CS 4504
Parallel and Distributed Computing

Process

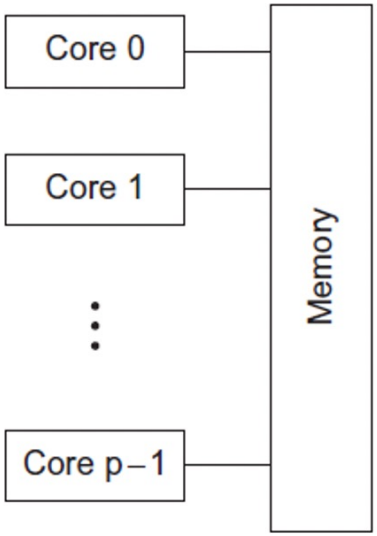
Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

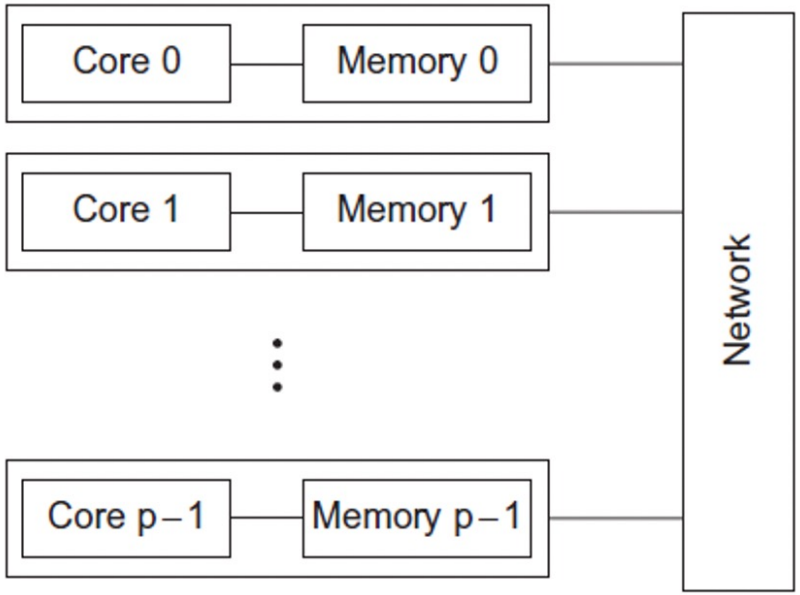
Test

- What is Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.



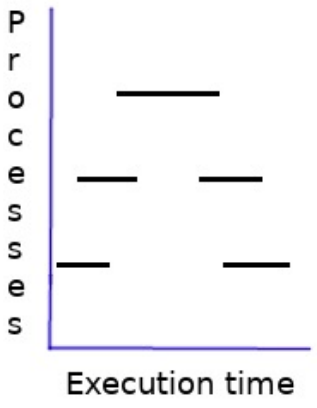
Test

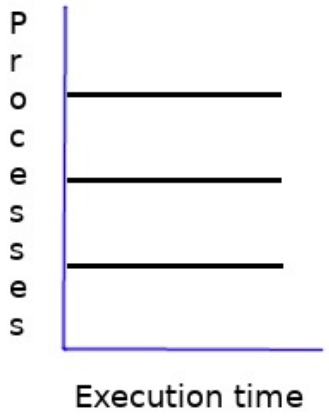
- What is Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.



Test

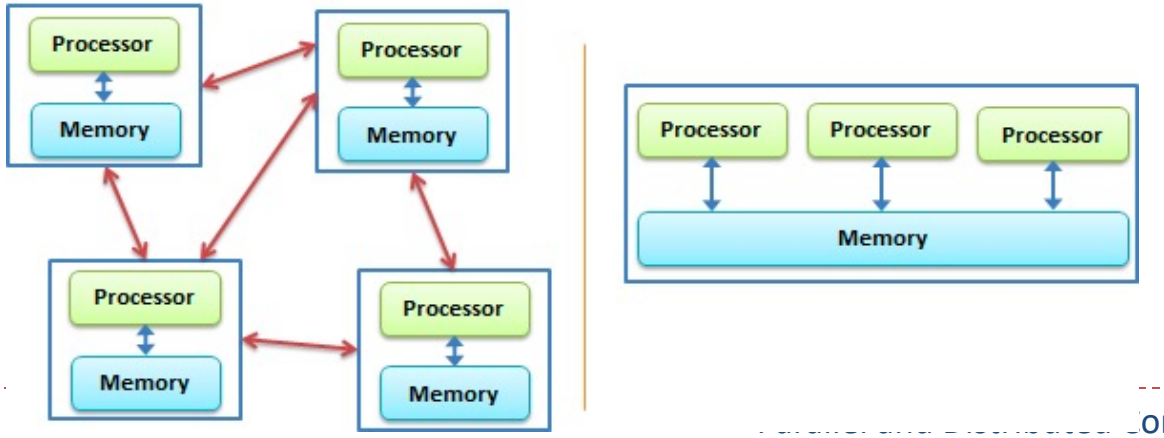
- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.





Test

- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.



Test

- Array A = Array B + Array C

```
int[] array1 = new int[] { 45, 36, 18, 53, 72, 30, 48, 93, 15, 3 };
int[] array2 = new int[] { 3, 46, 3, 53, 72, 30, 48, 93, 15, 3 };
int[] array3 = new int[10];
Parallel.For(0, array1.Length, i => {
    array3[i] = array1[i] + array2[i];
});
```

```
array3= {48,82,21,106,144,60,96,186,30,6};
```

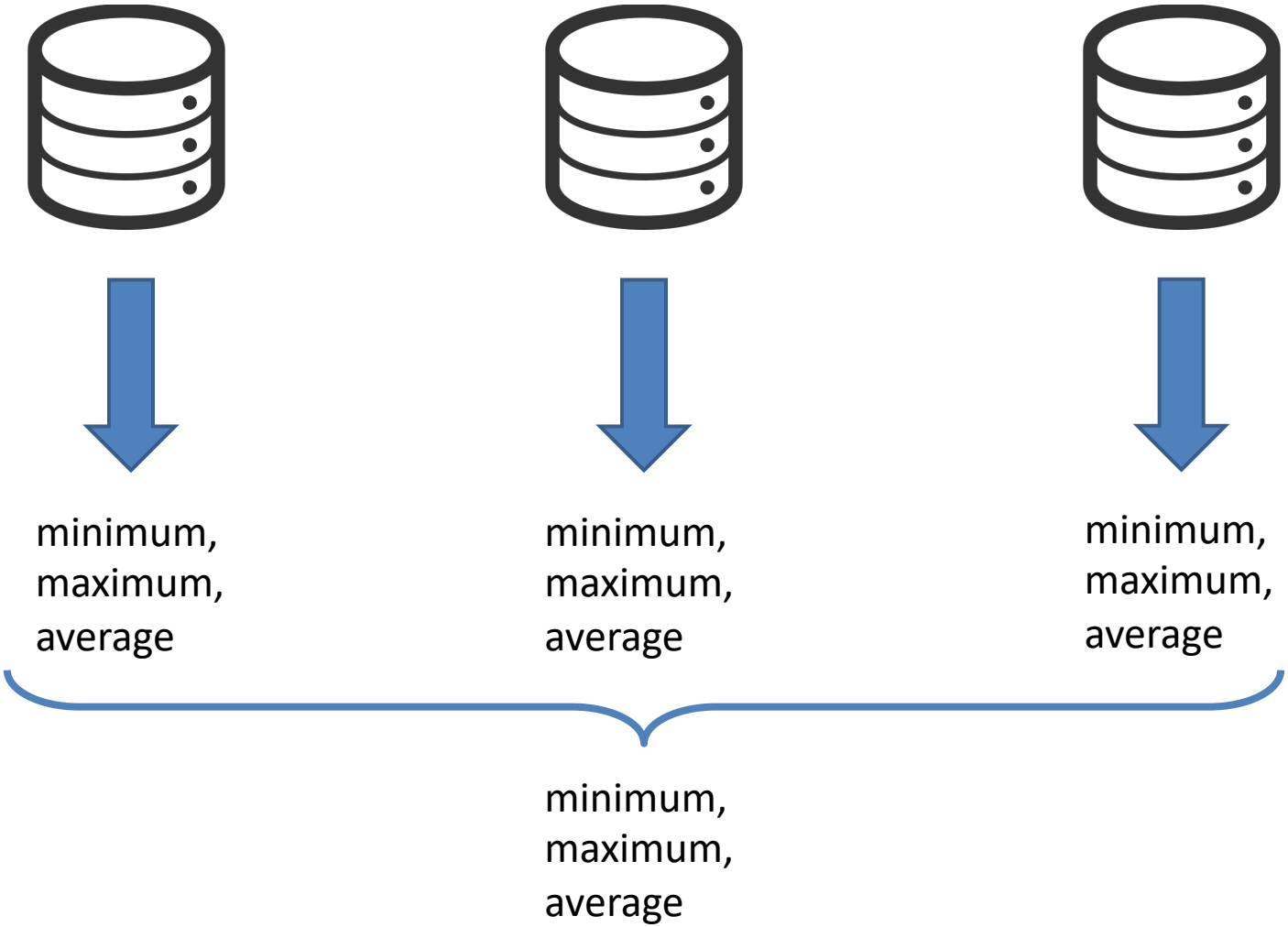
- Data operation in parallel (data parallelism) is relatively simple and concise, it does not require the programmer to care about how the parallel machine performs the operation

Test

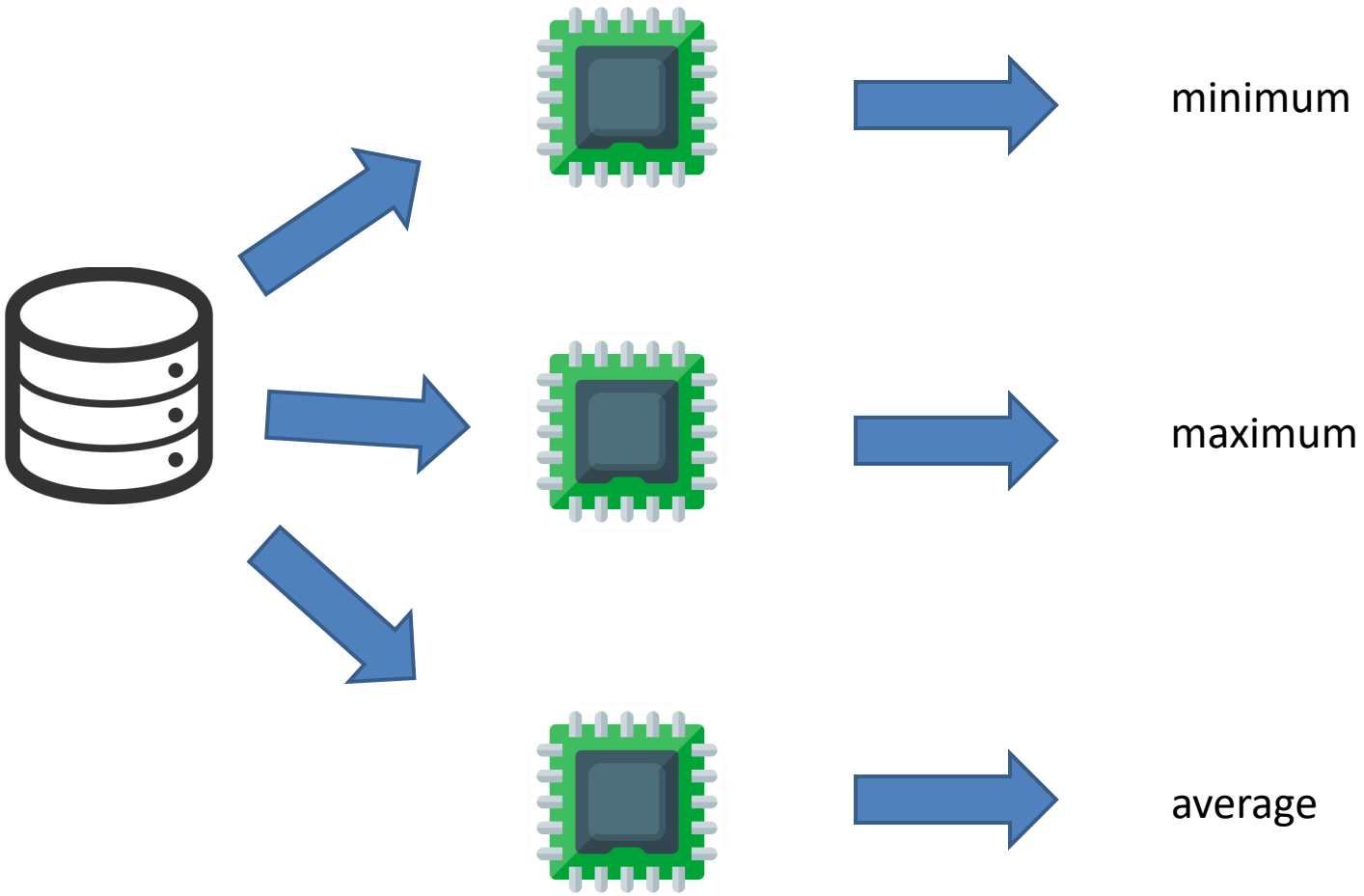
- You have a large data set, and you want to know the minimum, maximum, and average values of the data set

- How to design process this work using data parallelism or task parallelism?

