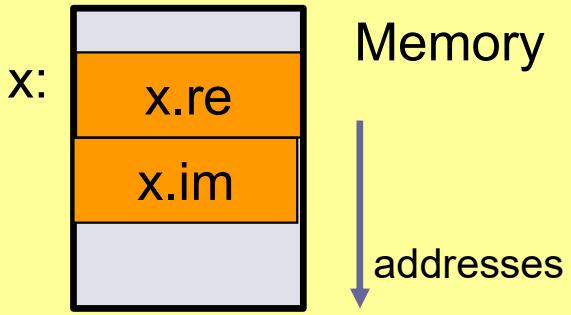
- **All data objects in a C program must be statically typed**
  - Declared by programmer or inferred by compiler (using rules)
- **Primitive types**
  - integral types: **char**, **short**, **int**, **long**; **enum**
    - can be signed or unsigned, e.g. **unsigned int** counter;
    - sizes are implementation dependent (compiler/platform), use **sizeof**(*datatype*) operator to write portable code
  - floatingpoint types: **float**, **double**, **long double**
  - pointers
- **Composite data types**
  - arrays
  - structs
  - unions
- Programmer can also define new type names with **typedef**

Constants and Enumerations

- **Constant variables:**
  - `const int red = 2;`
  - `const int blue = 4;`
  - `const int green = 5;`
  - `const int yellow = 6;`
- **Enumerations:**
  - `enum { red = 2, blue = 4, green, yellow } color;`
    `color = green;` // expanded by compiler into: color = 5;
- **With the preprocessor:**
  - symbolic names, textually expanded before compilation
    - `#define red 2`
    - ... (no =, no semicolon)
- Stylistic issue: In C, constants are often capitalized: RED, BLUE, ...

Composite data types (1): structs

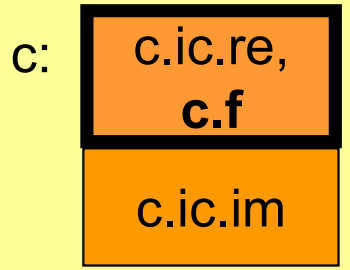
```
struct My_IComplex  {
   int re, im;
};
struct My_IComplex x;     //  definition of x
typedef struct My_IComplex icplx;    // introduces new type name icplx
icplx y;

void main ( void )
{
   x.re = 2;
   y.im = 3 * x.re;
   printf("x needs %d bytes, an icplx needs %d bytes\n",
          sizeof( x ), sizeof (icplx) );
}

//  Remark:  the sizeof operator is evaluated at compile time.
```



Composite data types (2): unions

- Unions implement variant records:
  all attributes share the same storage (start at offset 0)
- Unions break type safety
  - can interpret the same memory location with different types
- If handled with care, useful in low-level programming

```
union My_flexible_Complex  {
    struct My_IComplex ic; // integer complex (re, im) – see above
    float f;   // floatingpoint value – overlaps with ic
}  c;

c.ic.re = 1141123417;

printf(”Float value interpreted from this memory address contents is
%f\n”, c.f );
//  writes 528.646057
```



Composite data types (3): Arrays

- Declaration/definition very similar to Java:

```
int a[20];
int b[] = { 3, 6, 8, 4, 3 };
icplx c[4];
float matrix [3] [4];
```

- Addressing:

  Location of element a[i] starts at: (address of a) + i * *elsize*
  where *elsize* is the element size in bytes

- Use:

```
a[3] = 1234567;
a[21] = 345;        // ??, there is no array bound checking in C
c[1].re = c[2].im;
```

- Dynamic arrays: see later
- Arrays are just a different view of pointers

Pointers

- Each data item of a program exists somewhere in memory
  - Hence, it has a (start) **memory address**.
- A variable of type *pointer_to_type_X* may take as a value the memory address of a variable of type *X*
  - `int a;`
  - `int *p;` // defines just the pointer p, not its contents!
  - `p = 0x14a236f0;` // initialize p
- Dereferencing a pointer with the * operator:
  - `*p = 17;` // writes 17 to whatever address contained in p
- The address of a variable can be obtained by the & operator:
  `p = &a;` // now, p points to (i.e., contains) address of a
- Use the NULL pointer (which is just an alias for address 0) to denote invalid addresses, end of lists etc.

Pointer arithmetics

- Integral values can be added to / subtracted from pointers:

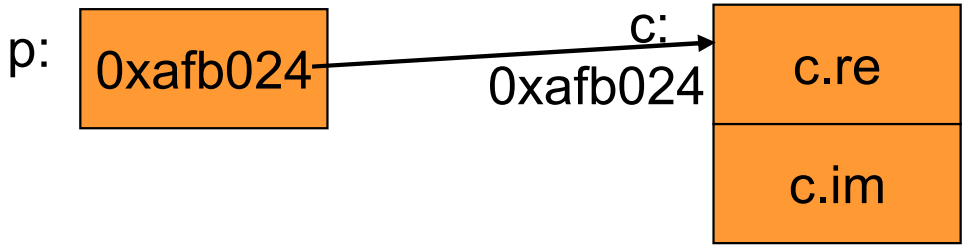
```
int *q = p + 7;
// new value of q is  (value of p) + 7 * sizeof(pointee-type of p)
```

- Arrays are simply constant pointers to their first element:
  - Notation `b[3]` is ”syntactic sugar” for `*(b + 3)`
  - `b[0]` is the same as `*b`

- A pointer can be subtracted from another pointer:
  - `unsigned int offset = q - p;`

Pointers and structs

- `struct My_IComplex { int re, im; } c, *p;`
  `p = &c;`
  - p is a pointer to a struct



- `p->re` is shorthand for `*(p + (`*offset of re*`) )`
  - Example: as p points to c,
    `&(p->re)` is the same as `&(c.re)`
  - Example: `elem->next = NULL;`

Why do we need pointers in C?

- Defining recursive data structures (lists, trees ...) – as in Java
- Argument passing to functions
  - simulates reference parameters – missing in C (not C++)
    - `void foo ( int *p ) { *p = 17; }`
      Example call: `foo ( &i );`
- Arrays are pointers
  - Handle to access dynamically allocated arrays
  - A 2D array is an array of pointers to arrays:
    - `int m[3][4];` // m[2] is a pointer to start of third row of m
      - hence, m is a pointer to a pointer to an int
- For representing strings – missing in C as basic data type
  - `char *s=”hello”;` // s points to char-array {’h’,’e’,’l’,’l’,’o’, 0 }
- For dirty hacks (low-level manipulation of data)

Pointer type casting

- Pointer types can be casted to other pointer types
  - ```
    int i = 1147114711;
    int *pi = &i;
    printf("%f\n", *((float *)pi));  // prints 894.325623
    ```
  - All pointers have the same size (1 address word)
  - But *no conversion* of the *pointed data*! (cf. unions)
    - Compare this to: `printf("%f\n", (float) i);`
  - A source of type unsafety,
    but often needed in low-level programming

- **Generic pointer type:** `void *`
  - Pointee type is undefined
  - Always requires a pointer type cast before dereferencing
    ```

Pointers to functions (1)

- Function declaration
  - `extern int f( float );`
- Function call: f(x)
  - f is actually a (constant) pointer
    to the first instruction of function f in program memory
  - Call f(x) dereferences pointer f
    - push argument x;
      save PC and other reg's;
      PC := f;
- Function pointer variable
  - `int (*pf)(float);` // pf is a pointer to a function
    // that takes a **float** and returns an **int**
  - `pf = f;` // pf now contains start address of f
  - `pf(x);` // or `(*pf)(x)` dereferencing (call): same effect as `f(x)`

Pointers to functions (2)

- Most frequent use: generic functions

Example: Ordinary sort routine

