Operating Systems Background

Overview

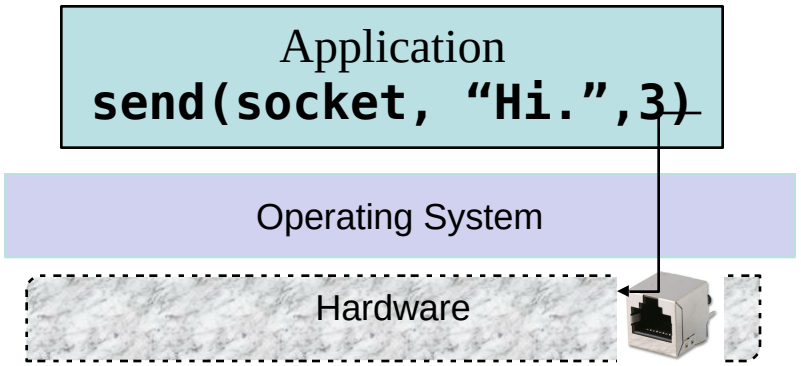
- Last time: Clouds are efficient because of resource sharing and multiplexing
- Multiple applications share computing resources
- This lecture: building block of safe multiplexing
  - Operating Systems
- Also useful in writing programs

Operating Systems

- Operating Systems: Easier to run applications
- OS provides a convenient interface to run multiple programs in a secure manner
- Portability: Decouple applications from hardware
  - Changing your USB keyboard => No need to rewrite and recompile programs
- Resource allocation and multiplexing
- OS provides all these features by:
  - Different abstractions & services
  - Interfacing with hardware features designed to help OS

OS Services

- Programs: Sequence of CPU instructions
  - Mov, add, jmp,…
- Programs often build on top of and make use of other programs (“libraries”)
- OS provides a wide range of services to applications



Operating System Services

- User interface - Almost all operating systems have a user interface (UI)
  - Command-Line (CLI), Graphical User Interface (GUI), Batch
- Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- I/O operations - A running program may require I/O, which may involve a file or an I/O device.
- File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont.)

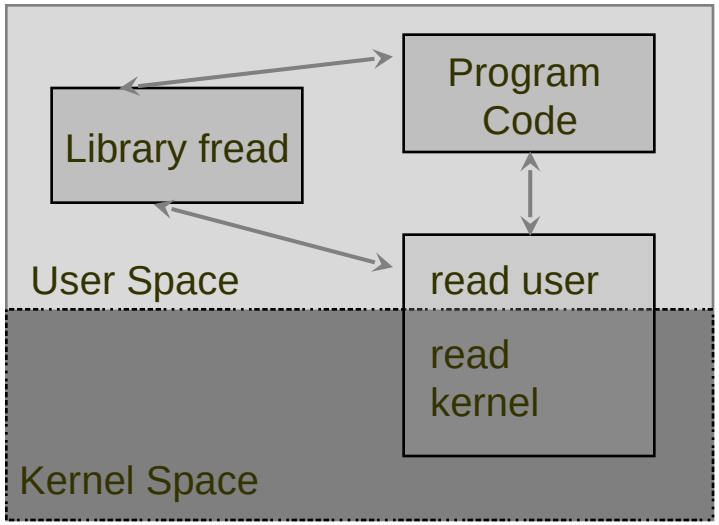
- Communications – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

System Calls

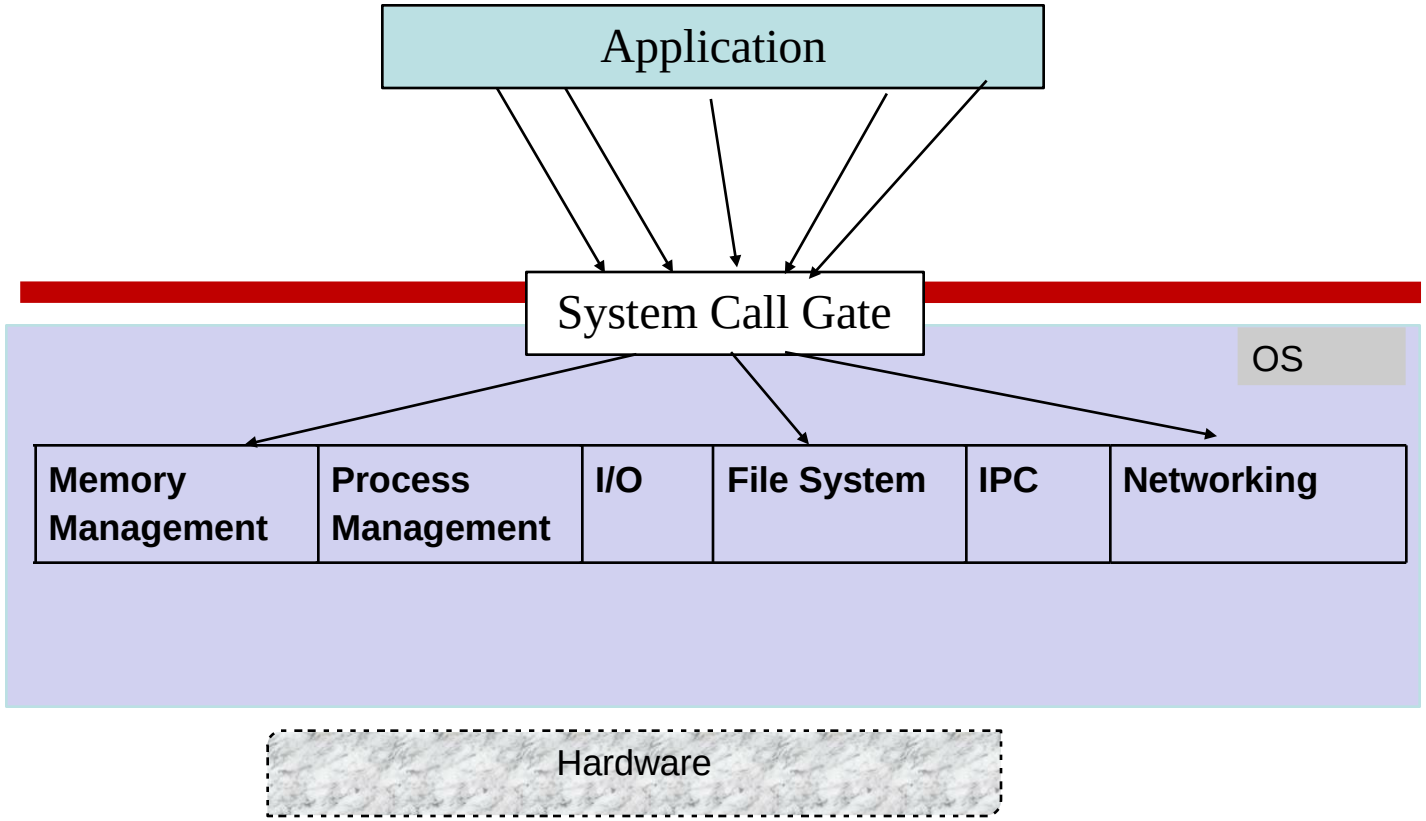
- Applications access OS services by making *system calls*
  - A function call that invokes the kernel
- This is the view of what the OS is and does from the application perspective



