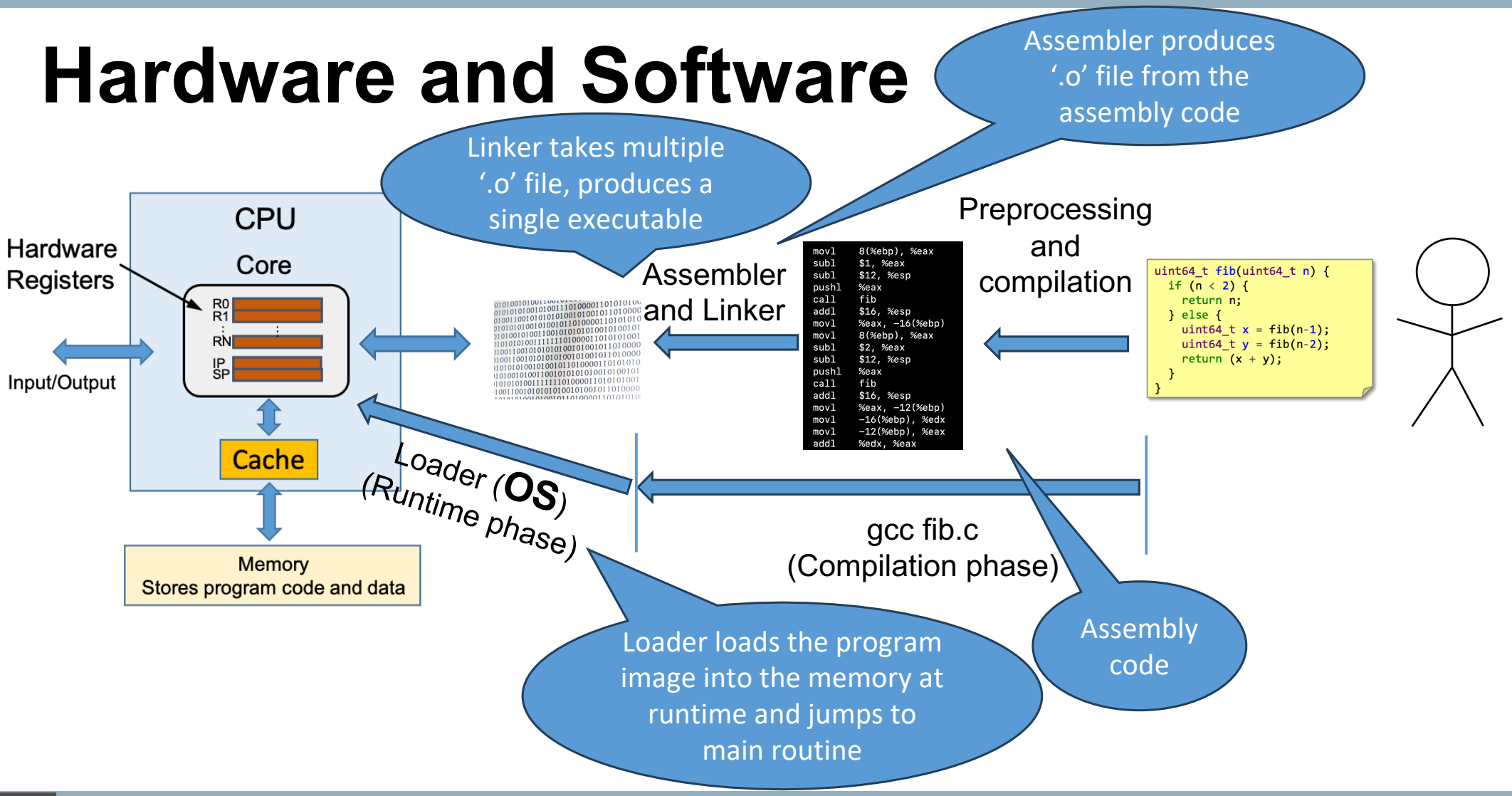
Lecture 14: Mid-Semester Review

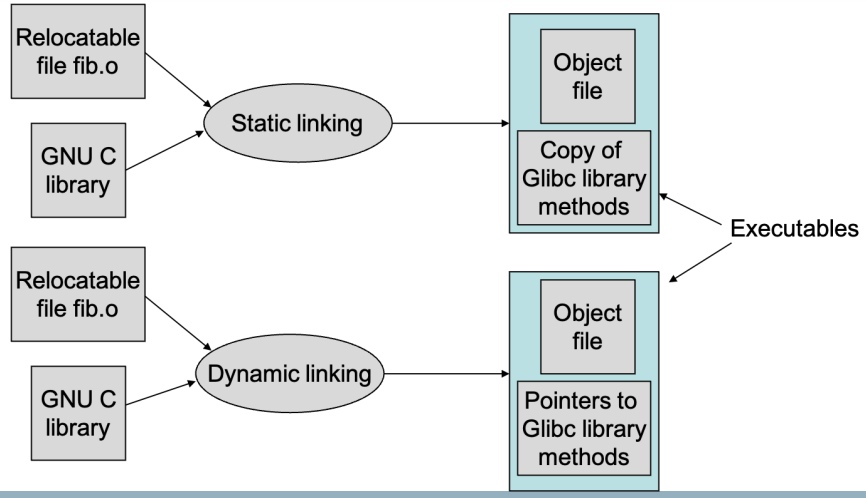
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Hardware and Software