```
void bubble_sort( int arr[], int asize )
{  int i, j;
   for (i=0; i<asize-1; i++)
      for (j=i+1; j<asize; j++)
         if ( arr[i]  >  arr[j] )
            ...  // interchange arr[i] and arr[j]
}
```

- Need to rewrite this for sorting in a different order?
- Idea: Make bubble_sort generic in the compare function

Pointers to functions (3)

- Most frequent use: generic functions

- Example: Generic sort routine

```
void bubble_sort( int arr[], int asize,
                  int (*cmp)(int,int) )
{  int i, j;
   for (i=0; i<asize-1; i++)
      for (j=i+1; j<asize; j++)
         if ( cmp ( arr[i] , arr[j] ) )
            ...  // interchange arr[i] and arr[j]
}
```

```
int gt ( int a, int b )  {
   if (a > b)  return 1;
   else return 0;
}
```

```
bubble_sort ( somearray, 100, gt );
bubble_sort ( otherarray, 200, lt );
```

Storage classes in C (1)

- **Automatic variables**
  - Local variables and formal parameters of a function
  - Exist once per call
  - Visible only within that function (and function call)
  - Space allocated automatically on the function's stack frame
  - Live only as long as the function executes

```
int *foo(void)  // function returning a pointer to an int.
{
    int t = 3;   // local variable
    return &t;   // ??  t is deallocated on return from foo,
     // so its address should not make sense to the caller...
}
```

Storage classes in C (2)

- **Global variables**
  - Declared and defined outside any function
  - Unless made globally visible by an **extern** declaration, visible only within this module (file scope)
    - `int x;` // at top level – outside any function
    - **`extern`** `int y;` // y seen from all modules; only declaration
    - `int y = 9;` // only 1 definition of y for all modules seeing y
  - Space allocated automatically when the program is loaded

- **Static variables**
  - **`static int`** `counter;`
  - Allocated once for this module (i.e., not on the stack) even if declared within a function!
  - Value will survive function return: next call sees it

Dynamic allocation of memory in C

- **`malloc`**(*N*)
  - allocates a block of *N* bytes on the heap
  - and returns a generic (**void** *) pointer to it;
  - this pointer can be *type-casted* to the expected pointer type
  - Example:
    ```
    icplx *p = (icplx *) malloc( sizeof(icplx) );
    ```
- **`free`**(p)
  - deallocates heap memory block pointed to by p
- Can be used e.g. for simulating ***dynamic arrays***:
  - Recall: arrays are pointers
  - ```
    int *a = (int *) malloc(k * sizeof(int));
    a[3] = 17;
    ```
    ```

Measuring Time

Using C standard library functions declared in system header file <time.h>:

```
#include <time.h>
```

```
struct timespec {
   time_t   tv_sec;    /* seconds */
   long     tv_nsec;   /* nanoseconds */
};
```

Important functions:

int **clock_getres**(clockid_t *clockid*, struct timespec **res*);

int **clock_gettime**(clockid_t *clockid*, struct timespec **tp*);

int **clock_settime**(clockid_t *clockid*, const struct timespec **tp*);

Measuring Time Example

```
struct timespec {
   time_t   tv_sec;    /* seconds */
   long     tv_nsec;   /* nanoseconds */
};
```

```
#include <time.h>
…
clockid_t clk = CLOCK_REALTIME;
struct timespec *time1, *time2;
time1 = (struct timespec *)malloc(sizeof(struct timespec));
time2 = (struct timespec *)malloc(sizeof(struct timespec));
…
clock_gettime( clk, time1 );
   …  // computation to be timed
clock_gettime( clk, time2 );
printf("time [s]: %f \n",
     (float)(time2->tv_sec - time1->tv_sec)
  + (float)(time2->tv_nsec - time1->tv_nsec) / 1000000000.0 );
```

Why are these two lines important?

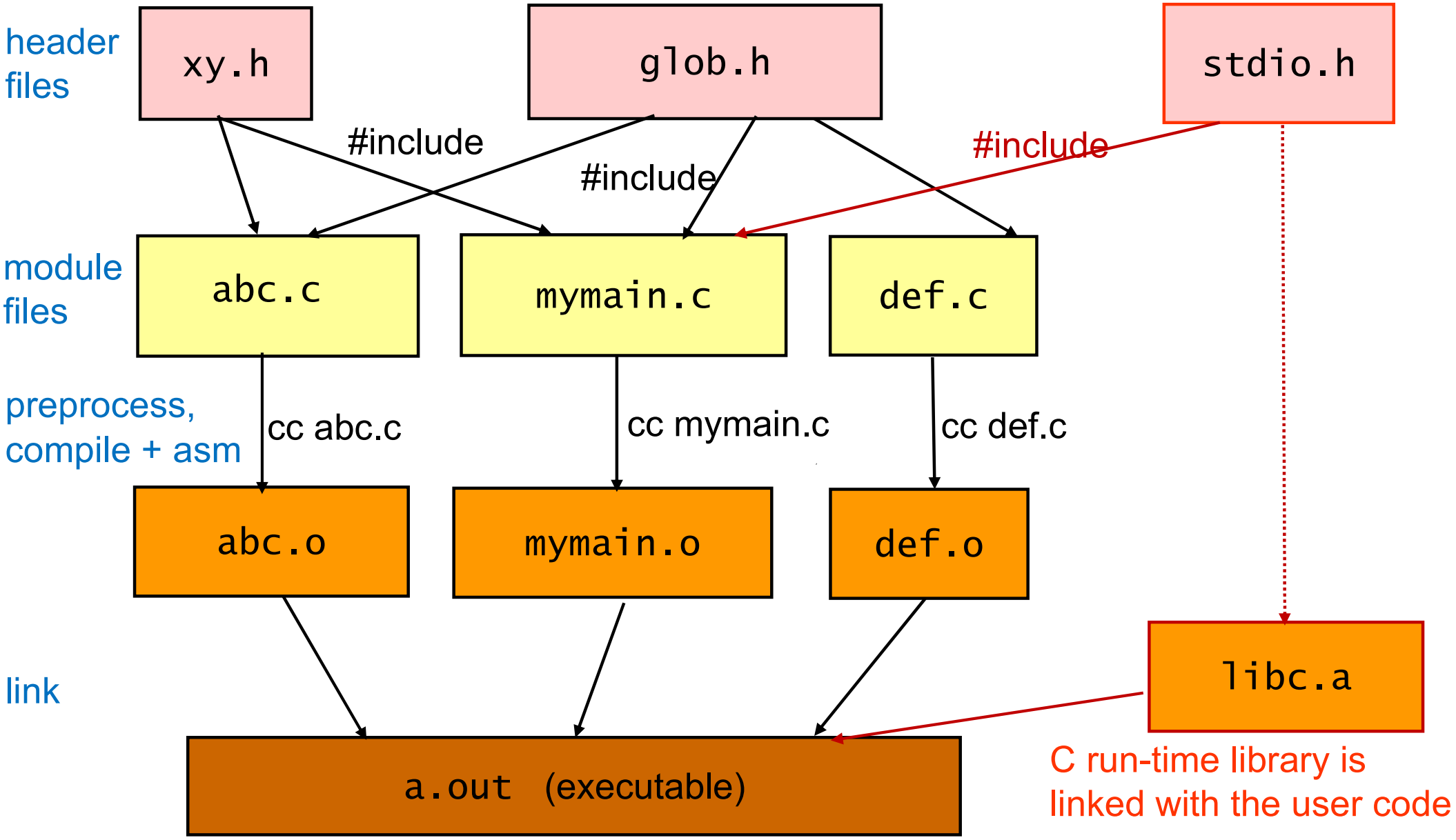
C: There is much more to say...