System Calls

- Programming interface to OS services
- Small, well-defined set of function calls into the OS kernel
  - Applications not allowed to call arbitrary kernel functions
  - Syscall implementation can change over time, but the semantics and API remains the same
    - Linux fork() implementation optimized by > 10x, but same 40 year old semantics.
- System calls are NOT typical library API calls
  - Privilege separation between OS and applications
  - Syscalls involve a user --> kernel “mode switch” for the CPU

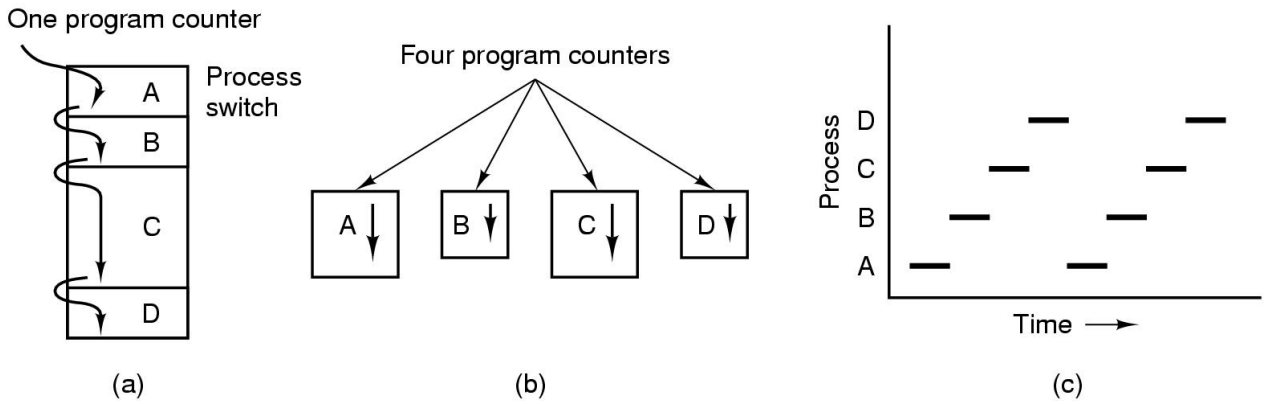
API – System Call – OS Relationship



Programs and processes

- A program is a series of instructions
  - code for a single “process” of control
- Process: running program + state
- State: Input, output, memory, code, file, etc.
- A Thread is an execution context with register state, a program counter (PC) and a stack
  - “Thread of execution”
- Multiple processes can be running the same program, even sharing the code in the same memory space
  - reduces memory overhead, which is important in limited memory environments like embedded OSes

The process abstraction



- Multiprogramming of four programs in the same address space
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Concurrency

- Multiple processes can run “simultaneously”
- Number of processes >> Number of CPUs
  - How?
- Time-sharing: Run processes briefly
- Periodically, the OS ‘context-switches’ to a different process
  - OS saves process state (CPU registers etc)
- Each process under the illusion that it has full access to the CPU

CPU Virtualization

- Processes create the illusion of multiple “virtual” CPUs that programs fully control
- Process PCB contains program counter and other register state, allowing it to be “resumed”
- Timesharing: OS switches process running on physical CPU at high frequency (context switch)
- Virtualization is a key OS principle
  - Applies to CPU, memory, I/O, …