Test: data parallelism



Test: task parallelism



Test

- Suppose we want to sort 8GB of data, and our machine's memory can hold so much data at once.

- Theoretically speaking, the sorting problem is already difficult to optimize from the algorithm level.

- The optimal complexity is O(nlogn). Mergesort = Quicksort = Heapsort = …

- How to run this problem in parallel?

Test: merge sort

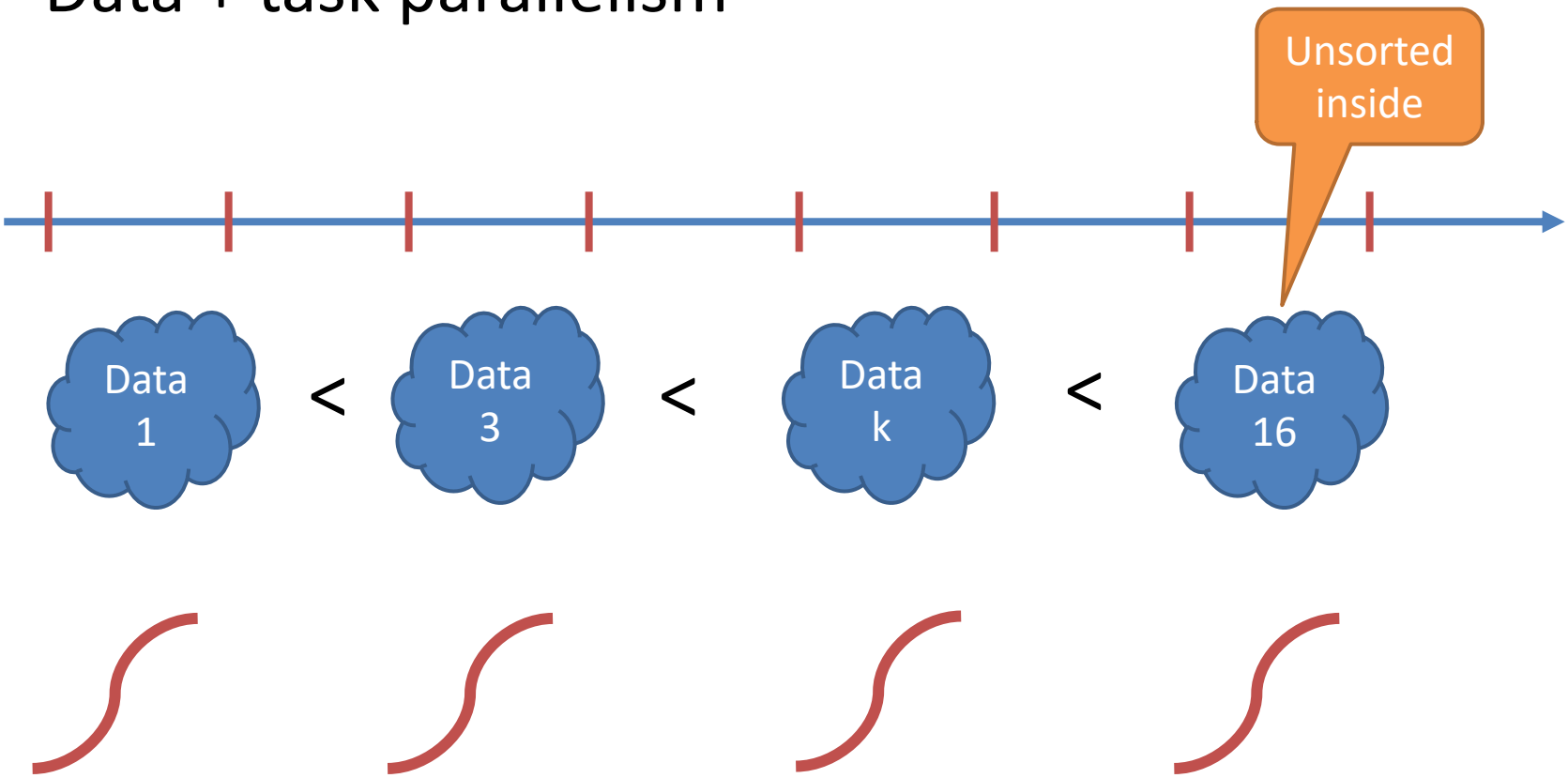
- Data + task parallelism
- 8GB data is divided into 16 small data sets, each set contains 500MB of data.
- We used 16 threads to sort these 16 500MB data sets in parallel. After the 16 small sets are sorted separately, we merge the 16 ordered sets.

Test: quick sort

- Data + task parallelism
- Scan the data to find the range of the data. We created 16 small intervals from small to large.
- We divide 8GB of data into corresponding intervals. For these 16 cells, we start 16 threads to sort in parallel.
- After the execution of all 16 threads, the data obtained is ordered data.

Test: quick sort

- Data + task parallelism



Outline

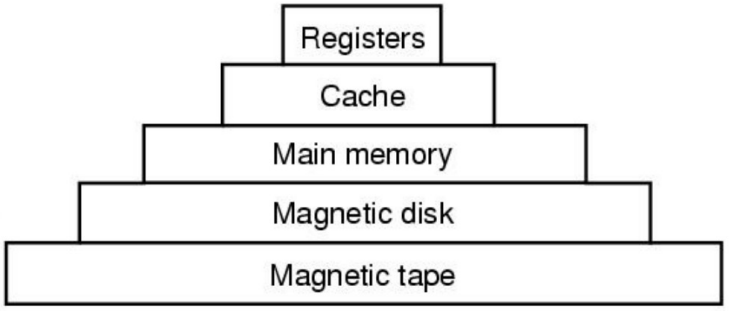
- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Process on Distributed OSes

Process

- Definition
  - An instance of a *program* running on a computer
  - An *abstraction* that supports running programs - -> cpu virtualization
  - An *execution stream* in the context of a particular *process state* - -> dynamic unit
  - A *sequential* stream of execution in its *own address space* - -> execution code line by line

Process

- Two parts of a process
  - Sequential execution of instructions
  - Process state
    - registers: PC (program counter), SP (stack pointer),…
    - Memory: address space, code, data, stack, heap …
    - I/O status: open files …

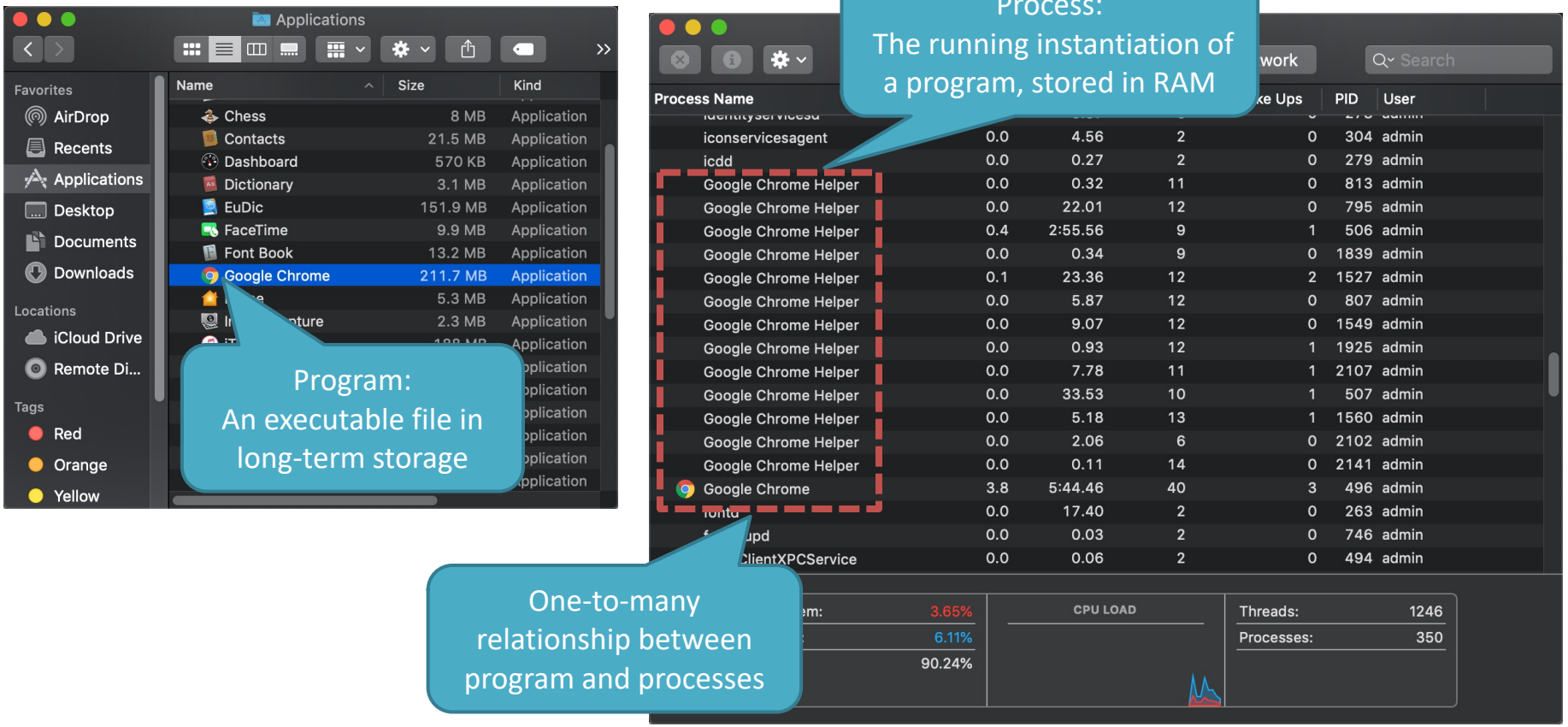


Program vs. Process

- Program != Process
  - Program = static code + data
  - Process = dynamic instantiation of code + data + files …

- No 1:1 mapping
  - Program : process = 1:N
    - A program can invoke many processes

Program vs. Process



Program vs. Process

BASIS FOR COMPARISON	PROGRAM	PROCESS
Basic	Program is a set of instruction.	When a program is executed, it is known as process.
Nature	Passive	Active
Lifespan	Longer	Limited
Required resources	Program is stored on disk in some file and does not require any other resources.	Process holds resources such as CPU, memory address, disk, I/O etc.