See any good textbook / tutorial on C programming about:

- Type conversions and casting
- Bit-level operations
- Operator precedence order
- Variadic functions
  (with a variable number of arguments, e.g. printf())
- C standard library
- C preprocessor macros
- I/O in C
- ...

Now: Compiling, Linking, Loading, Debugging

Compiling and Linking
for Multi-Module C Programs

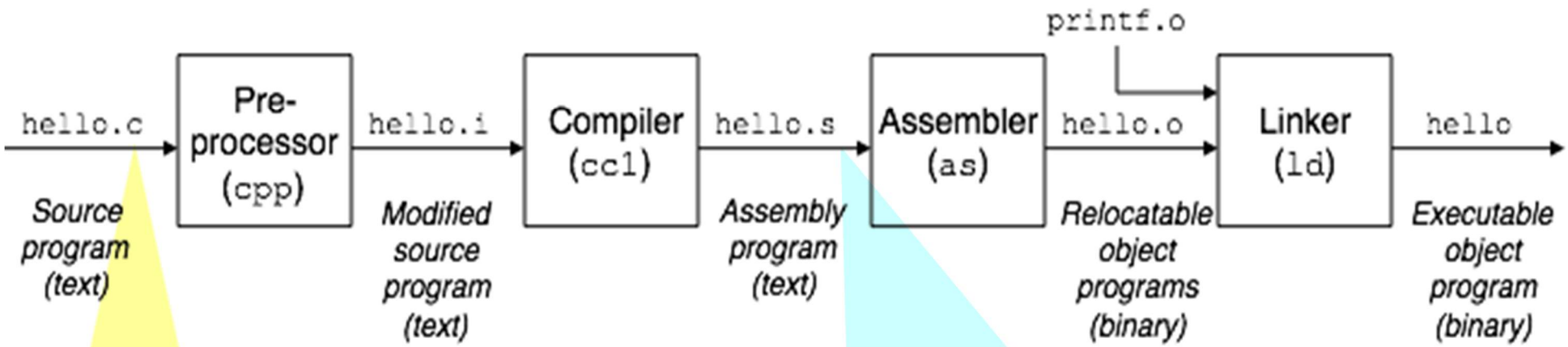


Compiling C Programs

- Examples: GNU C Compiler gcc (open source),
  LLVM C compiler (open source), …
- One command calls preprocessor, compiler, assembler, linker
- **Single module**: Compile and link (build executable):
  - gcc mymain.c — executable: a.out
  - gcc –o myprog mymain.c — -o: rename executable
- **Multiple modules:** Compile separately and link then:
  - gcc –c –o mymain.o mymain.c — -c: compiler only
  - gcc –c –o other.o other.c — compiles other.c
  - gcc other.o mymain.o — call the linker, -> a.out
- **make** (e.g. GNU gmake)
  - automatizes building process for several modules

How to build and execute programs on a real computer

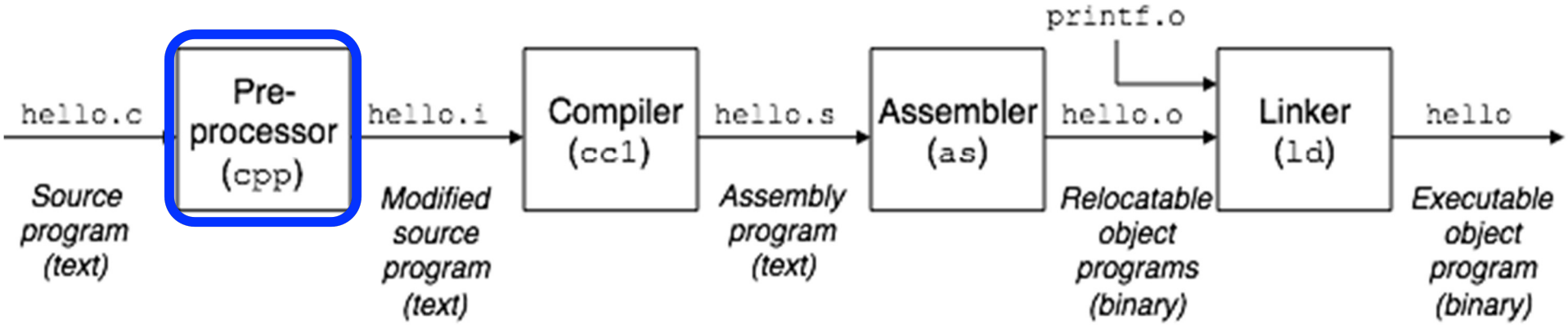
The Compilation Workflow




```c
/* file hello.c */

#include <stdio.h>

int main()
{
   printf("hello, world\n");
}
```

```
      .file   "hello.c"
      .section      .rodata.str1.1,"aMS",@progbits,1
.LC0:
      .string"hello, world"
      .text
      .p2align 4,,15
.globl main
      .type   main, @function
main:
      pushl   %ebp
      movl   %esp, %ebp
      subl   $8, %esp
      andl   $-16, %esp
      subl   $16, %esp
      movl   $.LC0, (%esp)
      call   puts
      leave
      xorl  %eax, %eax
      ret
```

Preprocessing Phase



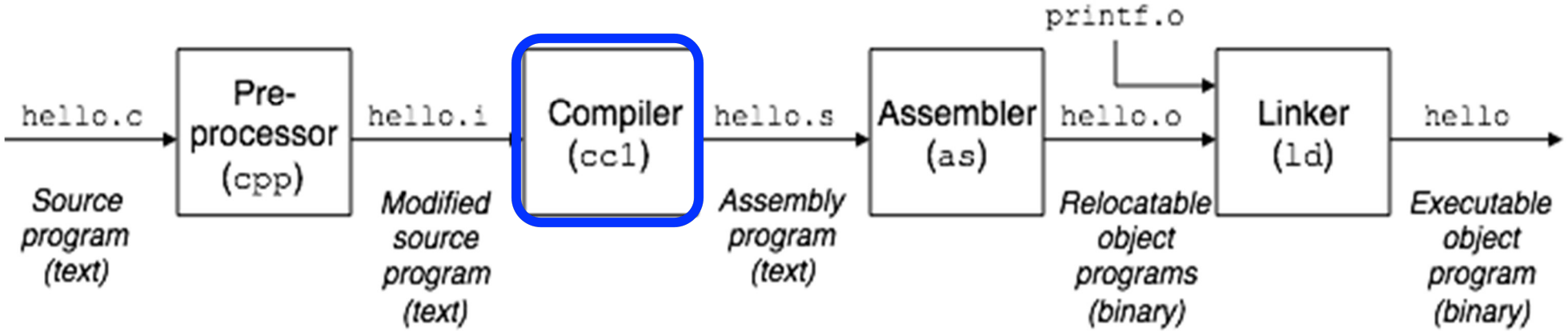
C Preprocessor (cpp):

modifies the original C source program according to **directives** that begin with the # character.