Concurrency and Parallelism

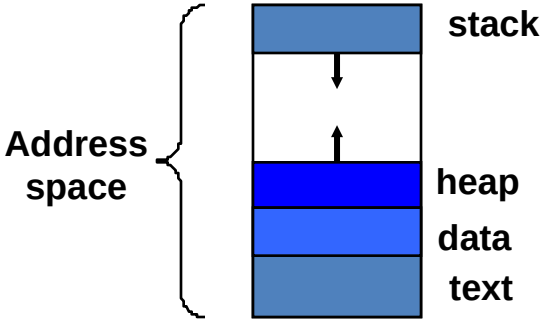
- Concurrency: Independent execution of multiple processes
- Ability to deal with multiple things at a time
- Parallelism: Actually doing things simultaneously on different hardware

Process Control Block

- OS stores all process state and “meta” data
- Process Id
- Process State : Running, Suspended, etc.
- CPU State: Program counter, registers
- Memory/Address space information
- Accounting Info: cycles running, sleeping
- IO: Open files, sockets, etc
- Scheduling class, priority
- Linux **task_struct**
  https://github.com/torvalds/linux/blob/master/include/linux/sched.h

UNIX Process Address Space

- Memory locations process is allowed to address
- Each process runs in its own virtual memory *address space* that consists of:
  - *Stack space* – used for function and system calls
  - *Data space* – static variables, initialized globals
  - *Heap space* – dynamically allocated variables
  - *Text* – the program code (usually read only)



- Invoking the same program multiple times results in the creation of multiple distinct address spaces

UNIX Process Creation

- Parent processes create child processes, which, in turn create other processes, forming a tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

UNIX Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process hierarchies

- Parent creates a child process,
  - System calls for communicating with and waiting for child processes
  - Each process is assigned a unique identifying number or process ID (PID)
- Child processes can create their own child processes
  - Forms a hierarchy
  - UNIX calls this a "process group"
  - Windows has no concept of process hierarchy
    - all processes are created equal

Process creation in UNIX

- All processes have a unique process id
  - *getpid(), getppid()* system calls allow processes to get their information
- Process creation
  - *fork()* system call creates a copy of a process and returns in both processes, but with a different return value
  - *exec()* replaces an address space with a new program
- Process termination, signaling
  - *signal(), kill()* system calls allow a process to be terminated or have specific signals sent to it

Example: process creation in UNIX

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

sh (pid = 24)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

sh (pid = 24)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

sh (pid = 24)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Process creation in UNIX example

sh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

ls (pid = 24)

```
//ls program
main(){
  //look up dir
  …
}
```

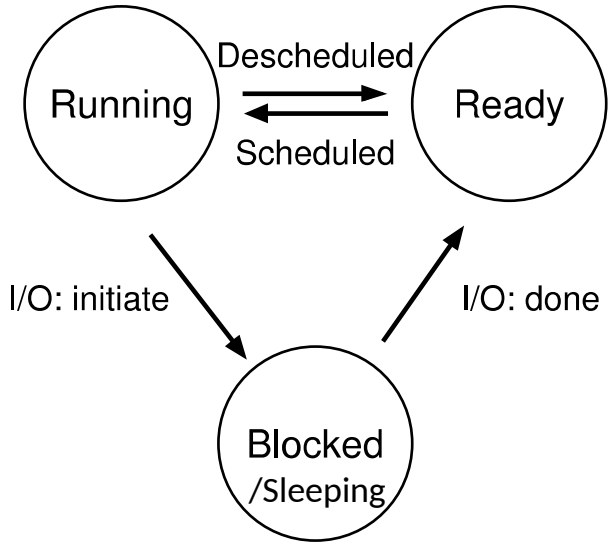
C Program Forking Separate Process

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

Process Fork In Linux

- https://elixir.bootlin.com/linux/latest/source/kernel/fork.c#L1604
- Address space marked copy on write, for impending exec
- PCB copied (dup_task_struct)
- New address space created (new page tables)

Process States



Files

- Files: Sequence of bytes
- Great UNIX Idea: (Almost) Everything is a file descriptor
  - Files on disk
  - I/O devices such as keyboards, consoles, (cat /dev/tty)
  - Network sockets
  - Pipes
  - Pseudo file systems to interact with OS (procfs, sysfs)
- Simple, yet powerful OS abstraction and service
- Same open, read, write, close operations

UNIX **read** syscall

- Reading from a file on disk into an in-memory buffer using **read**
- Unix system calls are described in the manual (man) pages
- **man 2 read , man 2 open, …**
- Reading a file:

```
int file_flags = O_RDONLY ; //defined in fcntl.h

int file_desc = open("/home/foo.txt", file_flags) ;

void* buffer = malloc(2048); //2KB buffer

ssize_t num_read = read(file_desc, buf, 200); //read only 200 bytes

//do something with data in buf

close(fd) ;
```