https://techdifferences.com/difference-between-program-and-process.html

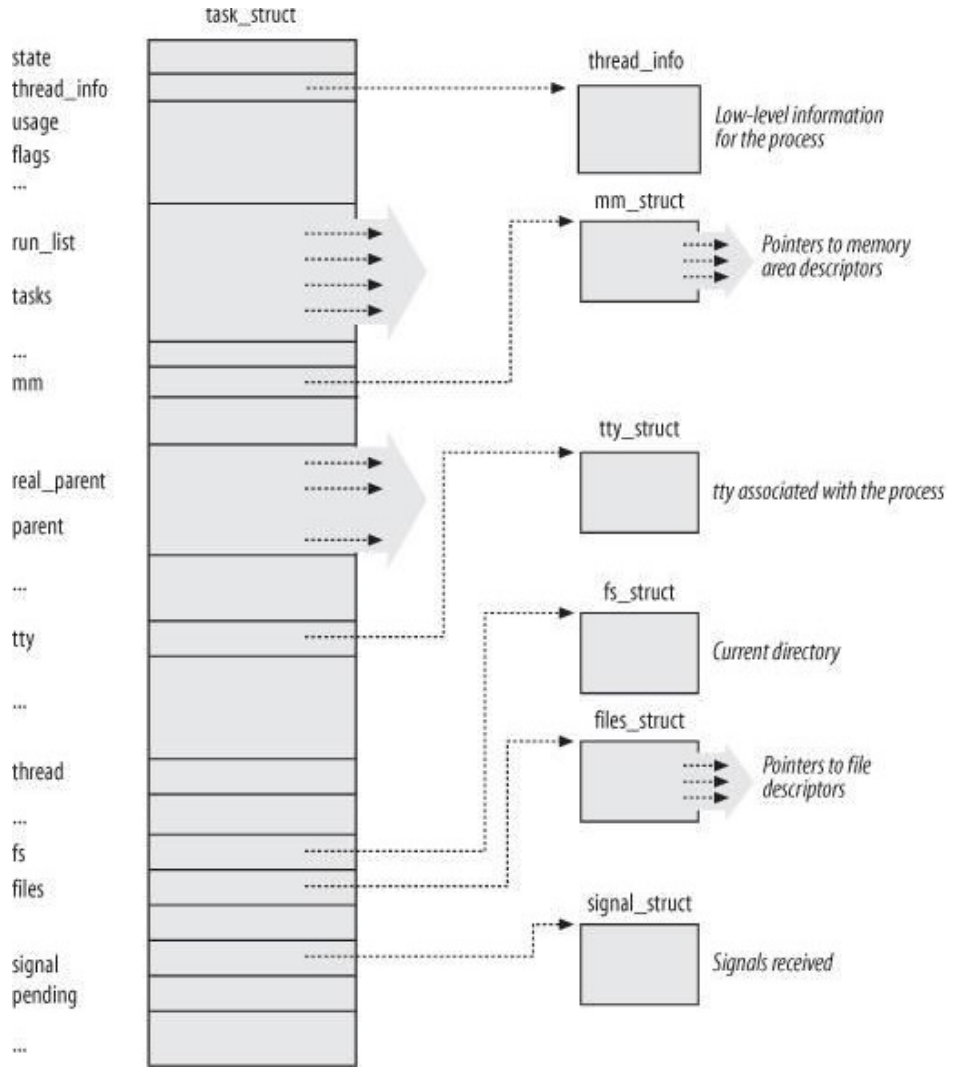
Process Descriptor

- Driving license
  - ID
  - Name
  - Address
  - Birth
  - Time
  - …
- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

Process in Linux

https://elixir.bootlin.com/linux/v5.4/source/include/linux/sched.h#L624

- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

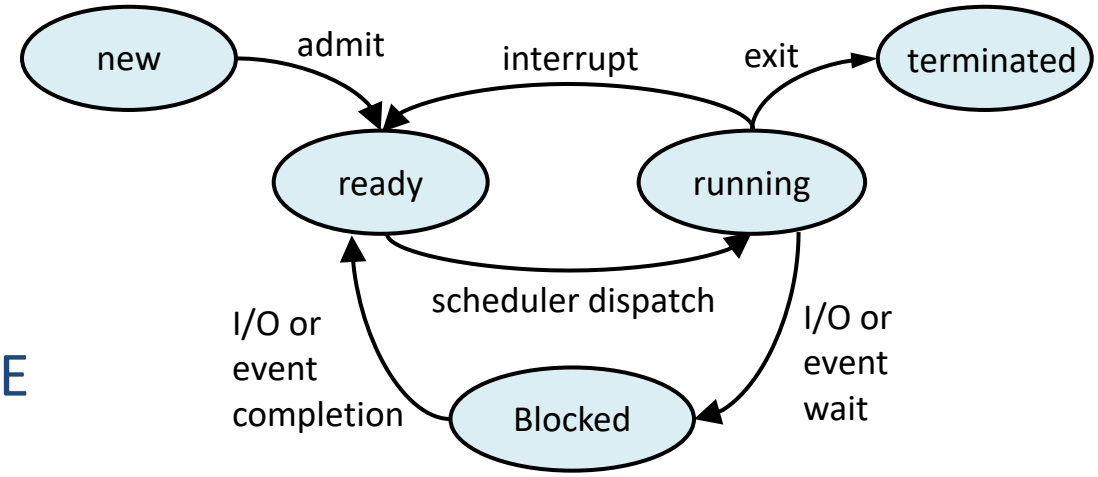


Process in Linux

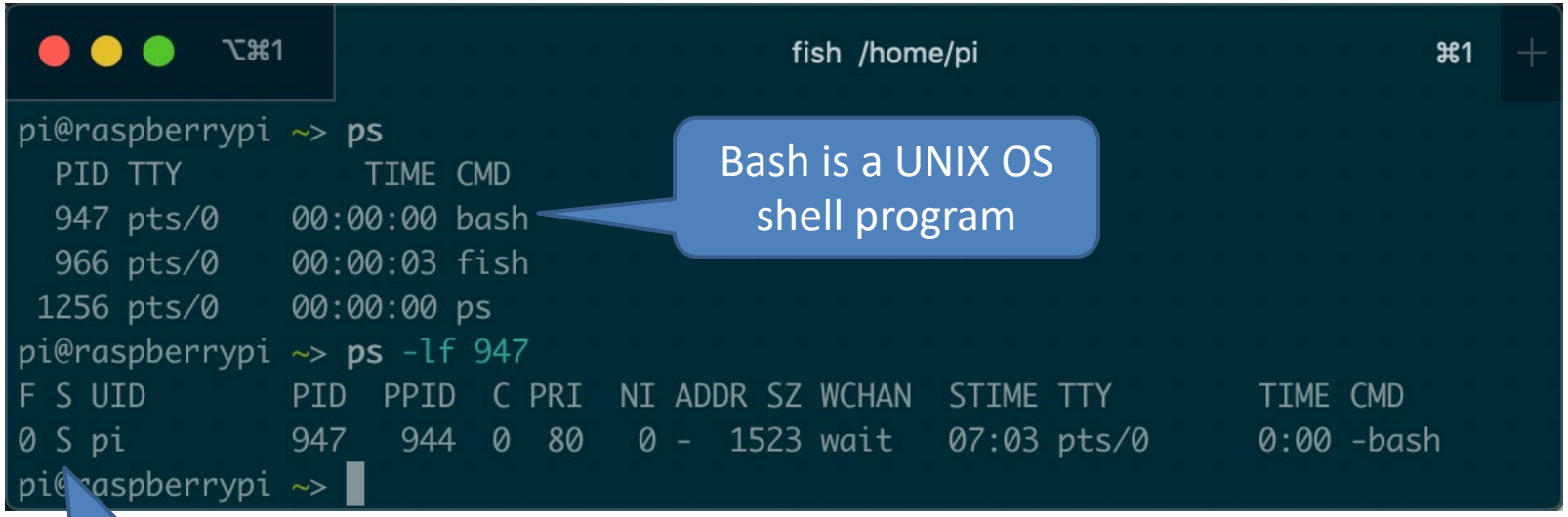
- Process control block (PCB)
  - State 
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

Linux PCB (5-state model to describe lifecycle of one process)

- State
  - TASK_RUNNING
    - Running, ready
  - TASK_INTERRUPTABLE
    - Blocked
  - EXIT_ZOMBIE
    - Terminated by not deallocated
  - EXIT_DEAD
    - Completely terminated



$ ps -lf : process information



R： The process is running
S： The process is sleeping/idle
T： The process is terminated
Z： The process is in zombie state

$ ps -lf : process information

https://github.com/kevinsuo/CS7172/blob/master/sleep.c

https://github.com/kevinsuo/CS7172/blob/master/loop.c

What is the state of the process?

Download: wget Raw-file-URL

Compile: gcc [file-name].c –o [file-name].o

Run: ./[file-name].o

```
pi@raspberrypi ~> ps -lf 8021
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S pi        8021  6190  0  80   0 -   450 hrtime 18:02 pts/0      0:00 ./test.o
```

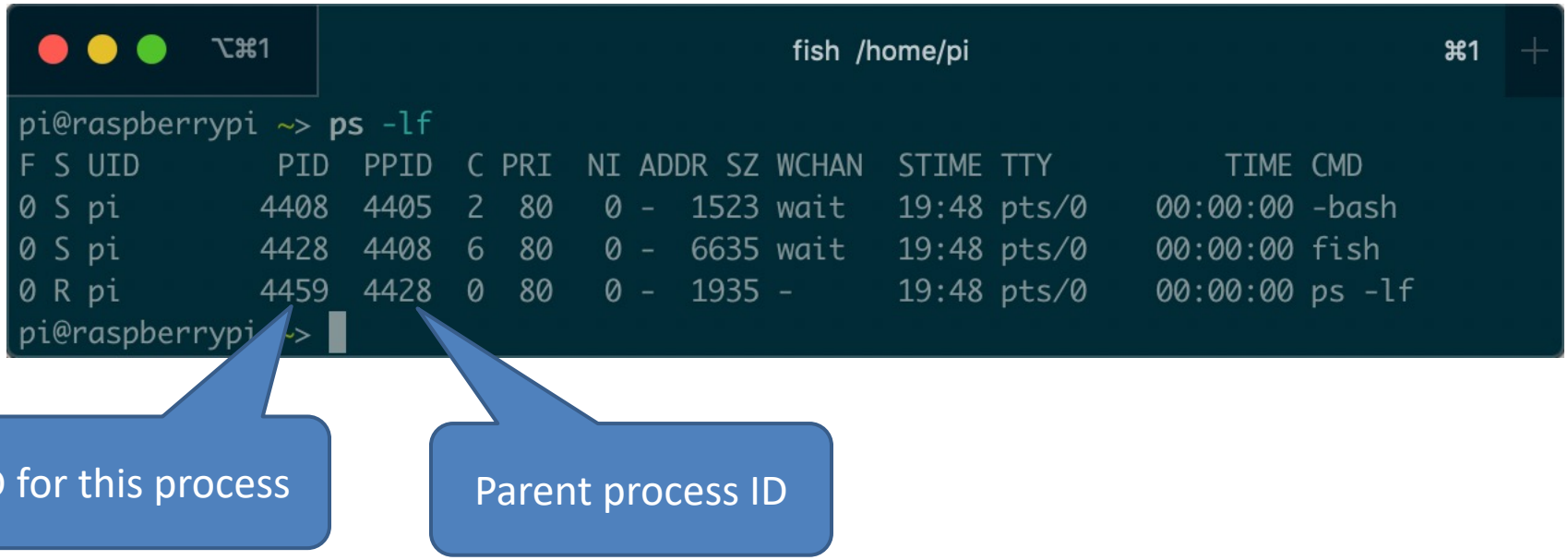
```
pi@raspberrypi ~> ps -lf 10057
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 R pi       10057  6190 99  80   0 -   450 -      18:06 pts/0      0:14 ./test.o
```

Process in Linux

- Process control block (PCB)
  - State
  - Identifiers 
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

Linux Process Control Block (cont')

- Identifiers
  - pid: ID of the process



Process in Linux

- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info 
  - File system
  - Virtual memory
  - Process specific context
  - …