#include <stdio.h>
tells the preprocessor to read the contents of the system library header file stdio.h and to insert it directly into the program text.

Output is another C source program, typically with the .i suffix.

Compilation Phase

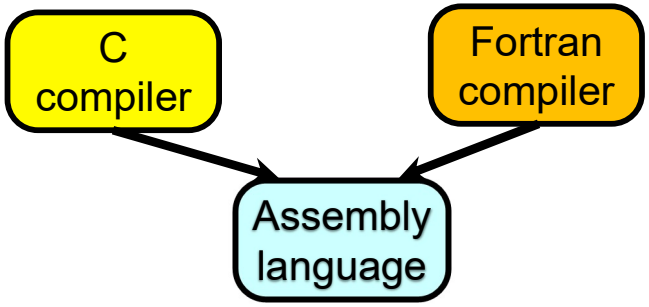


Compilation Phase (**cc1**):

Translate the text file hello.i into the text file hello.s.

hello.s contains an assembly-language program

Each statement in an assembly-language program exactly describes one low-level machine language instruction in a standard text form.



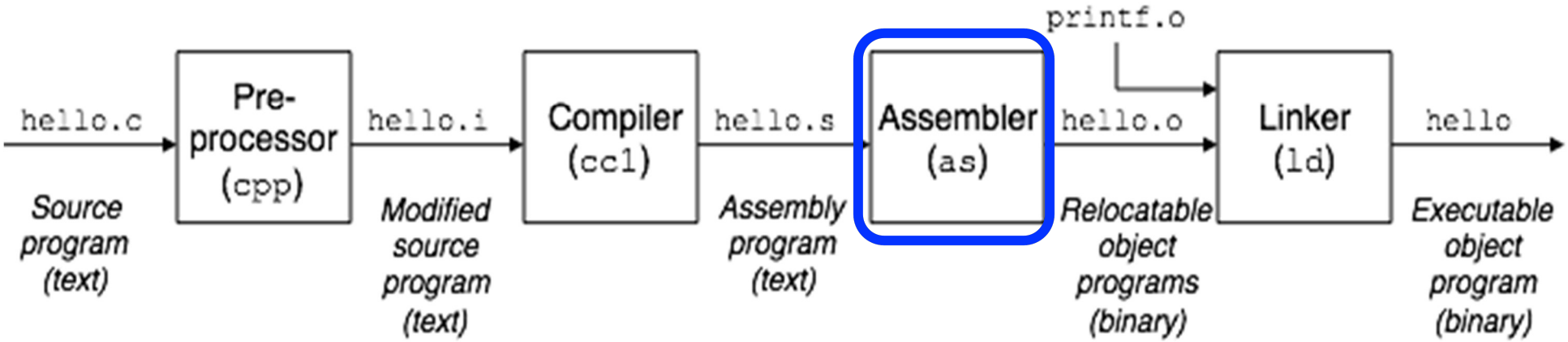
Assembly language provides a common output language for different compilers

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
```

hello.s (in x86 assembly language)

```
    .file  "hello.c"
    .section .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string   "hello, world"
    .text
    .p2align 4,,15
.globl main
    .type     main, @function
main:
    pushl     %ebp
    movl      %esp, %ebp
    subl $8, %esp
    andl$-16, %esp
    subl $16, %esp
    movl      $.LC0, (%esp)
    call  puts
    leave
    xorl %eax, %eax
    ret
    .size main, .-main
    .section .note.GNU-stack,"",@progbits
    .ident    "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
```

Assembly Phase



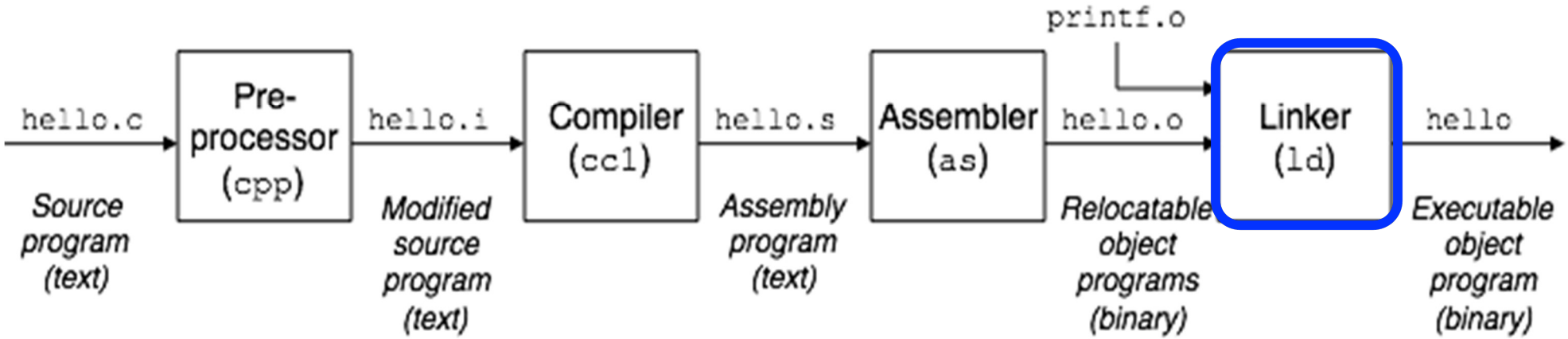
Assembler (as):

translates the text-based assembly program into machine language instructions,
packages them in a form known as a relocatable object program, and stores the result in the object file hello.o