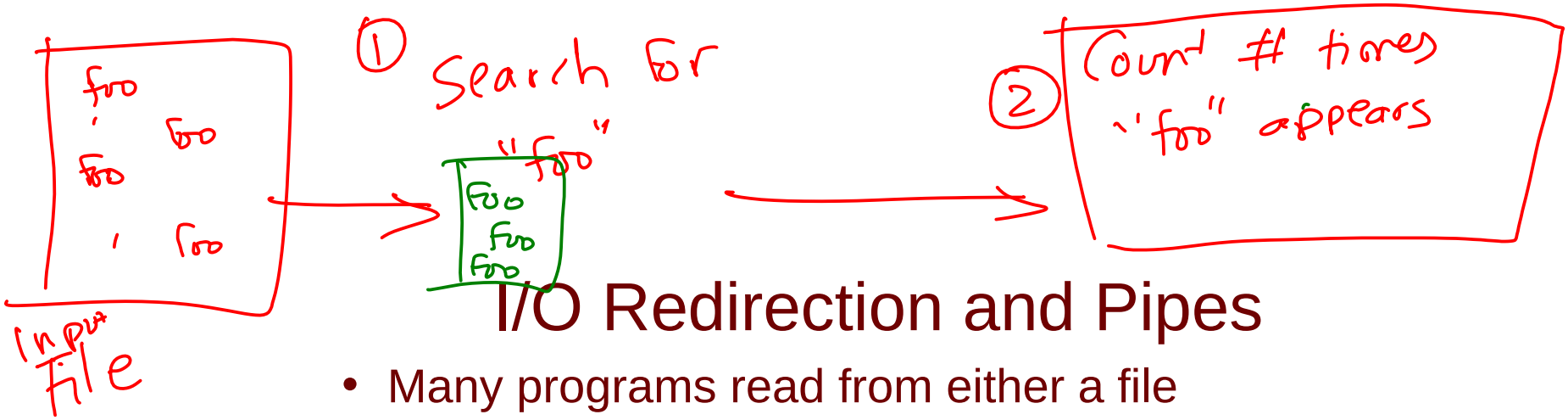
- **Note: Real programs must incorporate error handling.**
  - **What if file doesn't exist? What if we didn't read 200 bytes?**

Default File Descriptors

- By convention, Unix processes associate certain file descriptors with roles
- 0 - STDIN_FILENO (or stdin)
- 1 - STDOUT_FILENO (or stdout)
- 2 - STDERR_FILENO
- Just convention (not a feature of the kernel) but many things would break if it weren’t followed

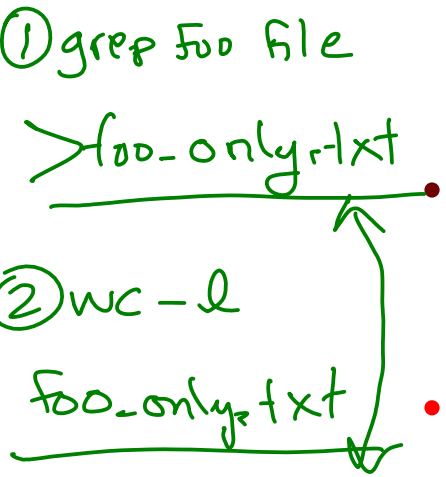
I/O Redirection

- The shell has mechanisms to control the initial associations of these descriptors
- **<** -- attach stdin to a file
  - Process reading from stdin will read from the file
  - Can be anywhere in the input
  - **`wc < /dev/stdin`**
- **>** -- attach stdout to a file
  - If it does not exist, it is created (with permission)
- **>>** -- attach stdout to a file and append all writes to end of the file
  - Just like **>** if the file doesn't exist



I/O Redirection and Pipes

- Many programs read from either a file specified as an argument or stdin
  - Again, only a convention
  - Thus "wc file" == "wc < file" == "cat file | wc"
- You can connect the stdout of one command to the stdin of another with the symbol **|**
  - Called a pipe
- **Pass the output file from one program as input to another.**
- **Pipes alleviate need for temporary files**
  - **grep foo file > temp ; wc –l temp**
  - **cat file | grep foo | wc –l**