Static and Dynamic Linking

```
$ gcc -static -o fib fib.c
$ file fib
...ELF 64-bit executable...statically linked
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 845304 Aug  6 10:26 fib
$ gcc -o fib fib.c
$ ls -l fib
-rwxrwxr-x 1 vivek vivek 8328 Aug  6 10:27 fib
```



- Static linking
  - Each and every library modules referenced in the relocatable file is copied into the final executable
    - Static binding at compile time
- Dynamic linking
  - Final executable only contains references (pointers) to the library method instead of the copy of a library method
    - Binding with library done at runtime during execution

Executable and Linkable Format (ELF)

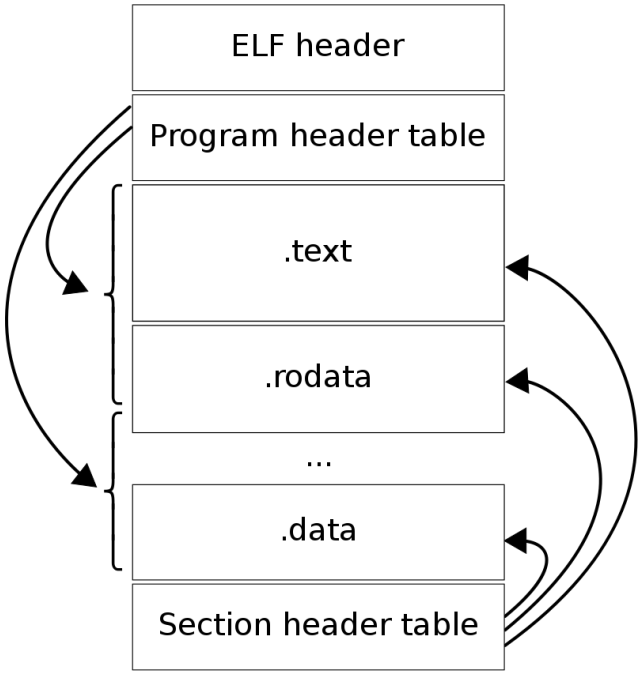


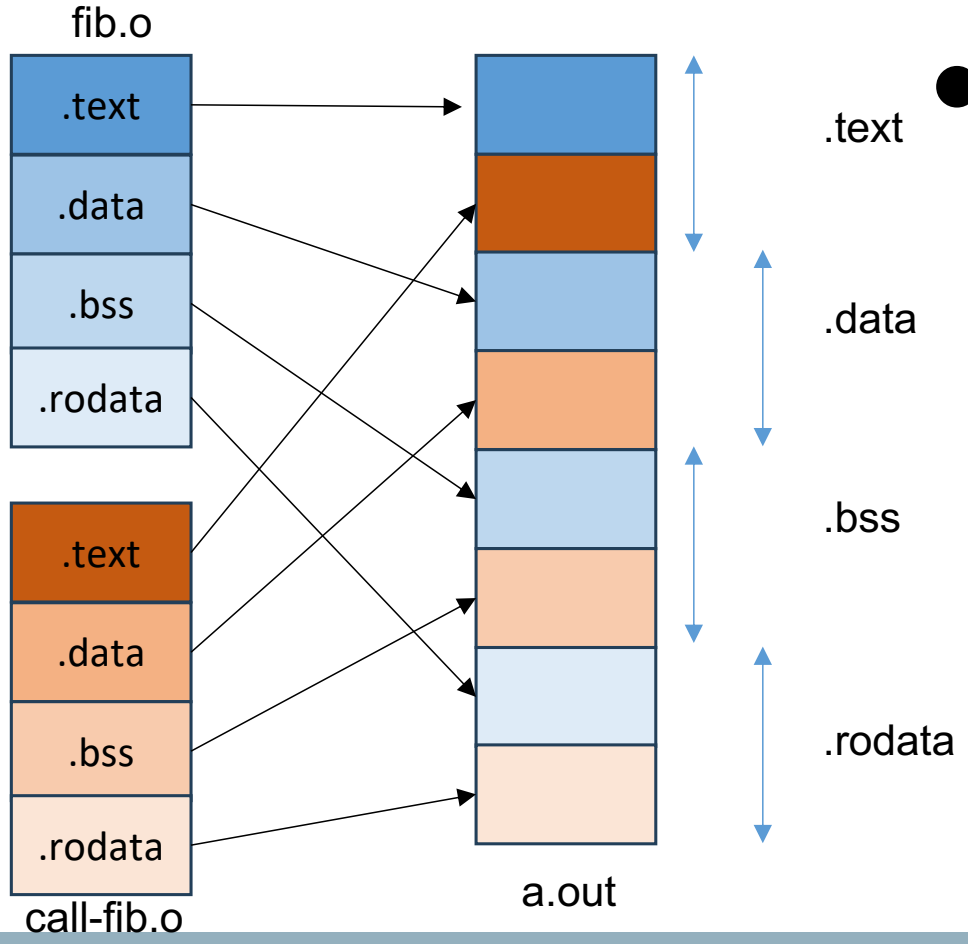
Image source: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