```
a4 f0 dd 07
7c 65 a6 b2
06 0f c3 dd
a2 ff bd 87
…
```

hello.o is a **binary file** (object file) whose bytes encode machine instructions and -data rather than characters.
Binary files use a system-specific binary file format, e.g. ELF, COFF.

Linking Phase



Linker (ld):

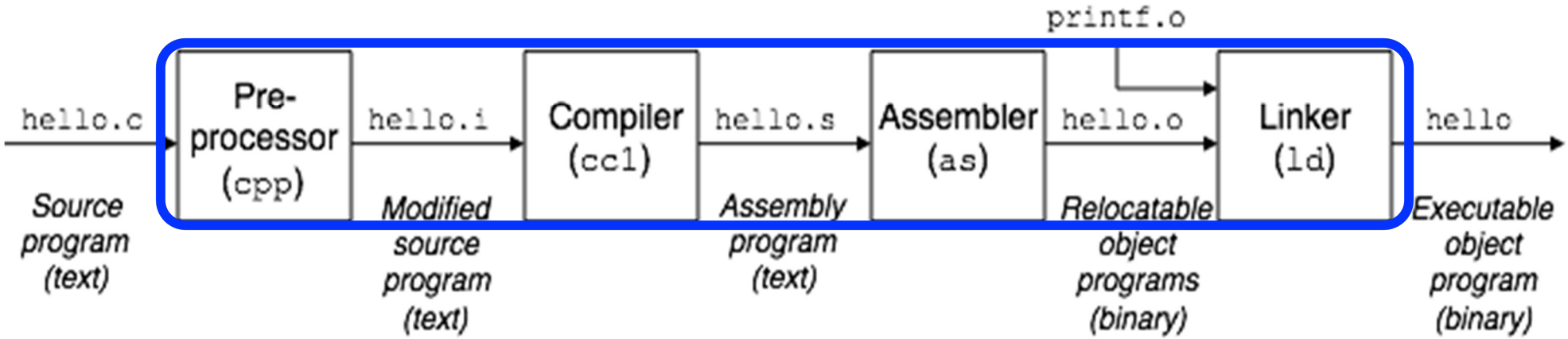
merges pre-compiled object files to a single one.

The result is an ***executable object file*** that is ready to be **loaded** into memory (using the OS loader) and executed by the system.

The hello program uses the **printf** function, which is part of the *standard C library*. This function resides in a separate precompiled object file (e.g. printf.o or in libc.a) that has to be **merged** with hello.o.

Also some additional code and data (program startup code, C runtime system, etc., in libc.a) is added by the linker.

Calling the entire toolchain



E.g., gcc hello.c –o hello

(here, for the GNU C compiler gcc)

calls cpp, cc1, as, ld for single-module program hello.c

For automatizing the build process of multi-module programs, build-tools like **make** or IDEs like **ECLIPSE** are convenient.

Running an Executable Program

```
/* hello.c */
#include <stdio.h>

int main()
{
   printf("hello, world\n");
}
```

Preprocess
Compile
Assemble
Link

hello

```
a4 f0 dd 07
7c 65 a6 b2
06 0f c3 dd
a2 ff bd 87
…
```

Executable Object File
(binary)

In a Unix system, a ***shell*** is a program which acts as a **command-line interpreter**:

The shell prints a *prompt*,
waits for the user to type in a command,
and then performs the command.

```
unix> ./hello
hello, world
unix>
```

Input

Output

If it is not a built-in shell command, then the shell assumes it is an executable file and that it should **load and run** it.

Background: How the Linker Works

- Read all object modules to be linked
  (including library archive modules if necessary)
- Merge the code, data, stack/heap segments of these
  into a single code, data, stack/heap segment
- Resolve global symbols (e.g., global functions, variables):
  check for duplicate globals, undefined globals
- Write the resulting object module,
  with a new relocation table
- and mark it executable.

- **Variants of static linking**: (need hardware support)
  - Dynamic linking (on demand at run time, as in Java)
  - Shared libraries

Sources of information

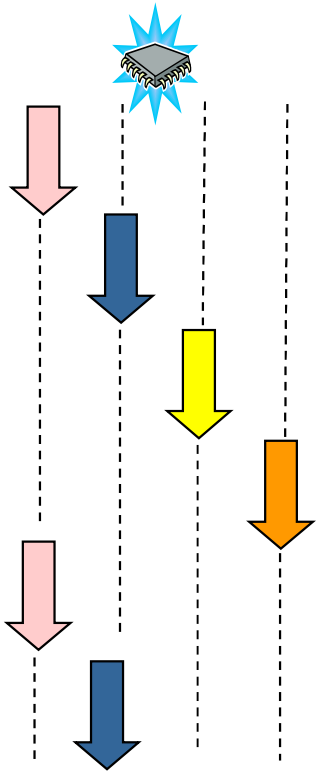
- **General intro to Programming Compiling, Loading, Executing etc.**
  - Y. Patt, S. Patel:
    Introduction to Computing Systems: From Bits & Gates to C & Beyond.
    Mc Graw Hill, 2003.

- **Online reference on C and C++**
  - cppreference.com

- **IDA courses on C and C++**
  - E.g. TDDD38 *Advanced Programming in C++.* Given every semester.

Concurrent Programming with Processes and Threads

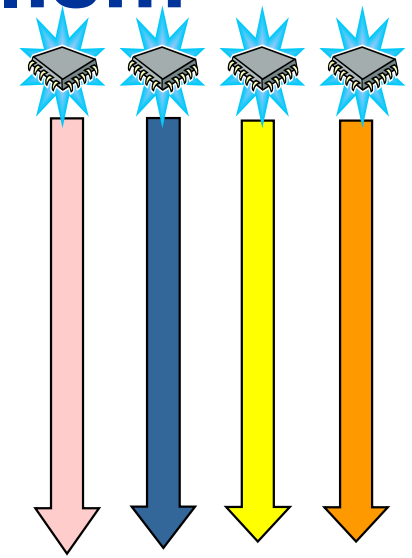
Concurrency vs. Parallelism



Concurrent computing

1 or few CPUs

Quasi-simultaneous execution



Parallel computing

Many CPUs

Simultaneous execution of many / all threads of the *same application*

Common issues:
- threads/processes for overlapping execution
- synchronization, communication
- resource contention, races, deadlocks

Goals of concurrent execution:
- Increase CPU utilization
- Increase responsitivity of a system
- Support multiple users

Central issues: Scheduling, priorities, …

Goals of parallel execution:
- Speedup of 1 application (large problem)

Central issues: Parallel algorithms and data structures, Mapping, Load balancing…

Processes and Threads

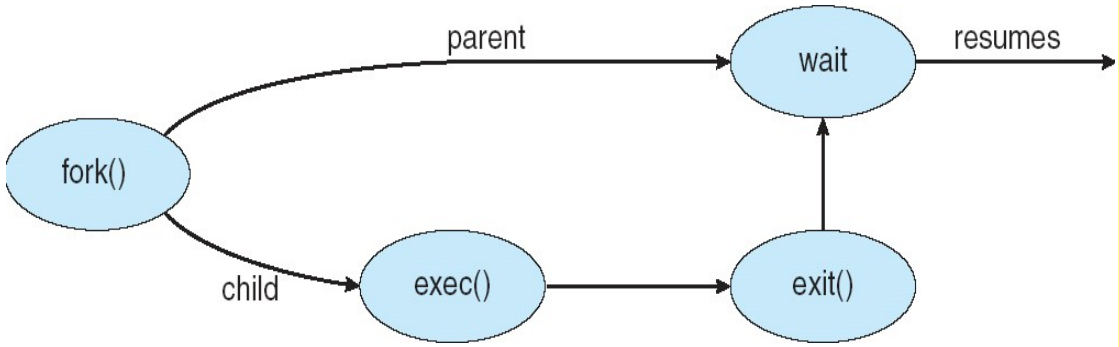
(quick refresher from the Operating Systems course)

Processes

- Processes are the fundamental units of work in a computer
- Processes are instantiations of programs
  - May be started by users (or user processes) or by the operating system
- Process = program + resources
  - Resources:
    - A block of memory allocated by the OS to (”owned by”) the process to accommodate its program code and data,
    - Program state (current position of program counter, register values …)
    - Access to CPU (usually time-shared with other processes, under the control of the OS’ CPU scheduler)
    - Open files
- Processes can execute concurrently (on same or different cores in the CPU) if they reside in memory at the same time
- Processes are units of protection
  - No process can write into another process’s memory nor into the memory area reserved for the OS kernel data structures
  - When needed, request help from the operating system (syscalls)
  - Enforced by hardware mechanisms (mode bit in CPU)

Example: Process Creation in UNIX

- **fork** system call
  - creates new child process
- **exec** system call
  - used after a **fork** to replace the process' memory space with a new program
- **wait** system call
  - by parent, suspends parent execution until child process has terminated



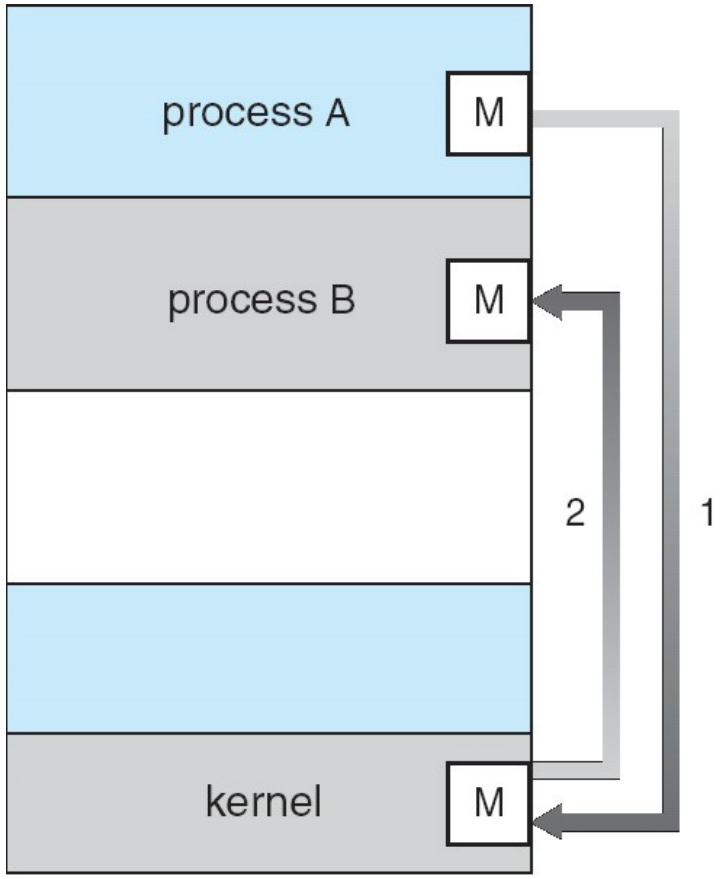
C program forking a separate process