Linux Process Control Block (cont')

- Scheduling information
  - prio, static_prio, normal_prio
  - rt_priority
  - sched_class

Linux Process Control Block (cont')

- Scheduling information
  - prio, static_prio, normal_prio

(1) Static priority: P1 > P2 = P3 = P4, P1 can execute whenever it needs;
(2) Normal priority: P1 = P2 = P3 = P4, P1 execute depending on the scheduling algorithm;
(3) Prio: dynamic priority, will change over the time

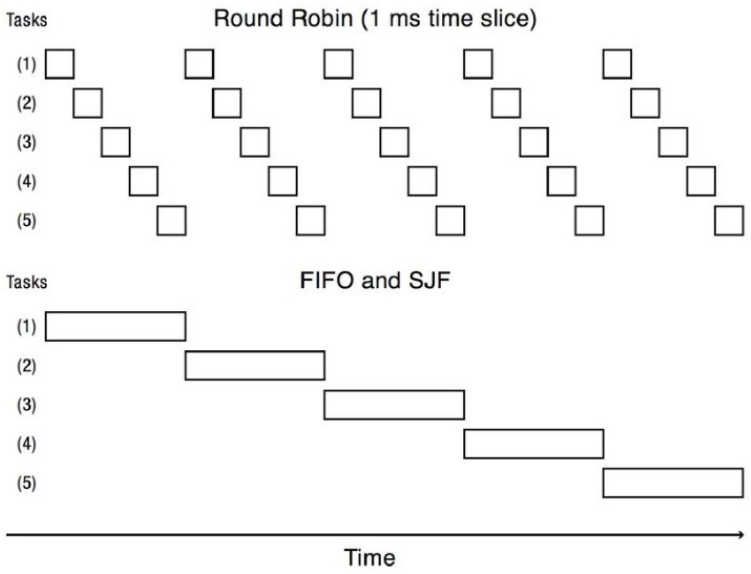
Linux Process Control Block (cont')

- Scheduling information
  - rt_priority

Rt_priority process is always higher than other priority of processes and will be scheduled immediately when it needs

Linux Process Control Block (cont')

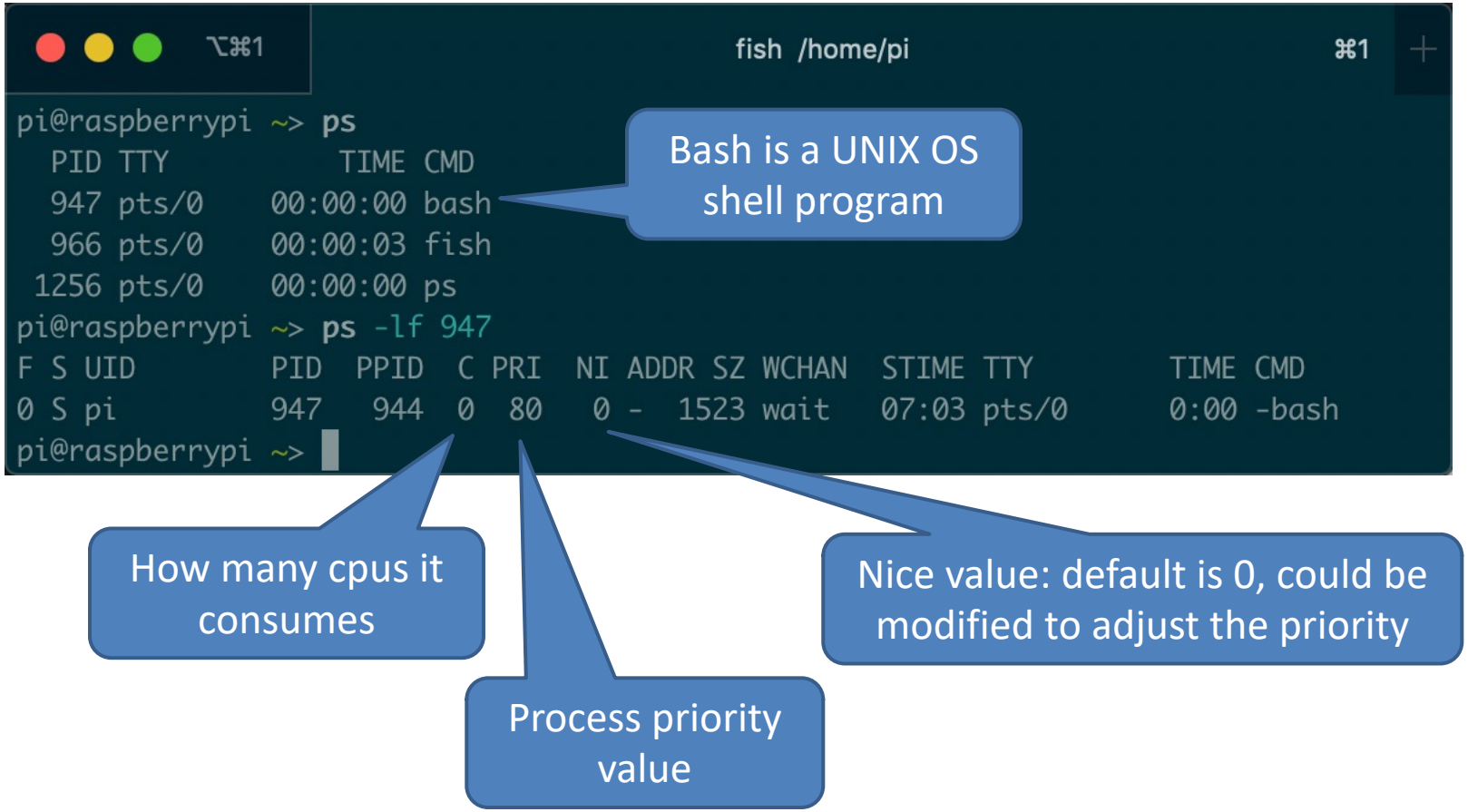
- Scheduling information
  - sched_class: different scheduling policy implementations, e.g., FIFO, SJF, RR…
    - Task->sched_class->pick_next_task(runqueue)



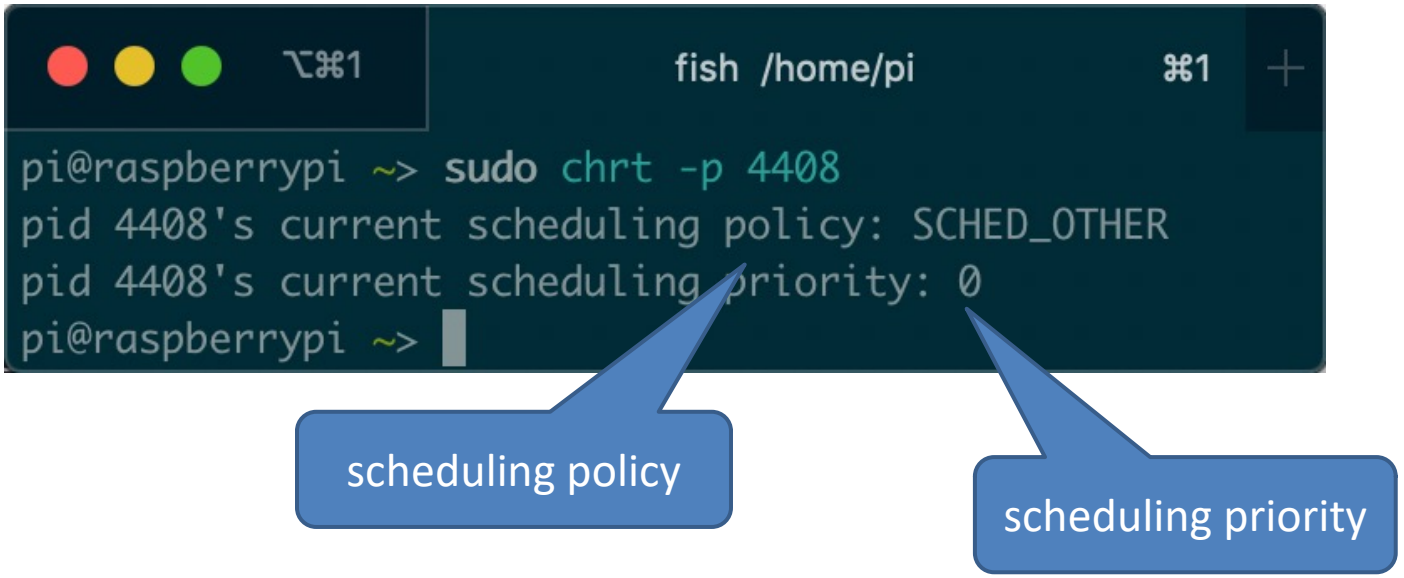
pick_next_task of RR:
pick based on time cycle

pick_next_task of FIFO:
pick based on order

$ ps -lf : process information



$ chrt: process scheduling info



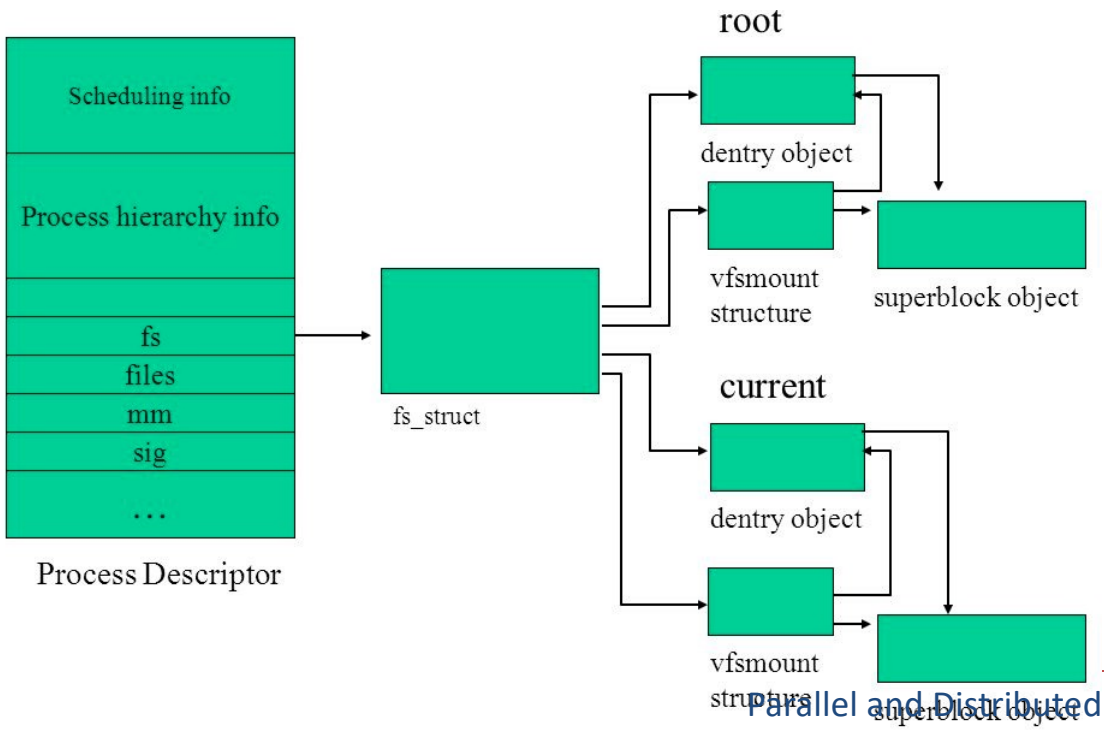
SCHED_OTHER	min/max priority : 0/0
SCHED_FIFO	min/max priority : 1/99
SCHED_RR	min/max priority : 1/99
SCHED_BATCH	min/max priority : 0/0

Process in Linux

- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system 
  - Virtual memory
  - Process specific context
  - …