- ELF header
  - Provides a roadmap for the entire file organization
    - Always at offset zero of the object file
    - Provides entry point address for execution
- Two views of an ELF file
  - Linkable view (relocatable file)
  - Execution view (executable file, shared object)
- Linkable view
  - Section header table
    - One section header for each section
  - Sections
    - Contains data required for linking
    - Machine code, global variables, (initialized and un-initialized), symbol tables, line mapping between machine code and original C code, etc.
- Execution (or Loader) view
  - Program header table
    - One program header for each segment
  - Segments
    - Created by merging several sections
    - Contains information required for by the loader for execution
- Contiguous chunk of memory (ELF header, PHT, SHT, each section, each segment)

Linker: Merges the Sections

```
[vivek@possum]$ cat fib.c
int get_number();

int fib(int n) {
  if(n<2) return n;
  else return fib(n-1)+fib(n-2);
}

int compute() {
  int n = get_number();
  return fib(n);
}
```

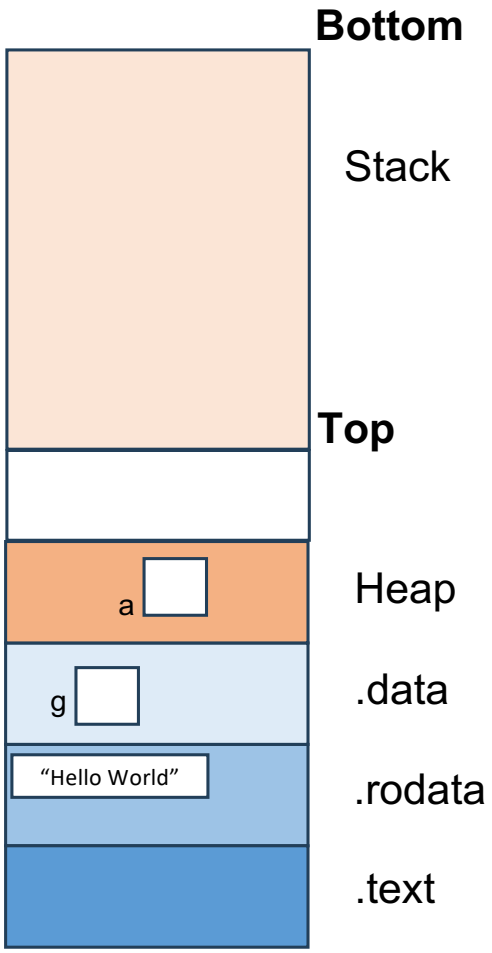
```
[vivek@possum]$ cat call-fib.c
int compute();
int get_number() {
  return 40;
}
int main() {
        int result = compute();
        return 0;
}
```



- Relocation is the process of merging the sections and resolving the symbol addresses

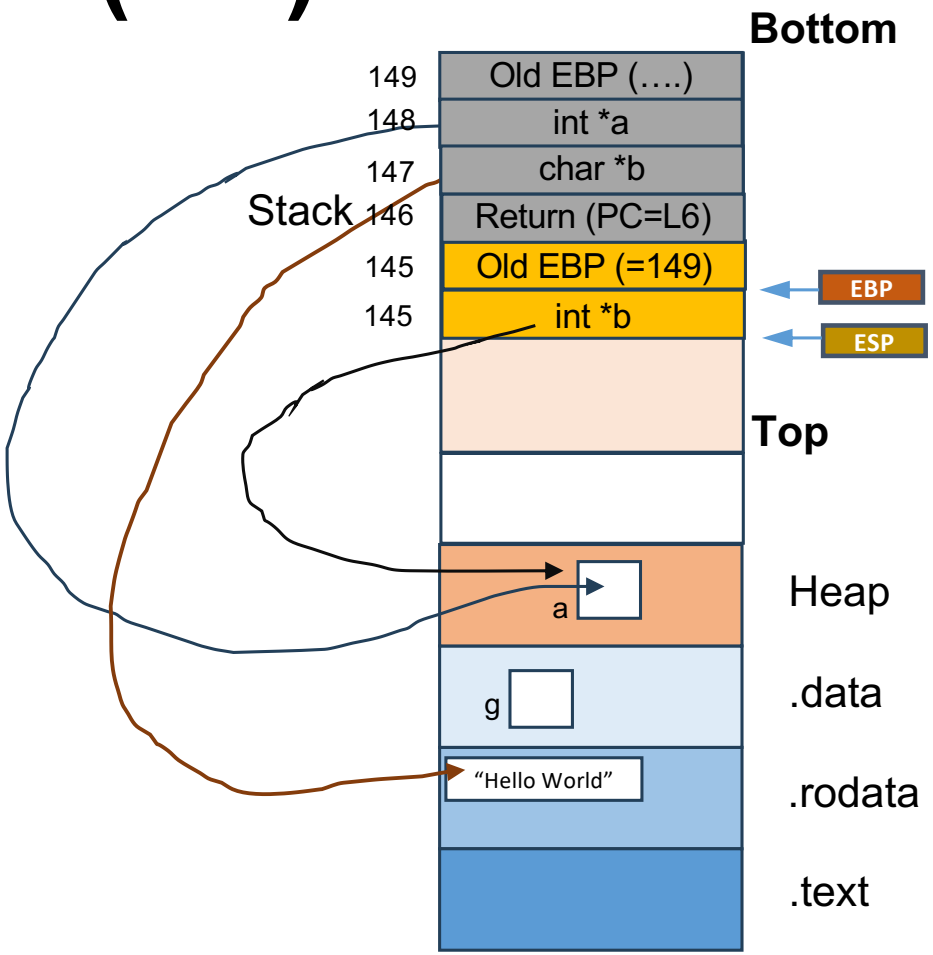
Program Execution (1/3)