```
int main()
{
    Pid_t  ret;
    /* fork another process: */
    ret = fork();
    if (ret < 0)  {  /* error occurred */
            fprintf ( stderr, "Fork Failed" );
            exit(-1);
    }
    else if (ret == 0)  {  /* child process */
            execlp ( "/bin/ls", "ls", NULL );
    }
    else  { /*parent process: ret=childPID *
            /* will wait for child to complete: *
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```

Parallel programming with processes

- Processes can create new processes that execute concurrently with the parent process
- OS scheduler – also for single-core CPUs
- Different processes share nothing by default
  - Inter-process communication via OS only,
    via shared memory (write/read)
    or message passing (send/recv)
- **Threads** are a more light-weight alternative for programming shared-memory applications
  - Sharing memory (except local stack) by default
  - Lower overhead for creation and scheduling/dispatch
    - E.g. Solaris: creation 30x, switching 5x faster

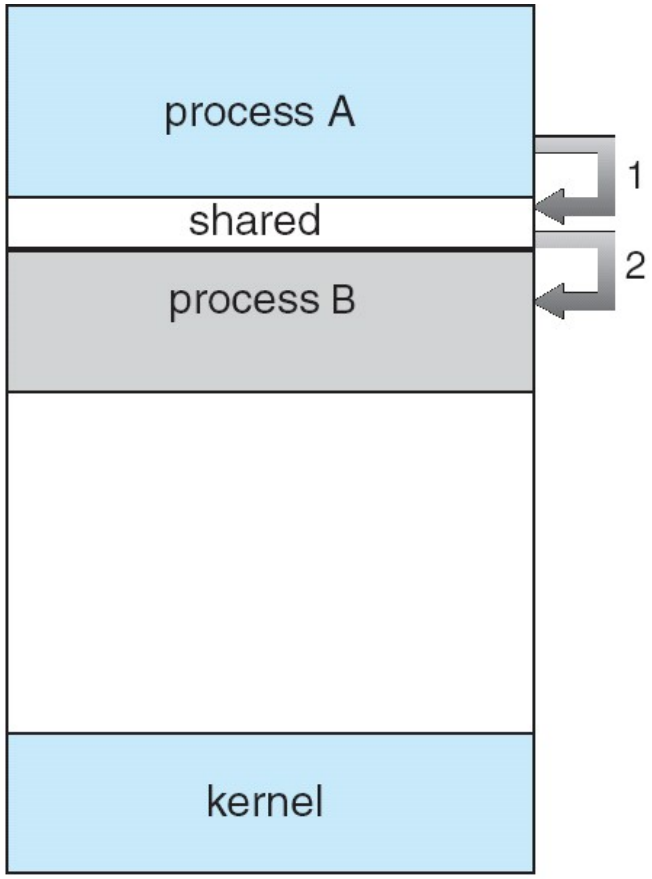
Inter-Process Communication Models – Realization by OS



(a)

IPC via **Message Passing**

Syscalls: send, recv



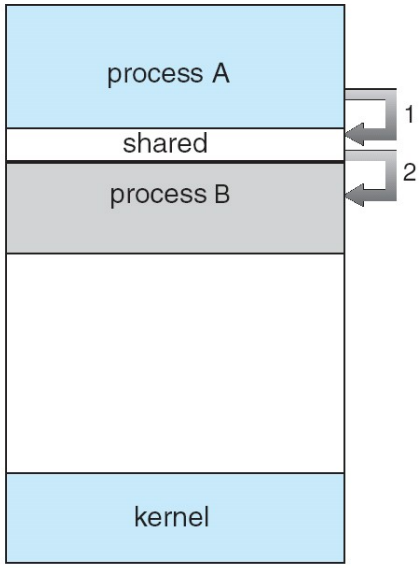
(b)