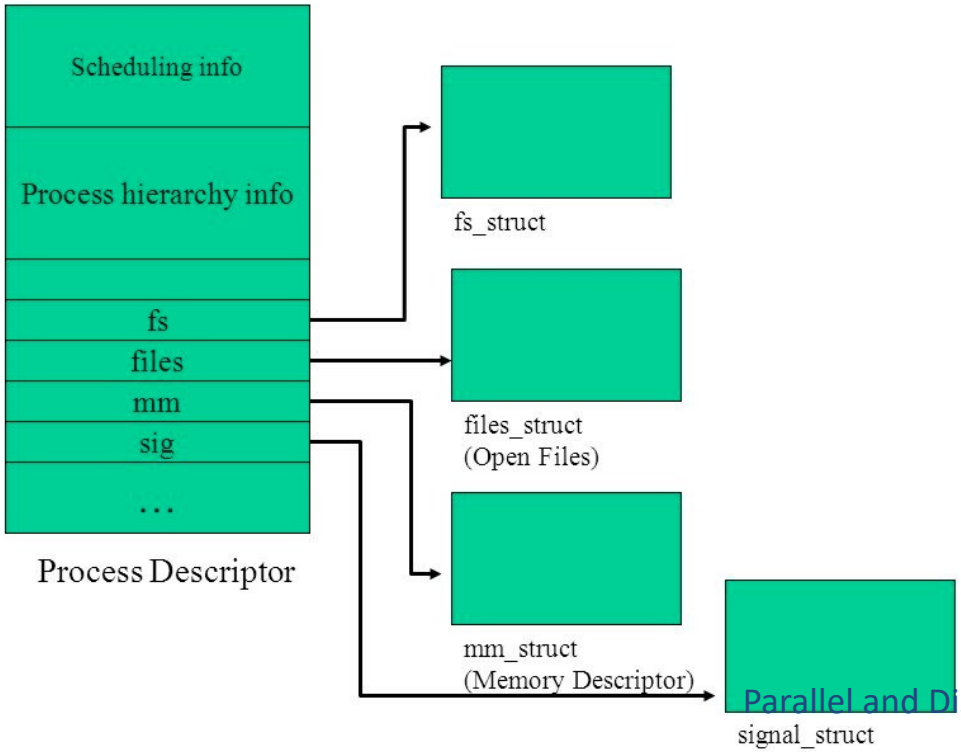
Linux Process Control Block (cont')

- Files
  - fs_struct https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1525
    - file system information: root directory, current directory

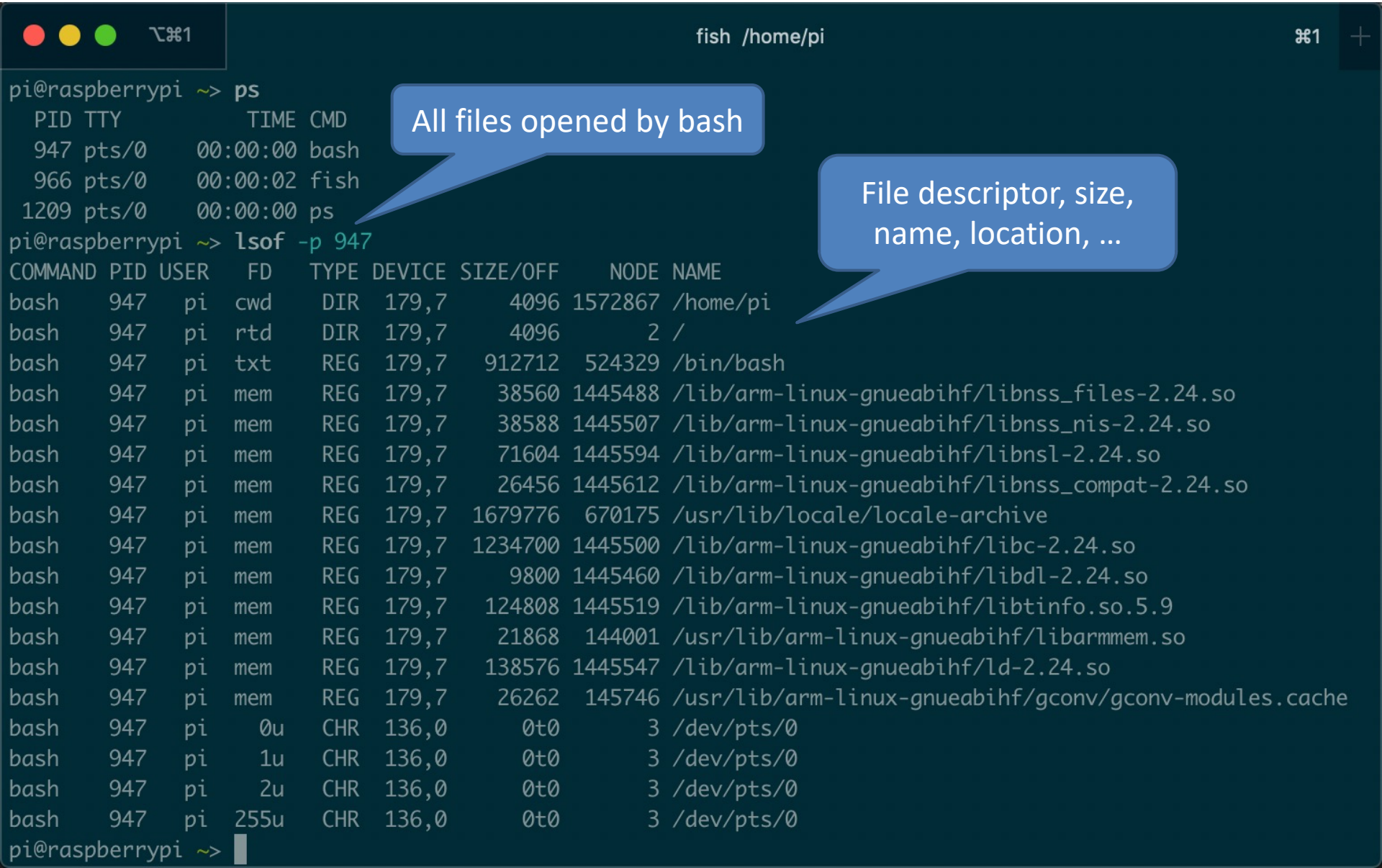


Linux Process Control Block (cont')

- Files
  - files_struct https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1528
    - Information on opened files



$lsof: list all open files

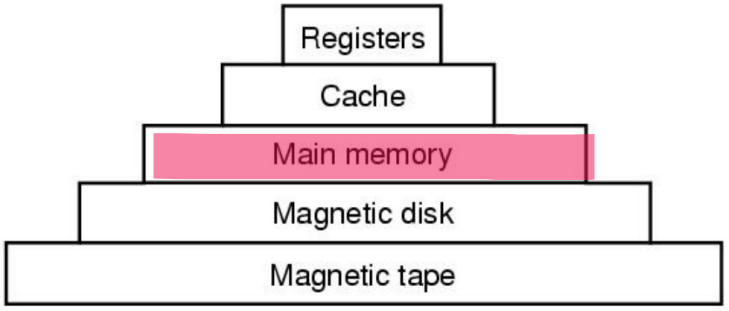


Process in Linux

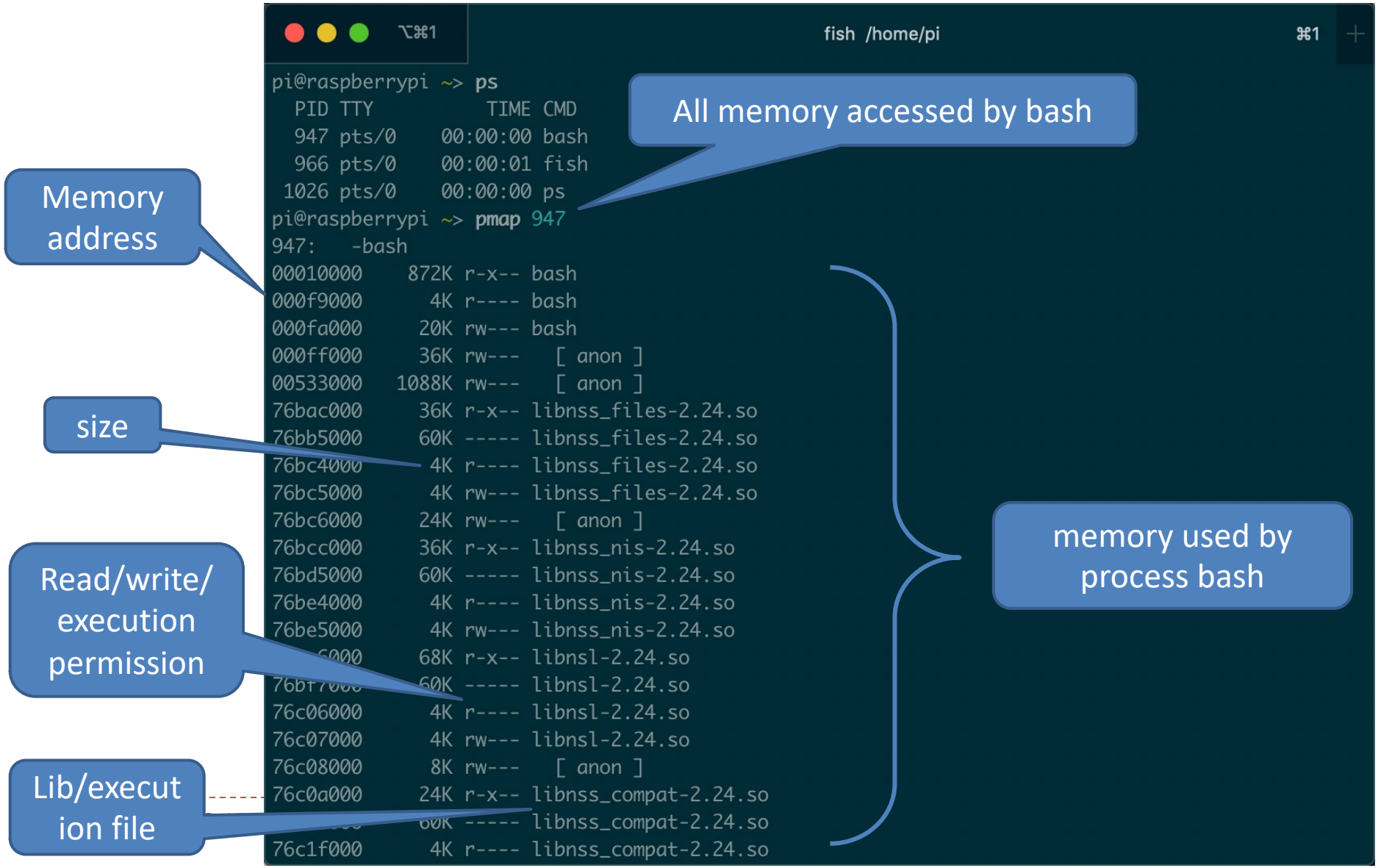
- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory 
  - Process specific context
  - …

Linux Process Control Block (cont')