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```



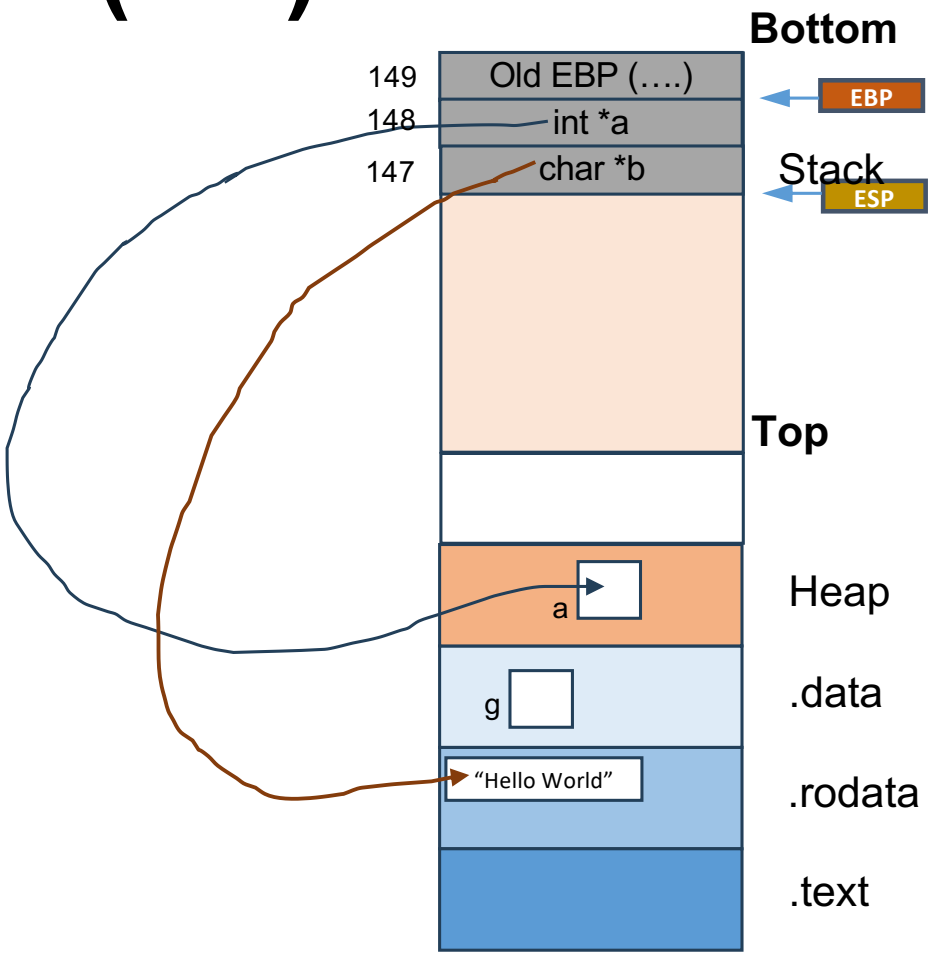
Program Execution (2/3)