pipes

Most unix programs exped stdin as their default argument/input

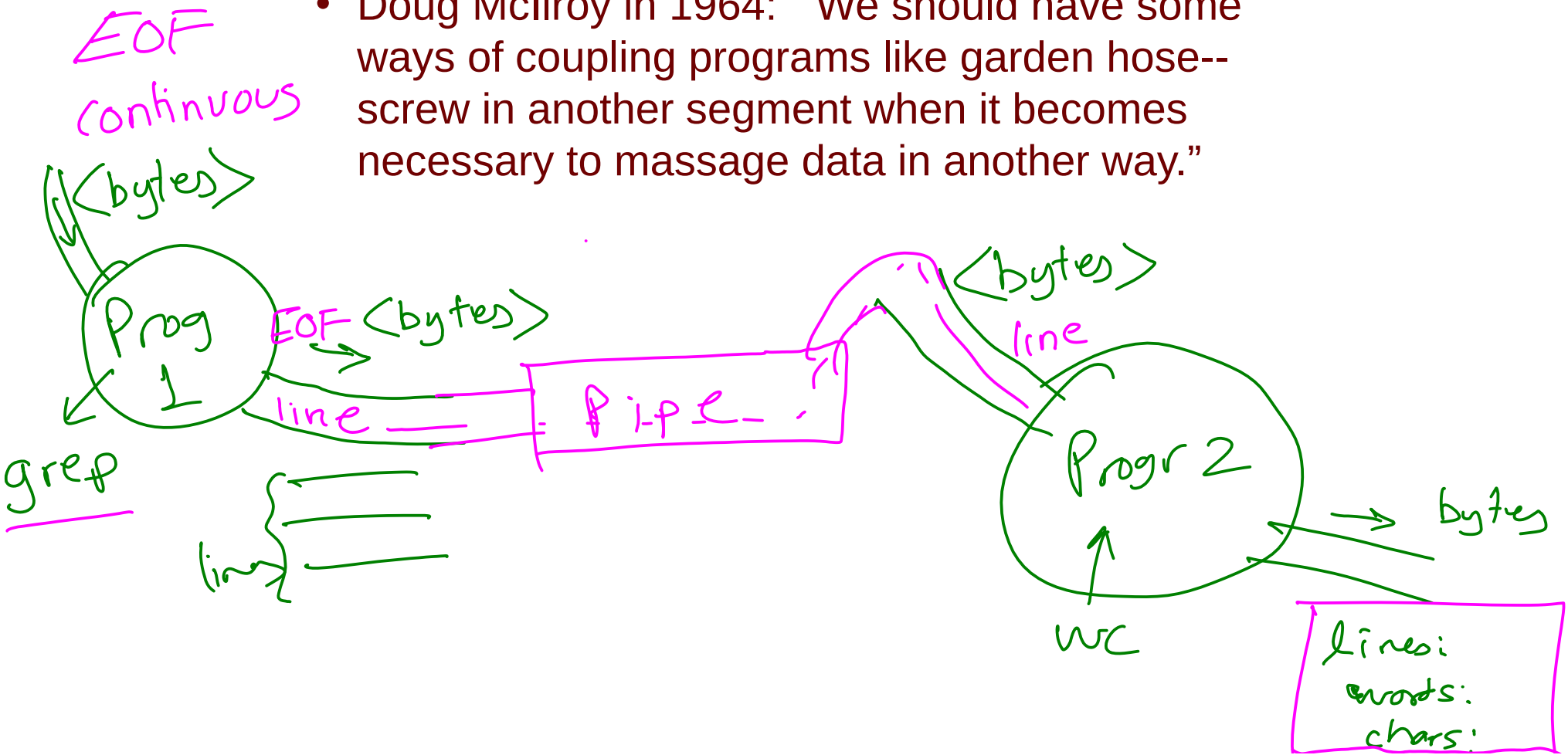
I/O Redirection

- You can send two file descriptors to one
  - In *sh **2>&1** will redirect stderr to stdout
  - **command1 2>&1 | command2**
  - In *csh, you can send both to a file with >& and to another process with |&
- cat < file | sort > output.txt

Pipes

seq of bytes

- Combination of “Everything a file” + pipes is a powerful “service” provided by UNIX
- Doug McIlroy in 1964: “ We should have some ways of coupling programs like garden hose--screw in another segment when it becomes necessary to massage data in another way.”