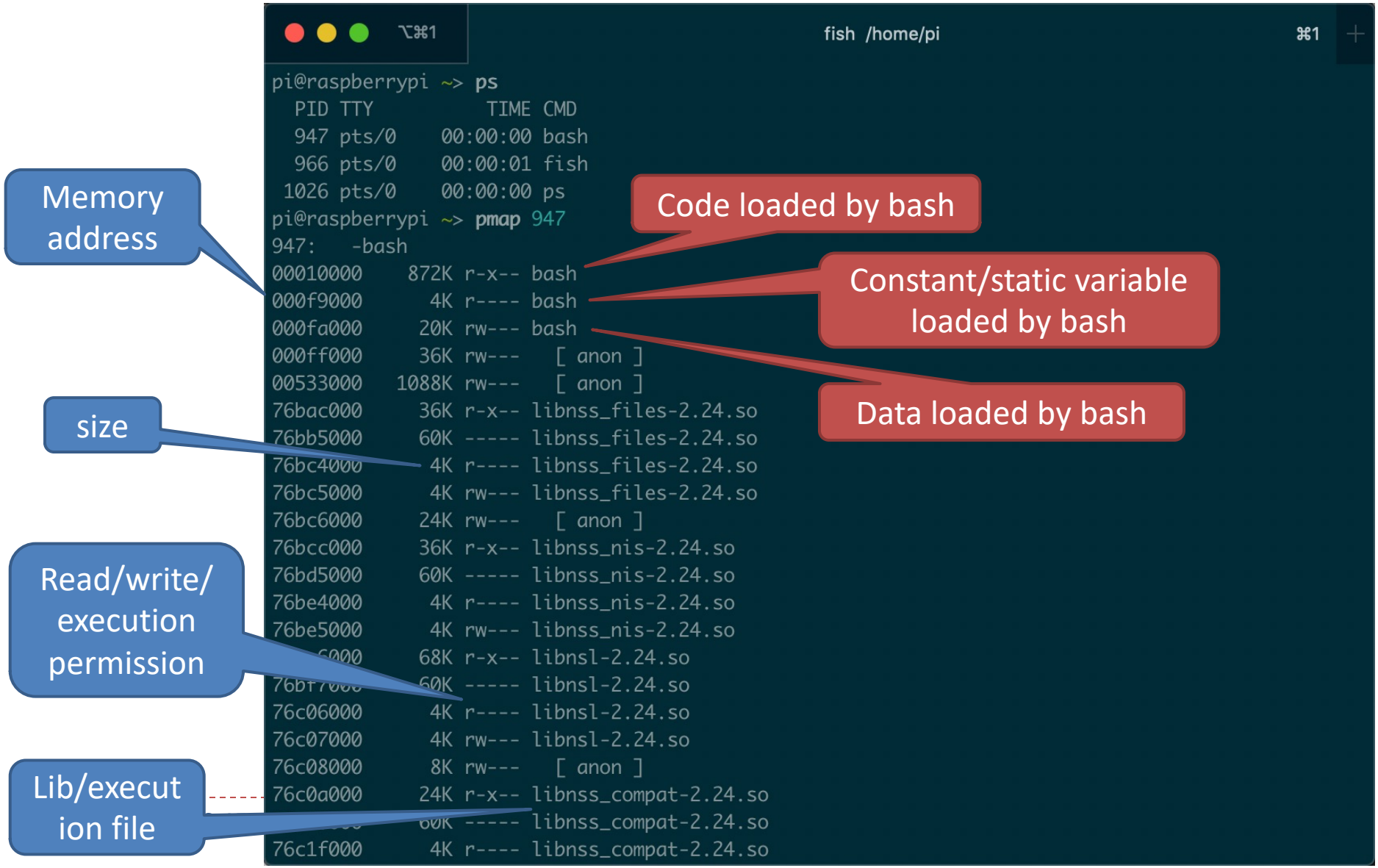
- Virtual memory
  - mm_struct: describes the content of a process's virtual memory
    - The pointer to the page table and the virtual memory areas



$pmap: memory mapping

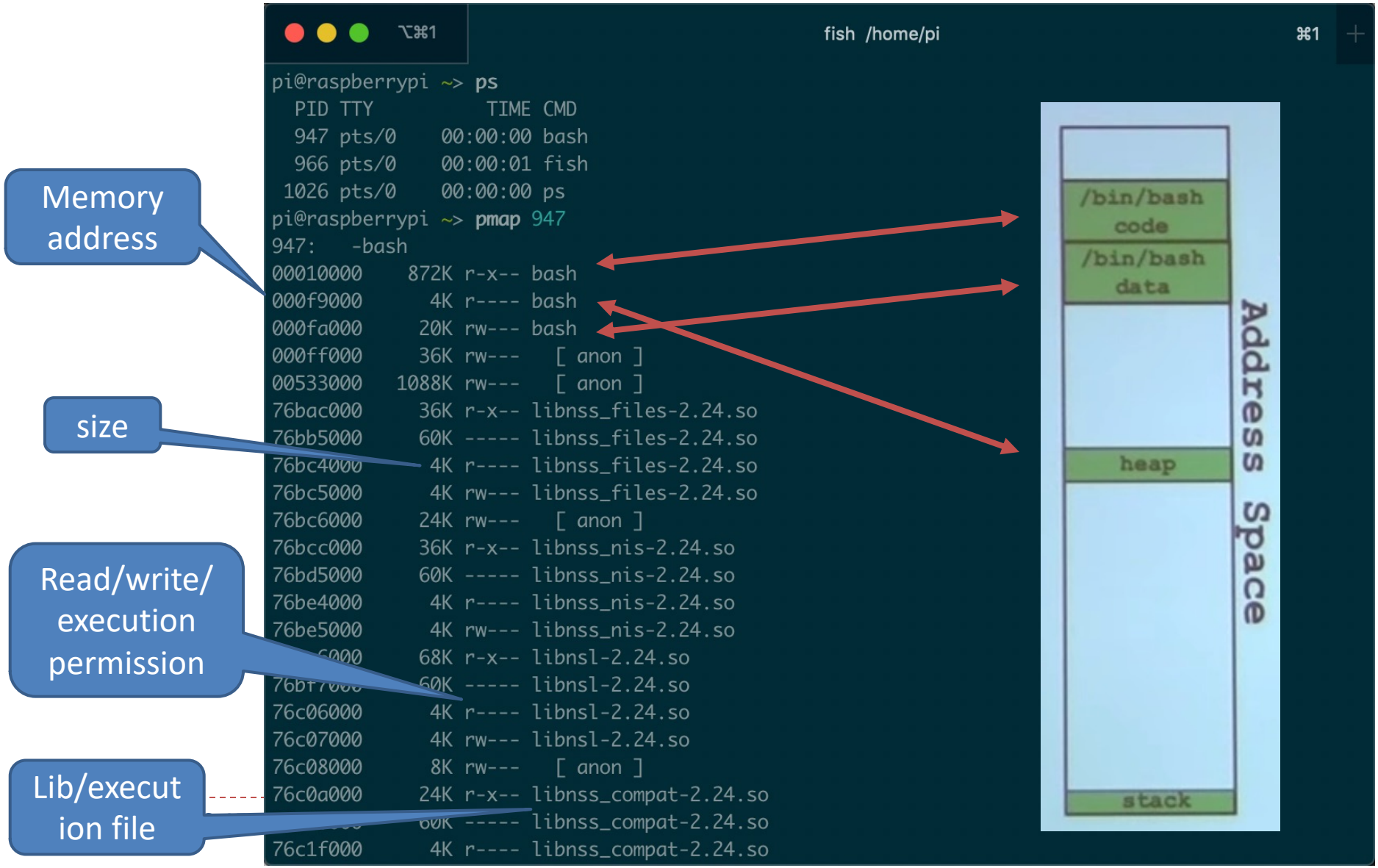


$pmap: memory mapping



$pmap: memory mapping

Bash in memory



Outline

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Process on Distributed OSes

Test: create a process and show the following information of it

https://github.com/kevinsuo/CS7172/blob/master/sleep.c

Download: wget Raw-file-URL

Compile: gcc [file-name].c –o [file-name].o

Run: ./[file-name].o

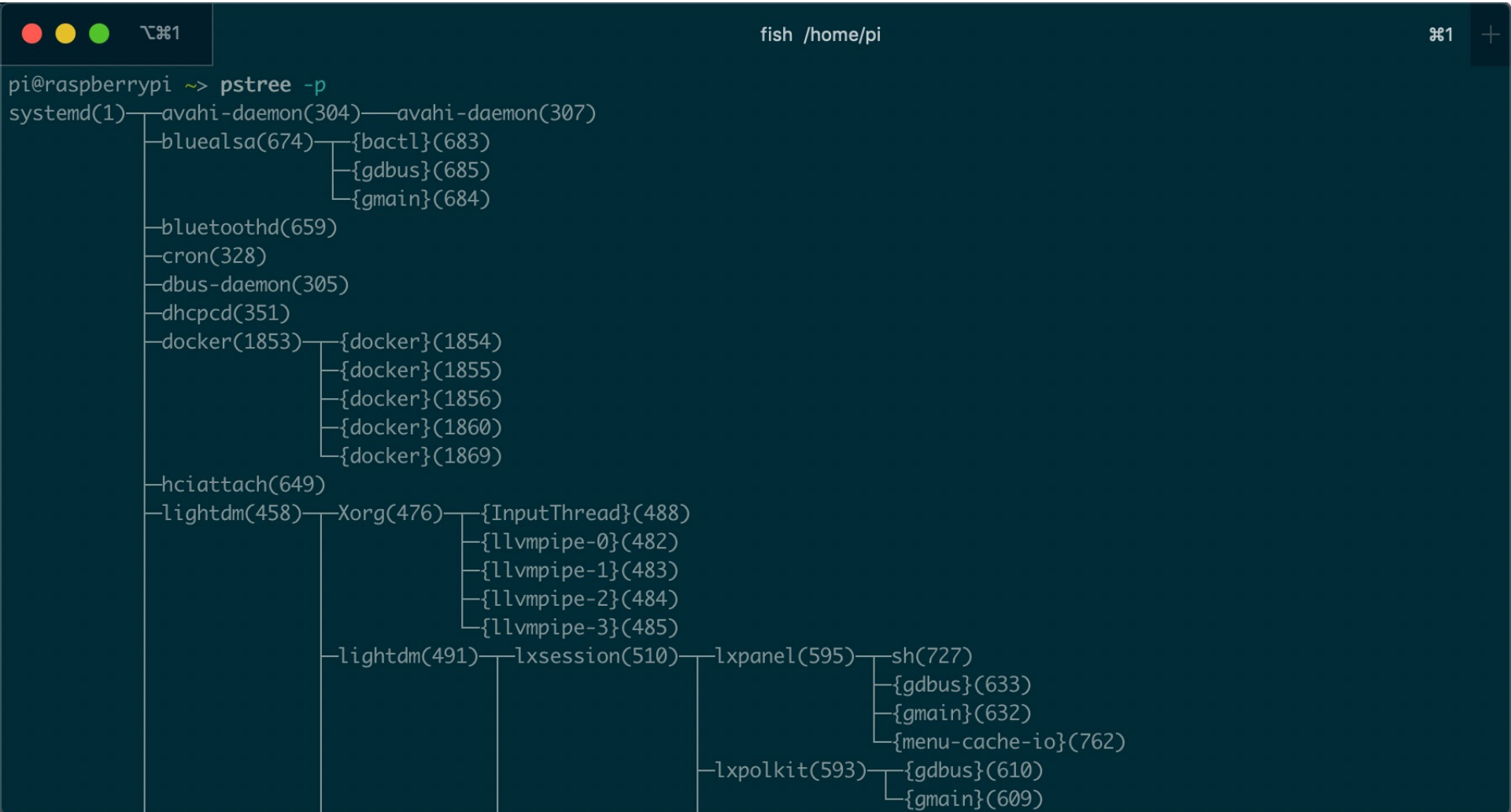
- What is the process ID?
- What is the process state?
- What is the scheduling policy for the process?

```
SCHED_OTHER     min/max priority : 0/0
SCHED_FIFO      min/max priority : 1/99
SCHED_RR        min/max priority : 1/99
SCHED_BATCH     min/max priority : 0/0
```

- Show all files the process it is using.
- Show all memory the process it is using.

Where do processes come from?