IPC via **Shared Memory**

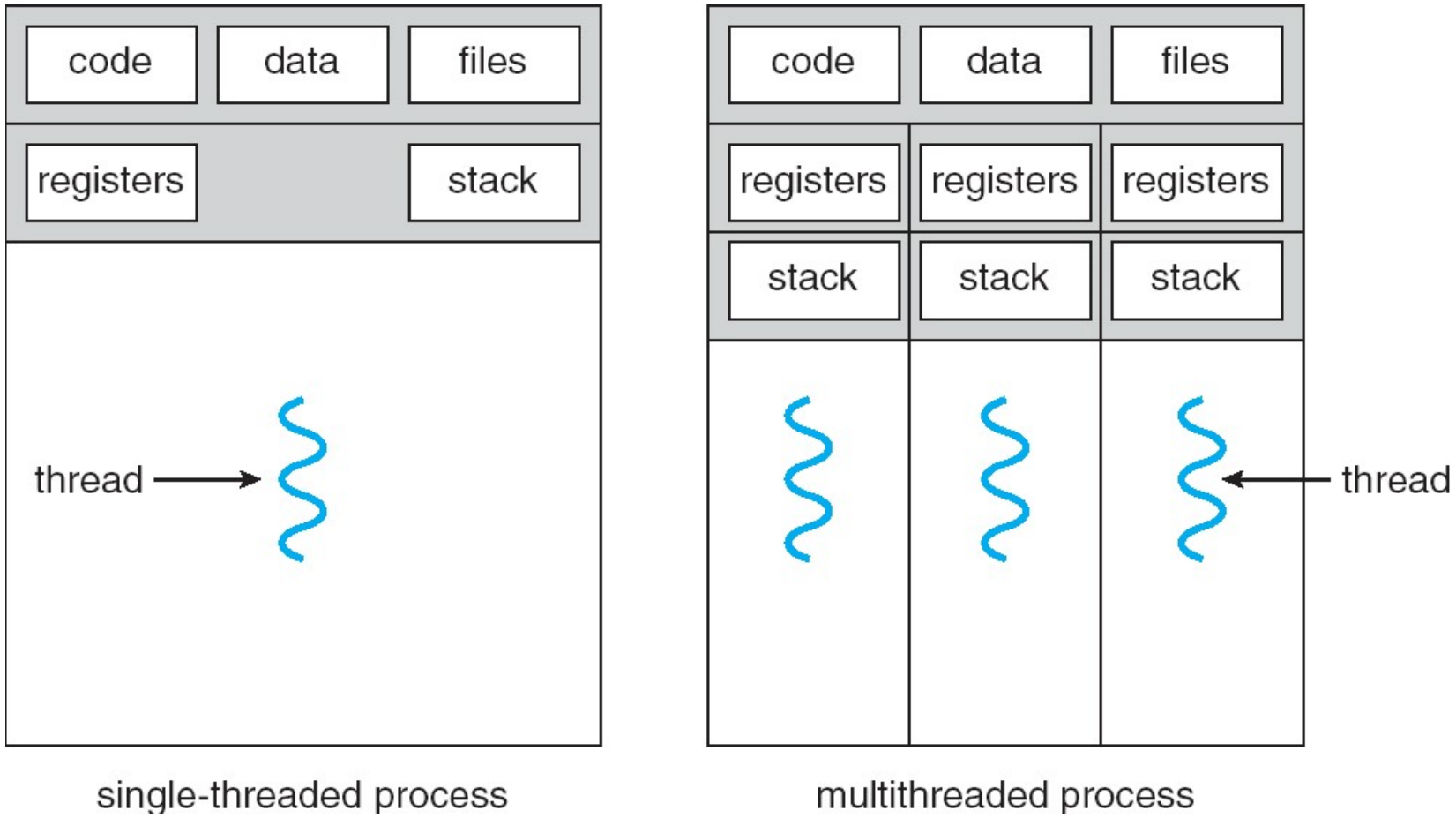
Syscalls: shmget, shmat, then load / store

Example: POSIX Shared Memory API

- #include <sys/shm.h>
  #include <sys/stat.h>
- Let OS create a shared memory segment (system call):
  - int segment_id = **shmget** (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
- Attach the segment to the executing process (system call):
  - void *shmemptr = **shmat** (segment_id, NULL, 0);
- Now access it:
  - strcpy ((char *)shmemptr, ”Hello world”); // Example: copy a string into it
  - …
- Detach it from executing process when no longer accessed:
  - **shmdt** (shmemptr);
- Let OS delete it when no longer used:
  - **shmctl** (segment_id, IPC_RMID, NULL);



Single- and Multithreaded Processes



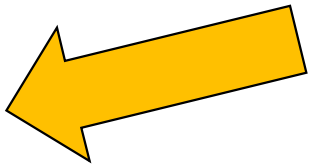
A (software) **thread** is a basic unit of CPU utilization:

- Thread ID, program counter, register set, stack.

A process may have one or several threads.

Benefits of Multithreading

- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
- **Utilization of Multiprocessor Architectures**
  - Convenient (but low-level) shared memory programming



POSIX Threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread programming in C
  - start and terminate threads
  - coordinate threads
  - regulate access to shared data structures
- API specifies behavior, not implementation, of the thread library
- C interface, e.g.
  - **int** pthread_create (pthread_t *thread, **const** pthread_attr_t *attr, **void** *(*start_routine)(**void***), **void** *arg);
- Library, relies on underlying OS and hardware!
- Common in UNIX-based operating systems (Solaris, Linux, Mac OS X)

Starting a Thread (1)

- A new thread is started by calling

```
int pthread_create ( pthread_t *thread,  const  pthread_attr_t *attr,
                     void *(*func)(void*),  void *arg);
```

  - Called func must have 1 parameter and return value of type **void** *
    - Exception: The first thread of the process is started with main()
  - Threads are created and started one by one

- A thread terminates when its called function terminates or by calling

```
pthread_exit ( void *status )
```

- Threads are represented by a data structure of type **pthread_t**

Starting a Thread (2)

- Example:

```
#include <pthread.h>


int main ( int argc, char *argv[] )
{
  int *ptr;
  pthread_t thr;

  pthread_create( &thr,
                  NULL,
                  foo,
                  (void*) ptr  );
  …
  pthread_join( &thr, NULL );
  return 0;
}
```

```
void *foo ( void *vp )
{
   int  i  =  (int) vp;;
   …
}
```

```
// for multiple arguments:
//  pass a pointer to a parameter block:
void *foo ( void *vp )
{
   Userdefinedstructtype *ptr;
   ptr=(Userdefinedstructtype*)vp;
   …
}
```

Access to Shared Data (0)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Caution: typically no protection between thread data – thread1 (foo1) could even write to thread2's (foo2) stack frame

- **Example 0: Parallel incrementing**

```
int a[N];   // shared,  assume P | N
pthread_t  thr[P];

int main( void )
{
   int t;
   for (t=0; t<P; t++)
     pthread_create( &(thr[t]), NULL,
                                incr, a + t*N/P );
   for (t=0; t<P; t++)
     pthread_join( thr[t], NULL );
    …
}
void *incr ( void *myptr_a )
{  int i;  // thread-local variable
   for (i=0; i<N/P; i++)
      ((int*)myptr_a[i])++;    }
```

Access to Shared Data (1)

- Globally defined variables are globally shared and visible to all threads.
- Locally defined variables are visible to the thread executing the function.
- But all data in shared memory publish an address of data: all threads could access…
- Caution: typically no protection between thread data – thread1 (foo1) could even write to thread2‘s (foo2) stack frame

- **Example 1**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;  // thread-local variable
   globalptr = &i;  // ??? dangerous!
     // if foo1 terminates, foo2 writes
     // somewhere, unless globalptr
     // value is reset to NULL manually
   …
}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```

Access to Shared Data (2)

- Globally defined variables are globally shared and visible to all threads
- Locally defined variables are visible to the thread executing the function
- But all data in shared memory publish an address of data: all threads could access…
- Caution: typically no protection between thread data – thread1 could even write to thread2‘s stack frame

- **Example 2**

```
int *globalptr = NULL;    // shared ptr

void *foo1 ( void *ptr1 )
{
   int i = 15;
   globalptr =(int*)malloc(sizeof(int));
   // safe, but possibly memory leak;
   // OK if garbage collection ok



}

void *foo2 ( void *ptr2 )
{
   if (globalptr) *globalptr = 17;
  …
}
```

Coordinating Shared Access (3)

What if several threads need to write a shared variable?