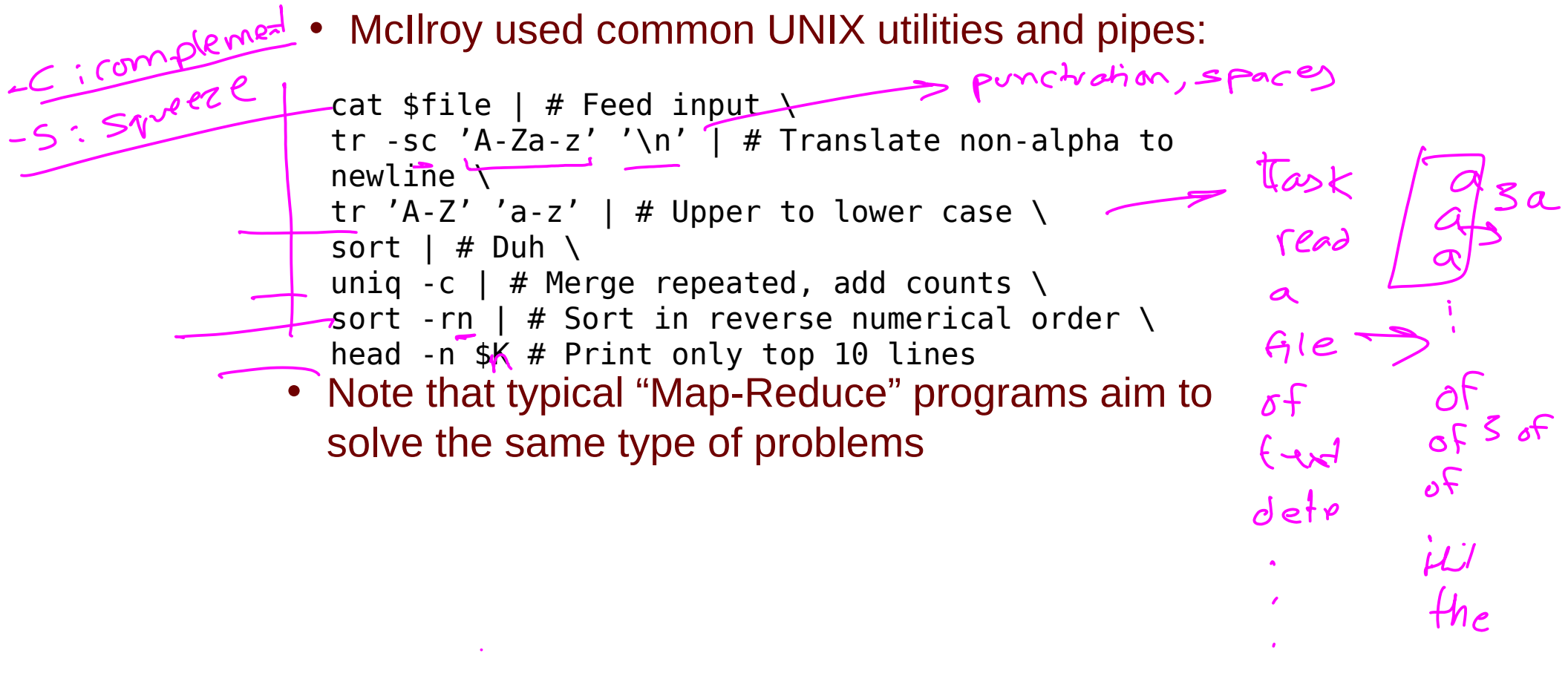


Knuth vs. McIlroy

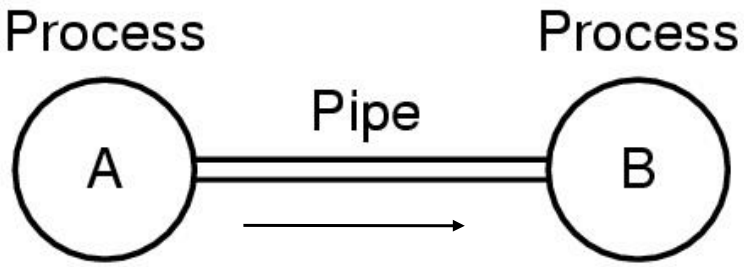
- Task: Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.
- Knuth: 8 pages program
- McIlroy used common UNIX utilities and pipes:

```
cat $file | # Feed input \
tr -sc 'A-Za-z' '\n' | # Translate non-alpha to
newline \
tr 'A-Z' 'a-z' | # Upper to lower case \
sort | # Duh \
uniq -c | # Merge repeated, add counts \
sort -rn | # Sort in reverse numerical order \
head -n $K # Print only top 10 lines
```

- Note that typical "Map-Reduce" programs aim to solve the same type of problems



Pipe System Call



man -s 2 pipe or man 2 pipe

int
pipe(int filedes[2]);

The **pipe**() function creates a pipe, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors.

filedes[1] is the write end, filedes[0] is the read end

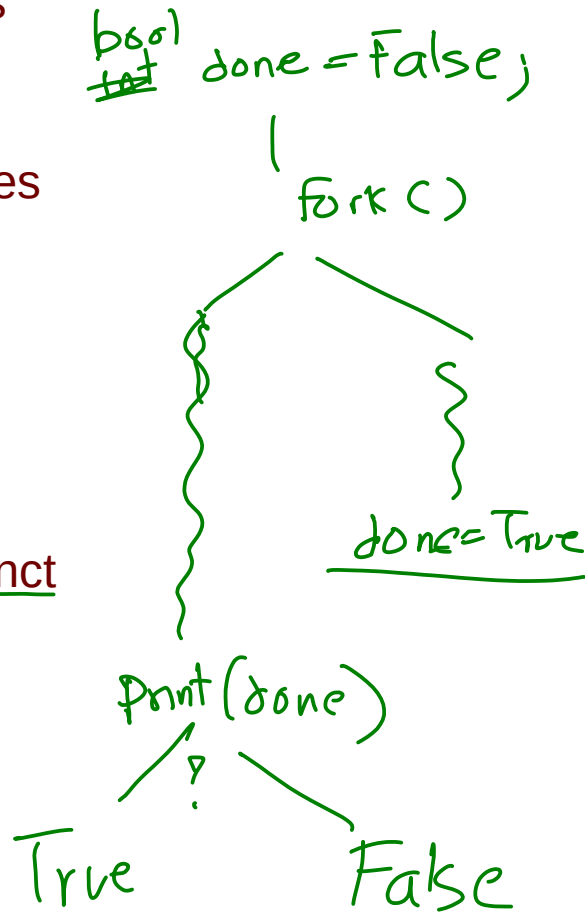
Processes In Python

- Python’s subprocess module
- subprocess.run([“ls”, “-l”]) (new in 3.5)
- subprocess.call(), check_call()
- subprocess.Popen([prog, args], stdin=, stdout=)

UNIX Threads

- Creation of a process using `fork()` is expensive (time and machine effort)
  - Memory copying to create a copy of the process
    - In many cases just to call `exec()` and replace it
    - There are ways to mitigate creating a complete copy
  - Coordinating activities across process boundaries requires effort
- Threads are sometimes called *lightweight processes*
  - What we have called a process is sometimes considered a *heavyweight* process
  - A thread contains the necessary state for a distinct activity (process in the most general sense)