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```

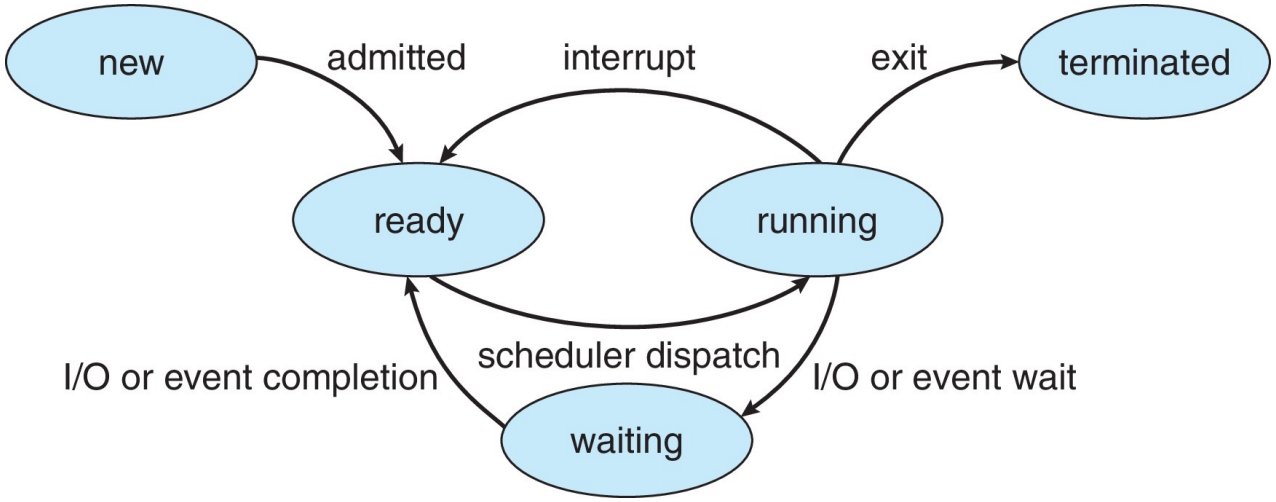


Program Execution (3/3)

```
L1: int g=0;

L2: void main() {
L3:     int *a = (int*) malloc(4);
L4:     char *b = “Hello World”;
L5:     foo(a);
L6:     g=*a;
L7: }

L8: void foo(int* b) {
L9:     *b = 20;
L10:}
```