- Process creation always uses fork() system call



Where do processes come from? First process in the kernel

```
asmlinkage __visible void __init start_kernel(void)
{
        char *command_line;
        char *after_dashes;

        set_task_stack_end_magic(&init_task);
        smp_setup_processor_id();
        debug_objects_early_init();

        cgroup_init_early();

        local_irq_disable();
        early_boot_irqs_disabled = true;

        /*
         * Interrupts are still disabled. Do necessary setups, then
         * enable them.
         */
        boot_cpu_init();
        page_address_init();
        pr_notice("%s", linux_banner);
        setup_arch(&command_line);
```

https://elixir.bootlin.com/linux/v5.4/source/init/main.c#L580

- The start of linux kernel begins from start_kernel() function, it is equal to the main function of kernel
- set_task_stack_end_magic() creates the first process in the OS
- The first process is the only one which is not created by fork function

Fork() system call

- Process creation always uses fork() system call
- When?
  - User runs a program at command line
    - $./test.o
  - OS creates a process to provide a service
    - Timer, networking, load-balance, daemon, etc.
  - One process starts another process
    - Parents and child process

Fork() system call

https://github.com/kevinsuo/CS7172/blob/master/fork.c

1. vim /home/pi/Downloads (ssh)

vim /home/pi/Dow... ⌘1

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
        int pid;
        pid = fork();

        if (pid < 0) {
                printf("Fork error!");
        } else if (pid == 0) {
                printf("The child pid is %d, pid: %d\n", getpid(), pid);
        } else {
                printf("The parent pid is %d, pid: %d\n", getpid(), pid);
        }

        return 0;
}
```

Fork() system call

- fork() is called once. But it returns twice!!
  - Once in the parent (return child id > 0)
  - Once in the child (return 0)

Fork() system call

https://github.com/kevinsuo/CS7172/blob/master/fork.c

1. vim /home/pi/Downloads (ssh)

vim /home/pi/Dow... ⌘1

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
        int pid;
        pid = fork();

        if (pid < 0) {
                printf("Fork error!");
        } else if (pid == 0) {
                printf("The child pid is %d, pid:%d\n", getpid(), pid);
        } else {
                printf("The parent pid is %d, pid:%d\n", getpid(), pid);
        }

        return 0;
}
```

fork() is called once.
But it returns twice!!

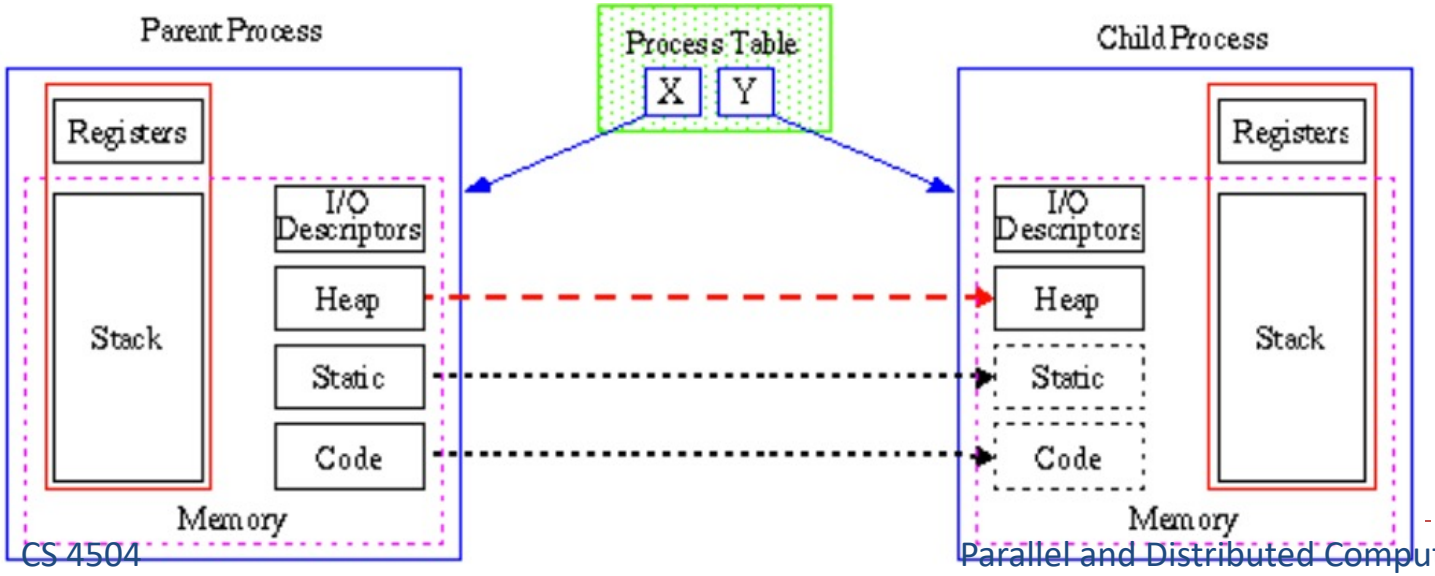
```
pi@raspberrypi ~/Downloads> ./test.o
The parent pid is 2510, pid:2511
The child pid is 2511, pid:0
```

Fork() system call

- fork() is the UNIX system call that creates a new process.
- fork() creates a new process that is a **copy** of the calling process.
- After fork () we refer to the caller as the **parent** and the newly-created process as the **child**. They have a special relationship and special responsibilities.

Parent process and child process

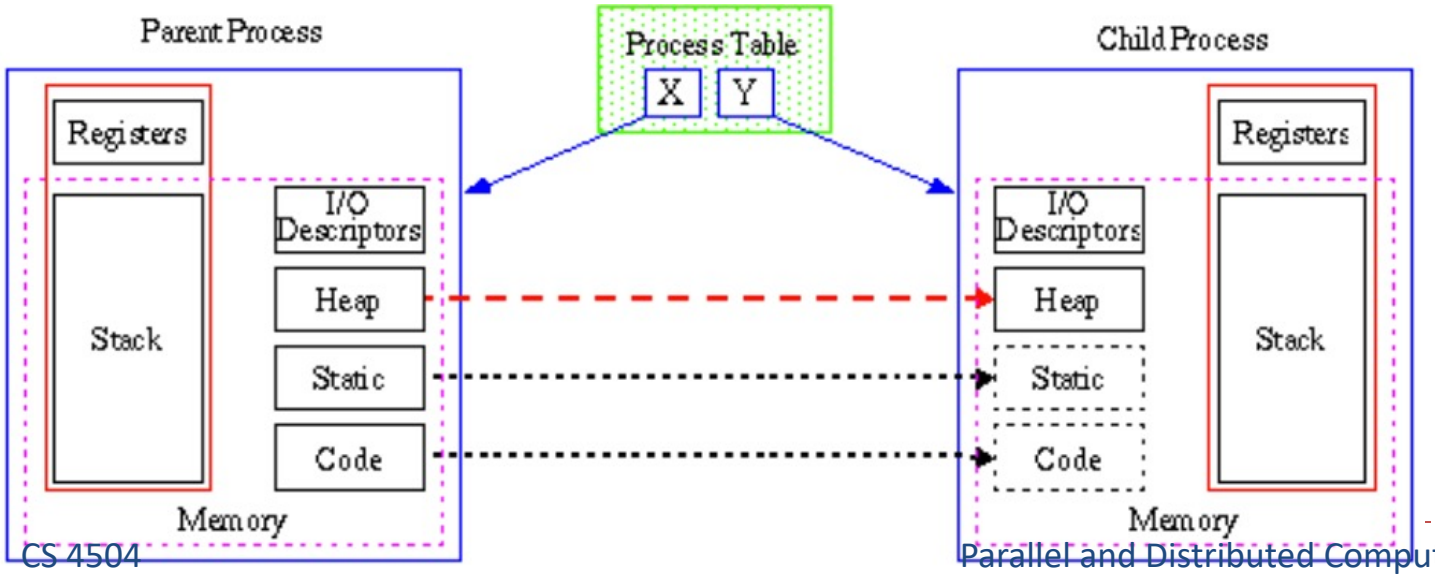
- When a parent process uses fork() to create a child process, the two processes have
  - the **same** program text.
  - but **separate** copies of the data, stack, and heap segments.



Parent process and child process

- The child's stack, data, and heap segments are **initially** exact **duplicates** of the corresponding parts the parent's memory.

- After the fork(), each process can modify the variables in its **own** data, stack, and heap segments without affecting the other process.



Fork() example

1. vim /home/pi/Downlo

vim /home/pi/Dow... ⌘1

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        fork();

        printf("Hello world!\n");

        return 0;
}
```

Parent

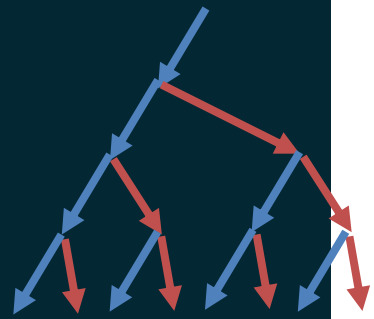
Child

```
pi@raspberrypi ~/Downloads> ./a.o
Hello world!
Hello world!
```

1. vim /home/pi/Dowr

vim /home/pi/Dow... ⌘1

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        fork();

        fork();

        fork();

        printf("Hello world!\n");

        return 0;
}
```

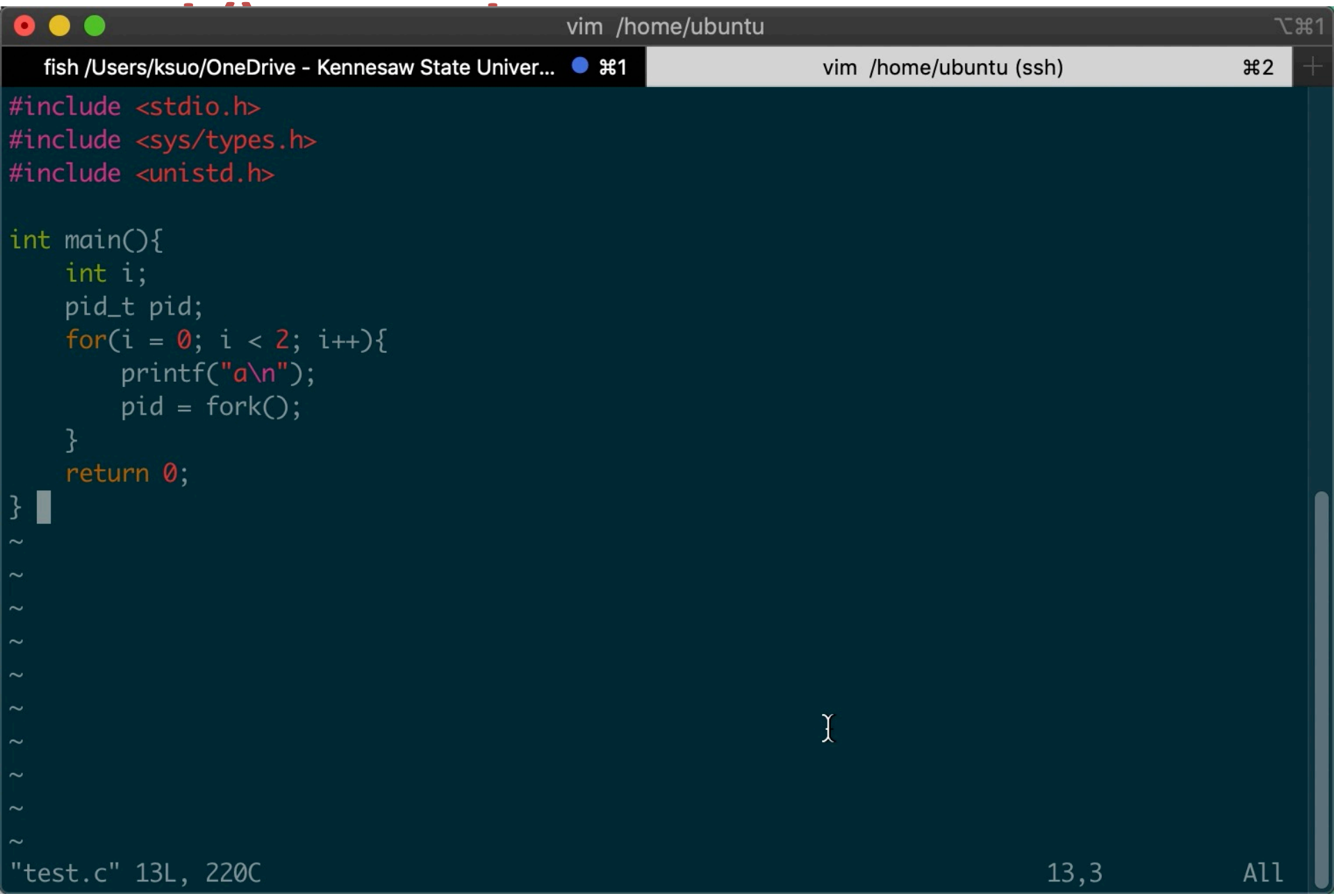


```
pi@raspberrypi ~/Downloads> ./a.o
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Fork() example

- How many a it will output?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a\n");
        pid = fork();
    }
    return 0;
}
```


vim /home/ubuntu
⌥⌘1
fish /Users/ksuo/OneDrive - Kennesaw State Univer... ⌘1
vim /home/ubuntu (ssh) ⌘2
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
int i;
pid_t pid;
for(i = 0; i < 2; i++){
printf("a\n");
pid = fork();
}
return 0;
}
"test.c" 13L, 220C
13,3
All

Fork() example

- How many a it will output?