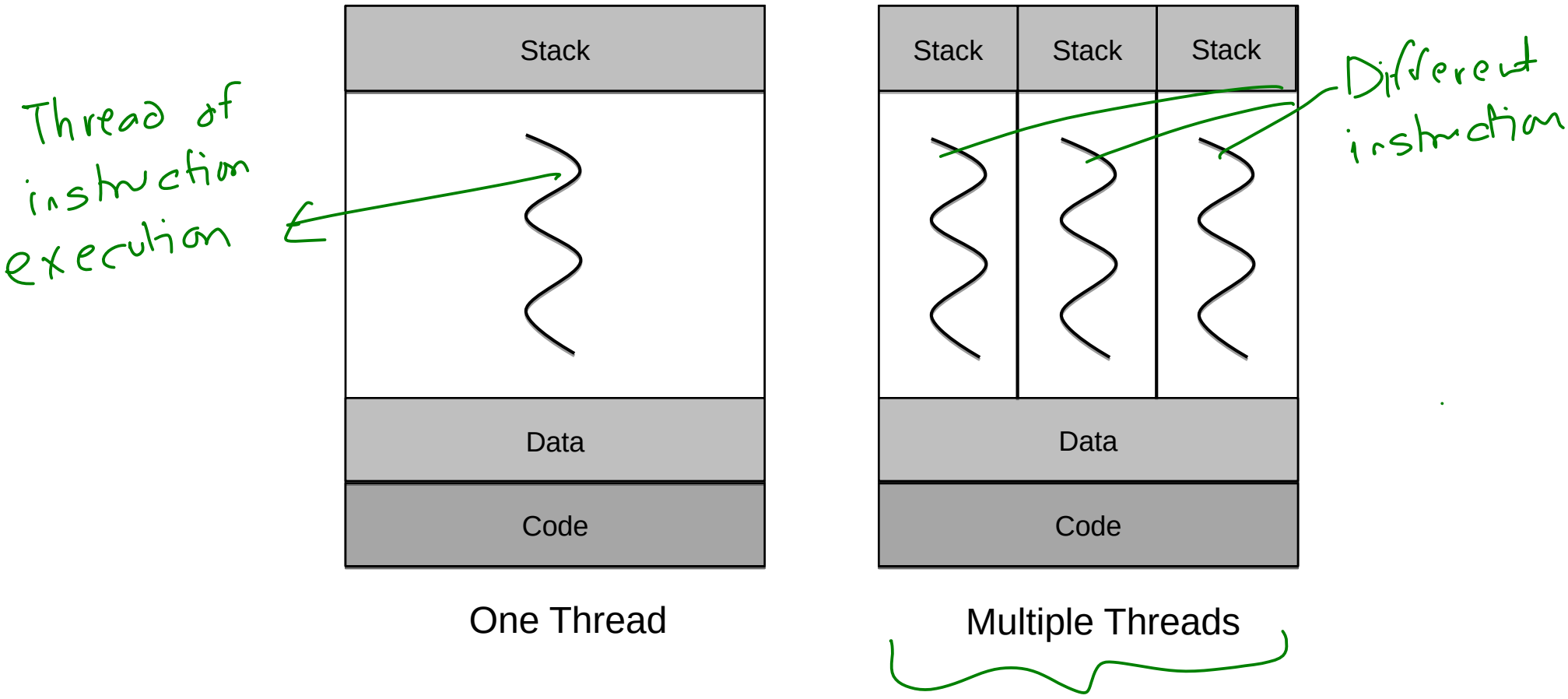
information sharing



Benefits of Threads

- Efficiency / economy
  - Less memory, fewer system resources
- Responsiveness
  - Lower startup time
- Easier resource sharing
  - Natural sharing of memory, open files, etc.
  - With caveats that we will discuss
- Concurrency
  - Utilization of multiple processors or cores