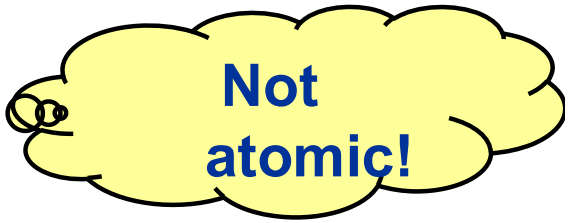
- If they simply write: ok if write order does not play a role
- If they read and write: encapsulate (critical section, monitor) and protect e.g. by mutual exclusion using mutex locks)
- **Example: Access to a taskpool**
  - threads maintain list of tasks to be performed
  - if thread is idle, gets a task and performs it

```
// each thread:
while (! workdone)
{
    task = gettask( Pooldescr );
    performtask ( task );
}

// may be called concurrently:
Tasktype gettask ( Pool p )
{
    // begin critical section
    task = p.queue [ p.index ];
    p.index++;
    // end critical section

    return task;
}
```

Race Conditions lead to Nondeterminism

- Example: p.index++
- could be implemented in machine code as

```
39: register1 = p.index              // load
40: register1 = register1 + 1        // add
41: p.index = register1              // store
```



- Consider this execution interleaving, with “index = 5” initially:

```
39: thread1 executes register1 = p.index          { T1.register1 = 5 }
39: thread2 executes register1 = p.index          { T2.register1 = 5 }
40: thread1 executes register1 = register1 + 1    { T1.register1 = 6 }
40: thread2 executes register1 = register1 + 1    { T2.register1 = 6 }
41: thread1 executes p.index = register1          { p.index = 6 }
41: thread2 executes p.index = register1          { p.index = 6 }
```

- Compare to a different interleaving,
  e.g., 39,40,41, 39,40,41…

  → Result depends on relative speed of the accessing threads
  (race condition)

Critical Section

- **Critical Section**: A set of instructions, operating on shared data or resources, that should be executed by a single thread at a time without interruption
  - **Atomicity** of execution
  - **Mutual exclusion**: At most one process should be allowed to operate inside at any time
  - **Consistency**: inconsistent intermediate states of shared data not visible to other processes outside
- May consist of different program parts for different threads
  - that access the same shared data
- General structure, with structured control flow:

  ...

  Entry of critical section C

  … critical section C: operation on shared data

  Exit of critical section C

  …

Coordinating Shared Access (4)

```
pthread_mutex_t  mutex;    // global variable definition - shared
    …
 // in main:
    pthread_mutex_init( &mutex, NULL );
    …

 // in gettask:
    …
    pthread_mutex_lock( &mutex );
   task = p.queue [p.index];
   p.index++;
   pthread_mutex_unlock( &mutex );
    …
```

Often implemented using test_and_set or other atomic instruction where available

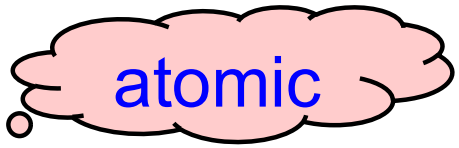
Hardware Support for Synchronization

- Most systems provide hardware support for protecting critical sections
- Uniprocessors – could ***disable interrupts***
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special ***atomic instructions***
  - **TestAndSet**: test memory word and set value atomically
    - Atomic = non-interruptable
    - If multiple TestAndSet instructions are executed *simultaneously* (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.
  - **AtomicSwap**: swap contents of two memory words atomically
  - **CompareAndSwap**
  - **Load-linked / Store-conditional**

TestAndSet Instruction

- Definition in pseudocode:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;      // return the OLD value
}
```



Mutual Exclusion using TestAndSet

- Shared boolean variable lock, initialized to FALSE (= unlocked)
- **do** {

```
    while ( TestAndSet (&lock ))
        ;    // do nothing but spinning on the lock (busy waiting)
```

```
    // …  critical section
```

```
    lock = FALSE;
```

```
    // …    remainder section
```

} **while** (TRUE);

Pitfalls with Semaphores

- Correct use of mutex operations:
  - Protect all possible entries/exits of control flow into/from critical section:

    pthread_mutex_lock (&mutex)
    ….
    pthread_mutex_unlock (&mutex)

- Possible sources of synchronization errors:
  - Omitting **lock**(&mutex) or **unlock**(&mutex) (or both) ??
  - **lock**(&mutex) …. **lock**(&mutex) ??
  - **lock**(&mutex1) …. **unlock**(&mutex2) ??
  - if-statement in critical section, **unlock** in then-branch only

Problems: Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be caused only by one of the waiting threads
  - Typical example: *Nested critical sections*
    - Guarded by locks S and Q, initialized to unlocked

time ↓

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
…	…
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking. A thread may never get the chance to acquire a lock if the mutex mechanism is not *fair*.

Deadlock Characterization [Coffman *et al.* 1971]

Deadlock can arise only if **four conditions** hold simultaneously:

- **Mutual exclusion**: only one thread at a time can use a resource.
- **Hold and wait**: a thread holding at least one resource is waiting to acquire additional resources held by other threads.
- **No preemption of resources**: a resource can be released only voluntarily by the thread holding it, *after* that thread has completed its task.
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting threads such that
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for a resource that is held by P_2, …,
  - P_{n-1} is waiting for a resource that is held by P_n, and
  - P_n is waiting for a resource that is held by P_0.

Coordinating Shared Access (5)

- Must also rely on implementation for efficiency
- Time to lock / unlock mutex or synchronize threads varies widely between different platforms
- A mutex that all threads access serializes the threads!
  - Convoying
  - Goal: Make critical section as short as possible

```
// in gettask():
int tmpindex;  // local (thread-private) variable
pthread_mutex_lock( &mutex );
tmpindex = p.index++;
pthread_mutex_unlock( &mutex );
task = p.queue [ tmpindex ];
```

Possibly slow shared memory access now outside critical section

Textbook references on thread programming (Selection)

- C. Lin, L. Snyder: *Principles of Parallel Programming*. Addison Wesley, 2008. (general introduction; **pthreads**)
- B. Wilkinson, M. Allen: *Parallel Programming, 2e*. Prentice Hall, 2005. (general introduction; **pthreads**, OpenMP)
- M. Herlihy, N. Shavit: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. (**threads**; nonblocking synchronization)