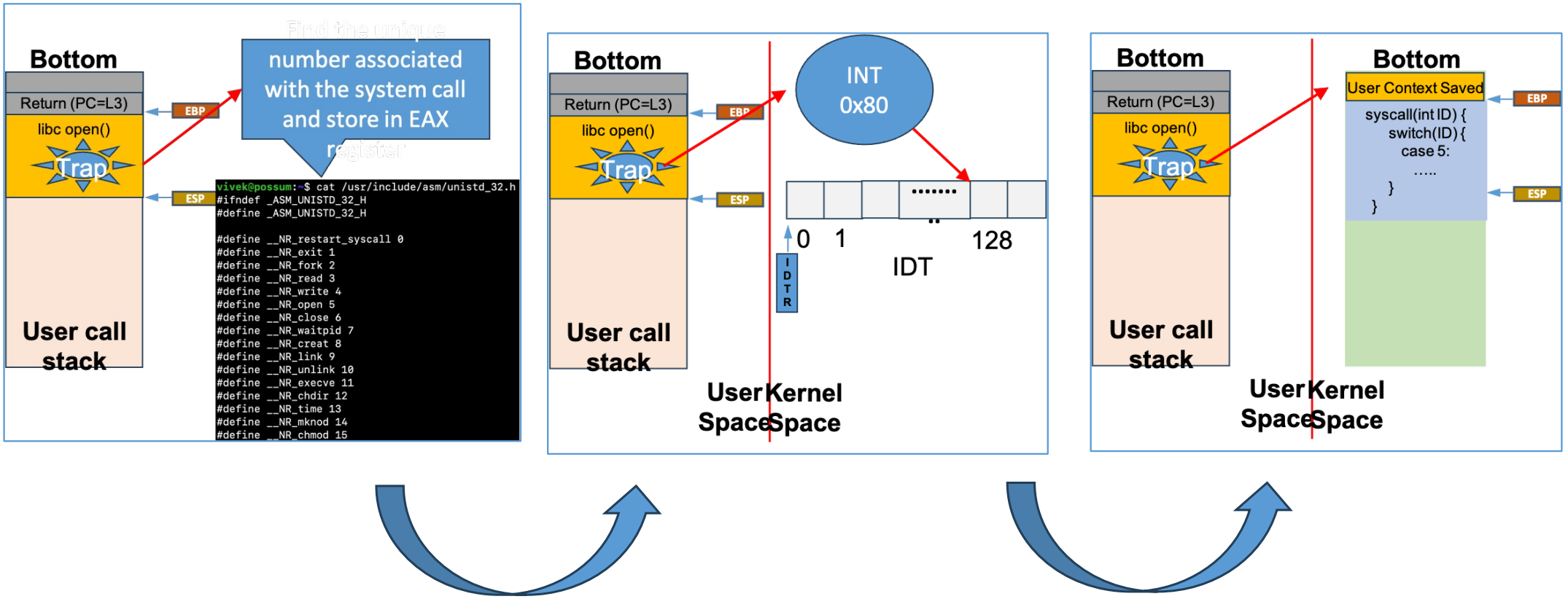


Process States



- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

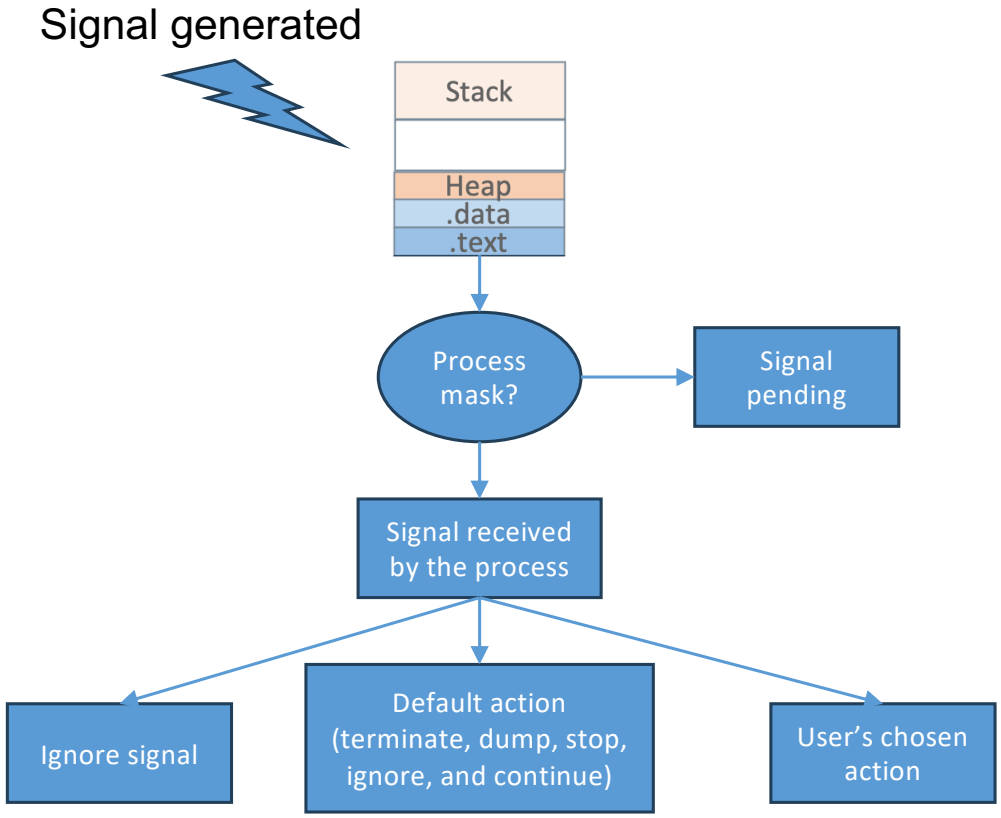
Steps for Making a System Call



Process Creation & Termination

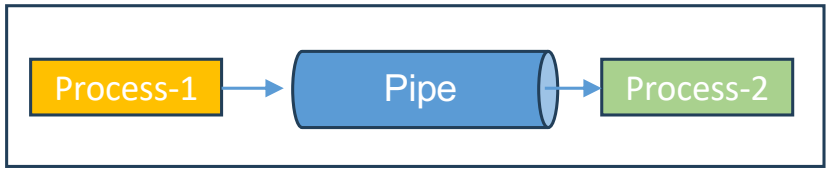
```
int global=0;
int main() {
    if(fork() == 0) {
        global++;
        printf("Child process: Global=%d\n", global);
        char* args[2] = {"./fib", "40"};
        execv(args[0], args);
        printf("I should never print\n");
    } else {
        printf("I am the Parent process\n");
        int status;
        wait(&status);
    }
    printf("Global=%d\n", global);
    return 0;
}
```

Signal Delivery and Handling



- Process can use sigprocmask function to make changes to its signal mask (member in task_struct) if it wants to temporarily block a set of signals (not all signals can be blocked, e.g., SIGKILL and SIGSTOP)
- User can create his own signal handler function
  - Except SIGKILL and SIGSTOP