Single and Multithreaded Processes



The UNIX Thread Model

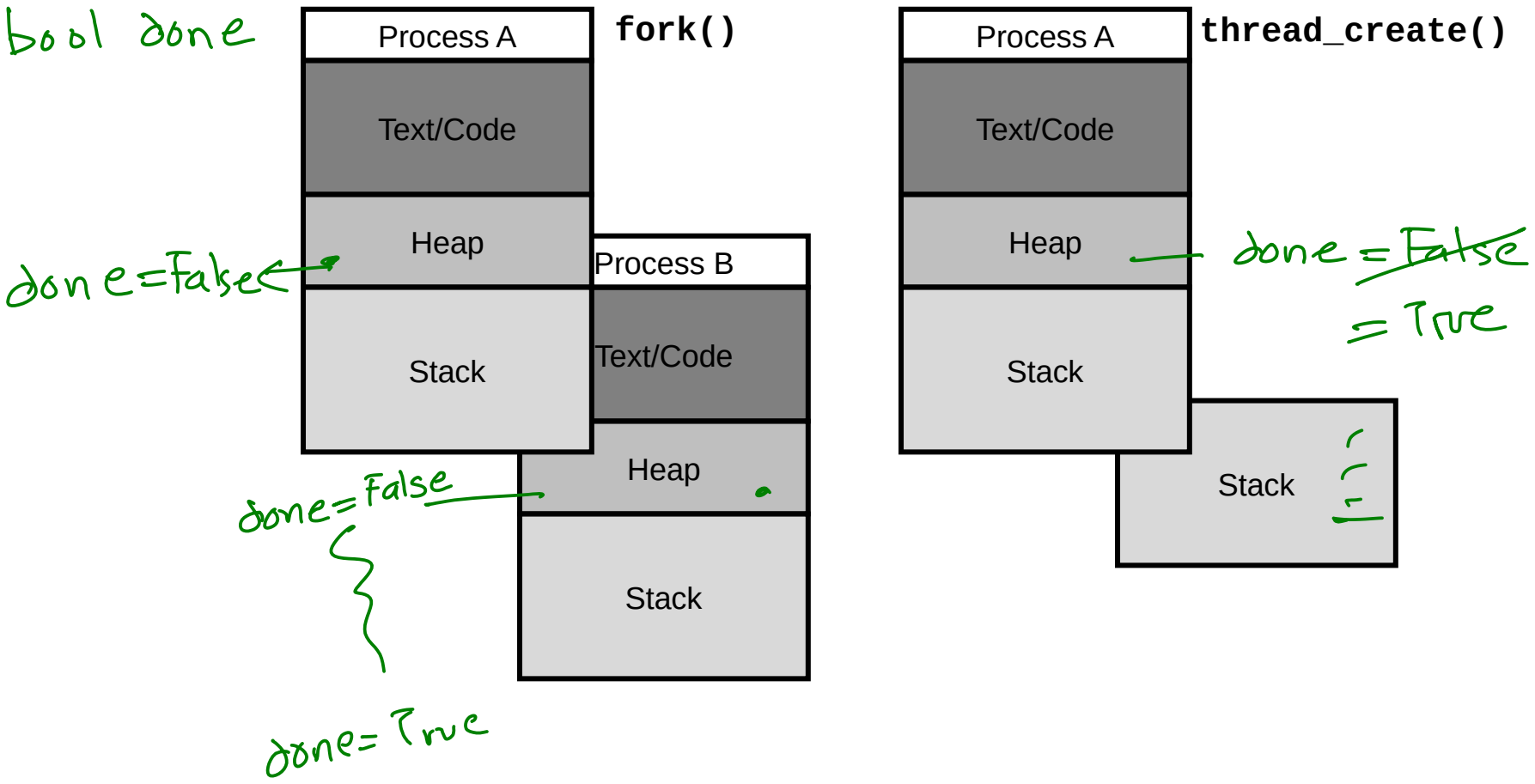
Per-Process Items

- Memory mapping
- Global variables
- Signal handlers
- Open files and file pointers

Per-Thread Items

- Program Counter
- Registers
- Stack
- Thread State [Running/Blocked]

Single and Multithreaded Processes



Pthreads

- In the old days, there were a variety of thread systems
  - Purely user-level systems
  - C Threads, -lthread
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to developers of the library
- Common in UNIX-like operating systems
  - Linux, Mac OS X
  - Available in Windows

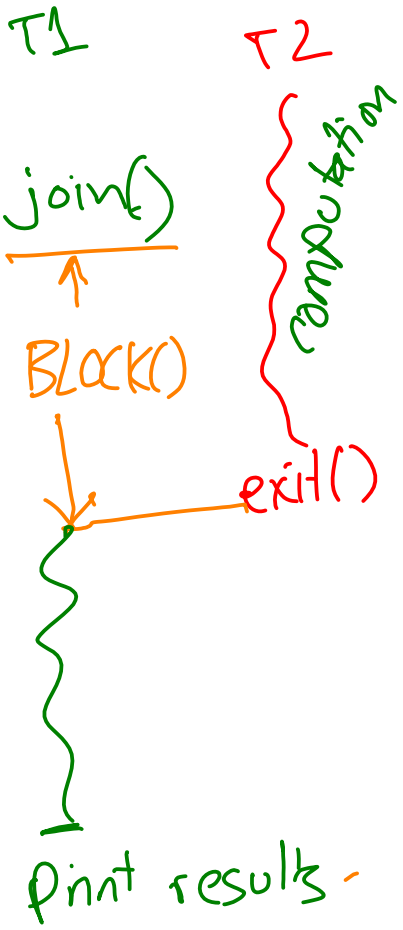
Creating a new thread

```
pthread_t thr_1, thr_2;

pthread_create(&thr_1, NULL, (void *)one, (void *)arg1);
```

- Pointer to a pthread_t
  - foo_t is POSIX convention for “of type foo”
  - pthread_t is a handle for the created thread
- Pointer to pthread_attr_t
  - Attributes of the thread, NULL gets the default
  - More in a bit
- Pointer to the entry function
- Pointer to the input data (void *)

Termination and joining (waiting)



- Pthreads terminate when the function returns, or the thread calls pthread_exit()

```
int pthread_exit (void *status);
```

- One thread can wait on the termination of another by using pthread_join()

```
pthread_t thr_1, thr_2;
pthread_join(&thr_1, void **status_ptr);
```

- pthread_t is the handle of the thread to be joined
- The 2nd argument is void **thread_return which will be filled with the value the thread gave to pthread_exit() or = to PTHREAD_CANCELLED

Complete Example

```
void f_one(int *);
void f_two(int *);
int result1, result2, arg1, arg2;

main(void) {
   pthread_t thr_1, thr_2;

   pthread_create(&thr_1, NULL, (void *)f_one, (void
  *)&arg1);
   pthread_create(&thr_2, NULL, (void *)f_two, (void
  *)&arg2);

   pthread_join(thr_1, NULL);
   pthread_join(thr_2, NULL);

   return 0;
}
```

Pthreads Summary

- Very useful programming tool
  - Changes the way all sorts of programs can be written – thread pools, etc.
- Include the header in a C program
  - #include <pthread.h>
  - Link with -lpthread
- pthread_create(&thr_1, NULL, (void *)one, (void *)arg1);
- pthread_join(&thr_1, NULL);
- sched_yield()
  - sleep will cause the thread to yield as well
- pthread_exit(void *retval)
- pthread_once
  - One-time initialization

Linux Threads

- The Linux scheduler deals with threads internally
  - refers to them as *tasks* rather than *threads*
- A thread is simply a new process that happens to share the same address space as its parent
- In this sense, Linux tasks are lightweight processes
- Linux processes are (heavyweight) processes
  - Groups of one or more tasks share
  - Memory map
  - Files

Linux Threads

- Thread creation is done through the **clone()** system call
- **clone()** allows a child task to share the address space of the parent task
- **fork()** creates a new process with its own entirely new process context
  - **fork()** is a wrapper for **clone()**
- Using **clone()** gives an application fine-grained control over exactly what is shared between two threads