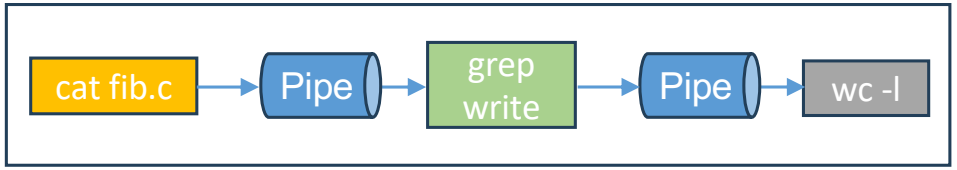
IPC Using Pipes



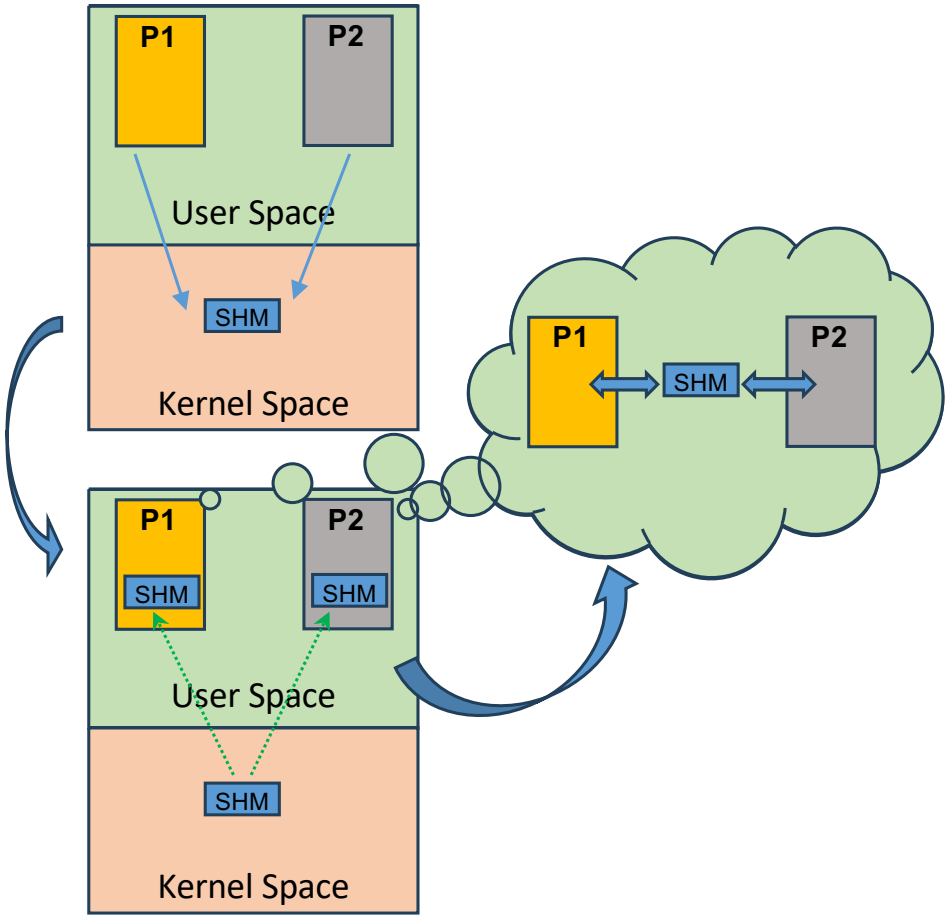
Processes **must** be running on the **SAME** machine

- Pipes (analogous to water pipe) is a unidirectional stream of data flowing from a source process to a destination process
  - Kernel buffer exposed to processes as a pair of file descriptors (readable end and writable end). Default buffer size on latest kernel is 16 pages (16x4096 bytes)
  - Data delivered in the same order as sent
  - Uses blocking IO and the writer process will block if the pipe is full
- We use it frequently on Shell
  - Better than using temporary files as pipes automatically clean up themselves unlike files that must be explicitly removed using the “rm” command on Shell

```
vivek@possum:~/os23$ cat fib.c | wc -c
1788
vivek@possum:~/os23$ cat fib.c | wc -l
77
vivek@possum:~/os23$ cat fib.c | grep write | wc -l
10
```



IPC Using POSIX Shared Memory



- Shared memory is the fastest form of IPC in which processes can asynchronously write to a shared region of memory **without** the need for switching to kernel space
  - User specified memory size
- Access to the shared memory is achieved as follows
  1. Create and open a new shared memory object using shm_open system call in the parent process
  2. Set the desired size for the newly created shared memory region using ftruncate
  3. Map the shared memory region into the process's address space using mmap (parent process calls mmap that gets shared in child's address space after fork)
  4. Both child and the parent processes un-maps the shared memory from their address space upon completion by using munmap system call, followed by closing the file descriptor for the shared memory object (close system call)
  5. Parent deletes the shared memory object by using shm_unlink

Producer Consumer Problem

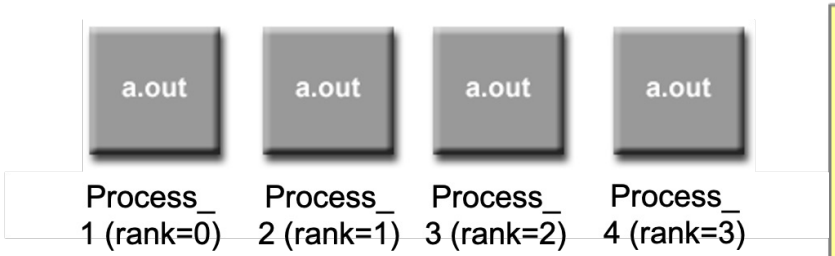
```
typedef struct cookiejar_t {
    int cookie;
    sem_t jar_empty;
    sem_t jar_full;
} cookiejar_t;

cookiejar_t* cookiejar;

int main() {
    cookiejar = setup();
    cookiejar->empty=1;
    sem_init(&cookiejar->jar_empty, 1, 1);
    sem_init(&cookiejar->jar_full, 1, 0);
    if(fork() == 0) { homer(); }
    if(fork() == 0) { marge(); }
    wait(NULL); // wait for Homer process
    wait(NULL); // wait for Marge process
    sem_destroy(&cookiejar->jar_empty);
    sem_destroy(&cookiejar->jar_full);
    cleanup();
    return 0;
}
```

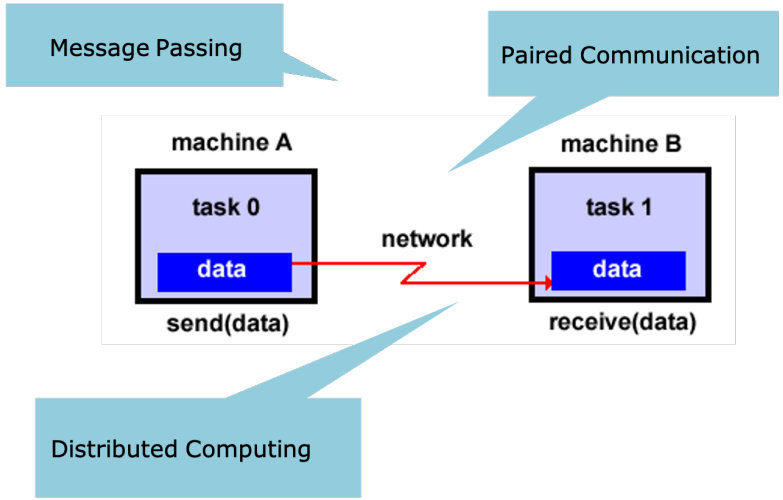
```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

Homer Process (CPU 0)		Sem (empty)	Sem (full)	Marge Process (CPU 1)	
Action	**Process State**	**Sem Value**	**Sem Value**	**Action**	**Process State**
Forked	Ready	1	0	Forked	Ready
wait(full)	Blocked	0	-1	wait(empty)	Running
	Blocked	0	-1	Bake Cookie-1	Running
	Ready	0	0	post(full)	Running
Eat Cookie-1	Running	-1	0	wait(empty)	Blocked
post(empty)	Running	0	0		Ready
wait(full)	Blocked	0	-1	Bake Cookie-2	Running
	Ready	0	0	post(full)	Running
…….				…….	

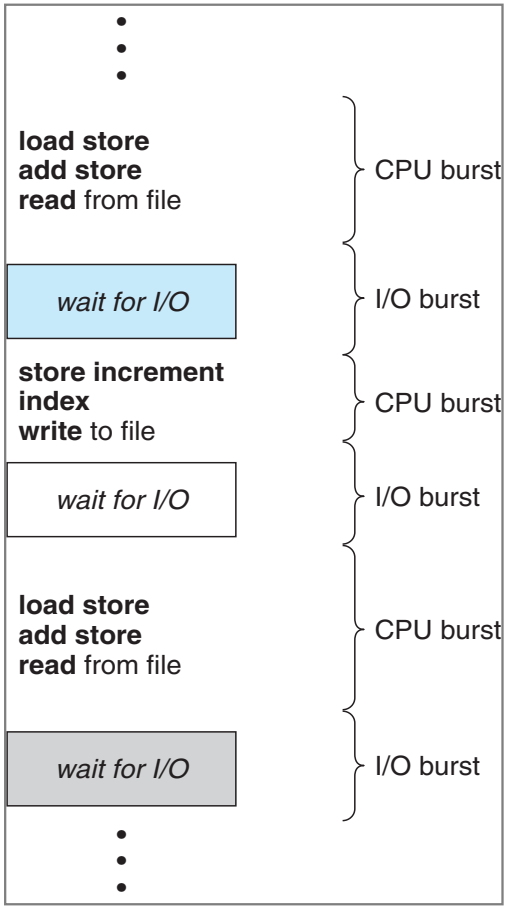
IPC in Distributed Memory






```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

Process Scheduling



Property	FCFS	SJF	STCF	RR
Optimize Average **Completion** Time	✗	✓	✓	✗
Optimize Average **Waiting** Time	✗	✗	✗	✓
Prevent Starvation	✓	✗	✗	✓
Prevent Convoy Effect	✗	✗	✓	✓
Psychic Skills **NOT** Needed	✓	✗	✗	✓

If jobs arrive simultaneously

Always, as it schedules jobs in order of estimated completion time

Doesn't favor any fixed property

Preemptible

Any scheduling policy that always favors a fixed property for scheduling leads to starvation

Where are we as of now

- **CSE231 Post Conditions**
  1. Students are able to create a Unix shell with complete clarity about process creation and process execution
  2. Students are able to write multi- threaded applications with synchronization primitives and ability to analyze effects of concurrency on process execution and correctness
  3. Students are able to analyze the impact of OS concepts, e.g. virtual memory, concurrency, on program execution and ability to fine-tune the program to run efficiently on a given OS
  4. Students are able to demonstrate deeper understanding of the Unix-like OSes and kernel programming

All the best for your exam !!

- Exam syllabus
  - Lectures **02-13**
- Exam schedule
  - Date: Thursday (6th Oct.)
  - Time: 3:00pm-4:00pm (One hour only)
  - Venue: C102
- Assignment-3 will be released after the mid semester exams (14th October)