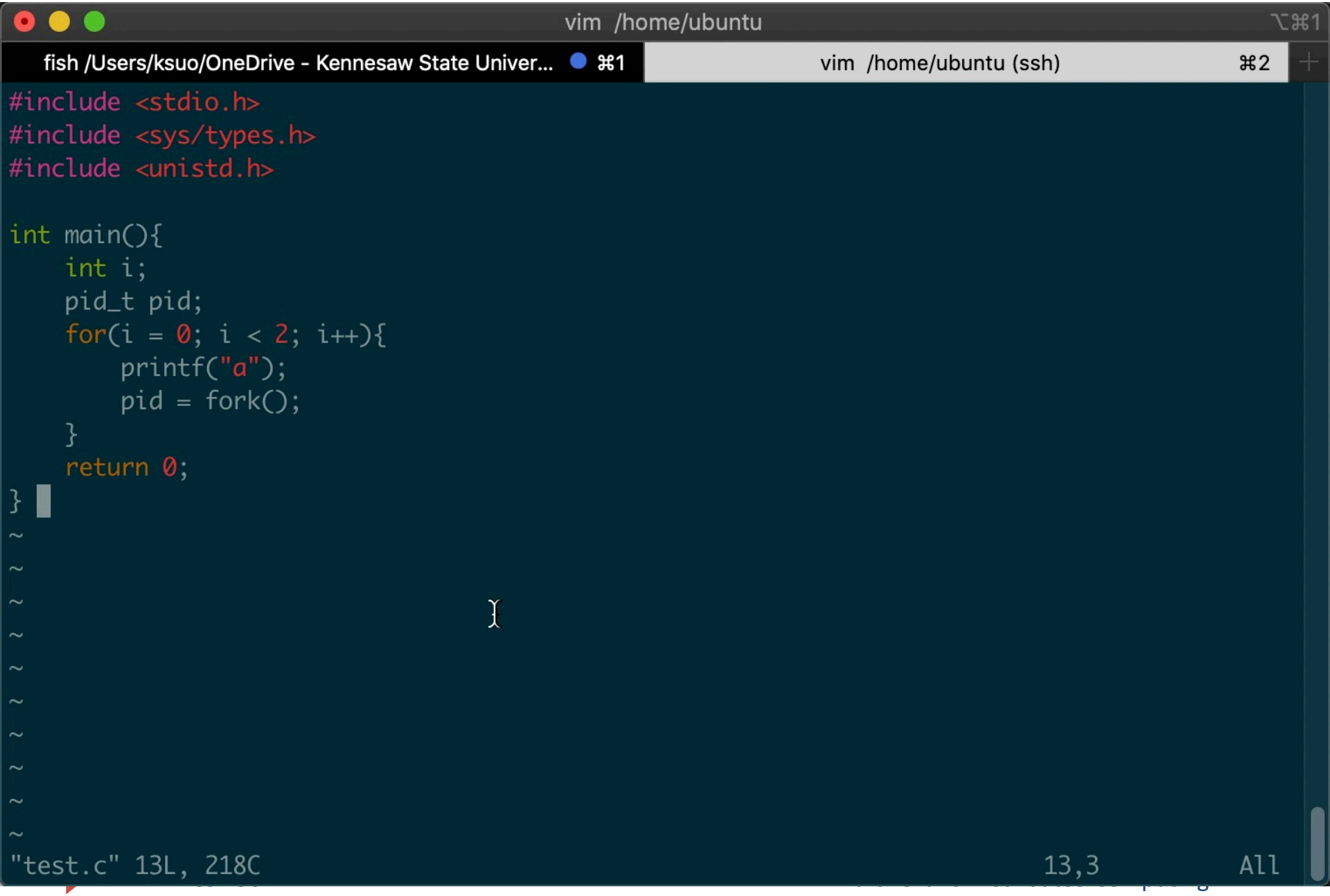
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a\n");
        pid = fork();
    }
    return 0;
}
```

i=0	Main:	a Create a process named 111
i=1	Main:	a Create a process named 222
	111:	a Create a process named 333

Fork() example

- How many a it will output?

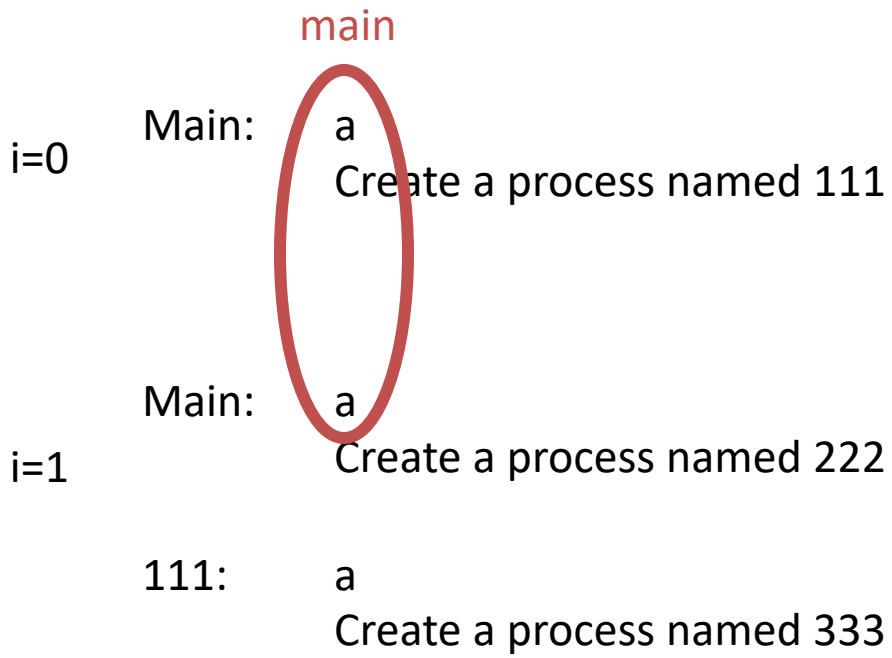
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a");
        pid = fork();
    }
    return 0;
}
```


vim /home/ubuntu
⌥⌘1
fish /Users/ksuo/OneDrive - Kennesaw State Univer... ⌘1
vim /home/ubuntu (ssh) ⌘2
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
int i;
pid_t pid;
for(i = 0; i < 2; i++){
printf("a");
pid = fork();
}
return 0;
}
"test.c" 13L, 218C
13,3
All

Fork() example

- Printf (without /n) will not flush the buffer until the program finished

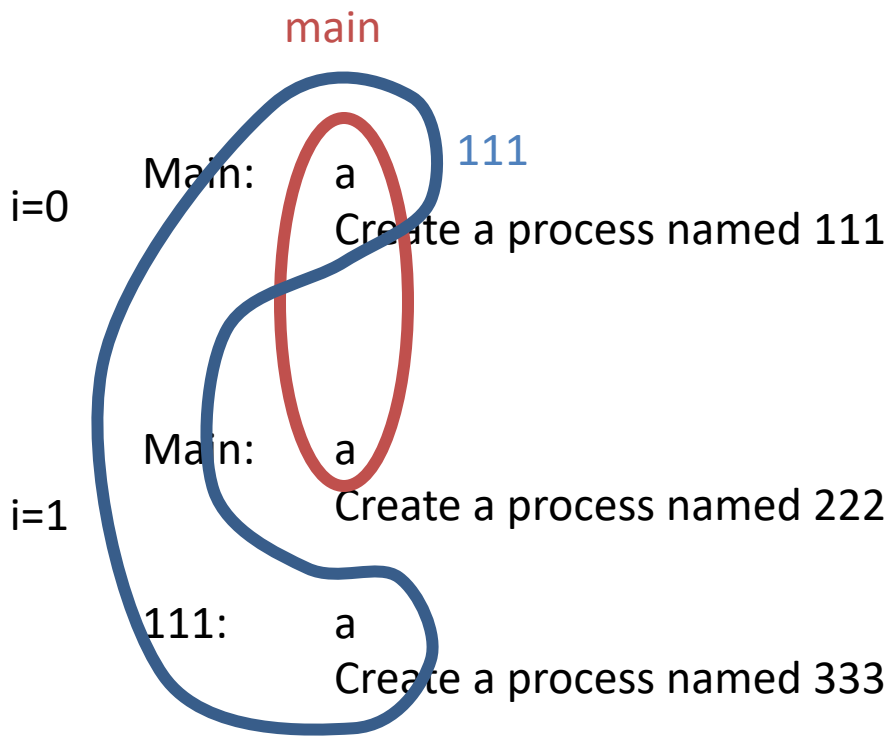
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a");
        pid = fork();
    }
    return 0;
}
```



Fork() example

- Printf (without /n) will not flush the buffer until the program finished

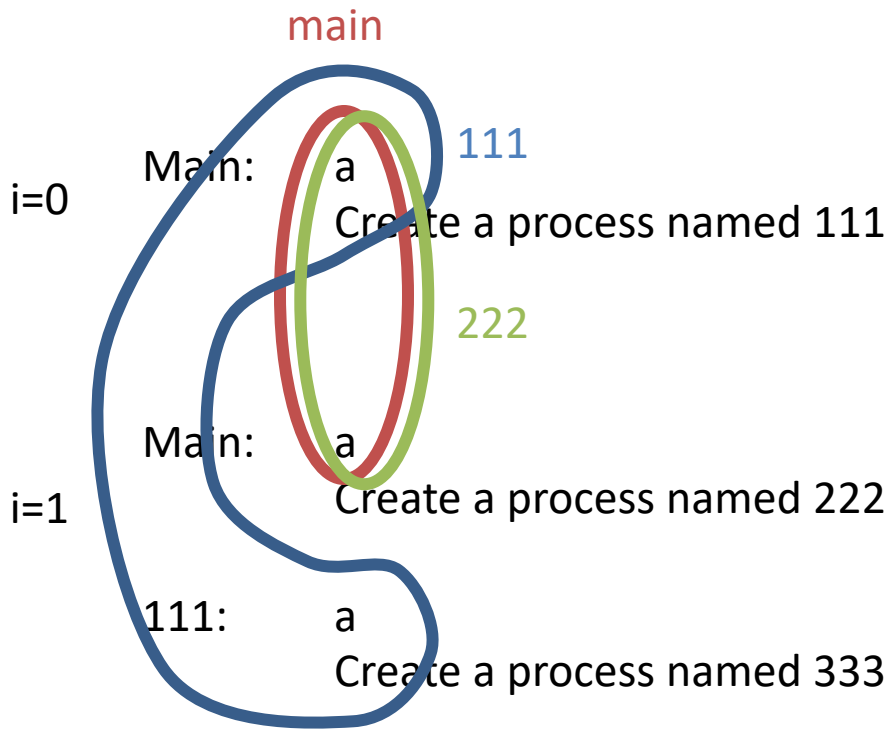
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a");
        pid = fork();
    }
    return 0;
}
```



Fork() example

- Printf (without /n) will not flush the buffer until the program finished

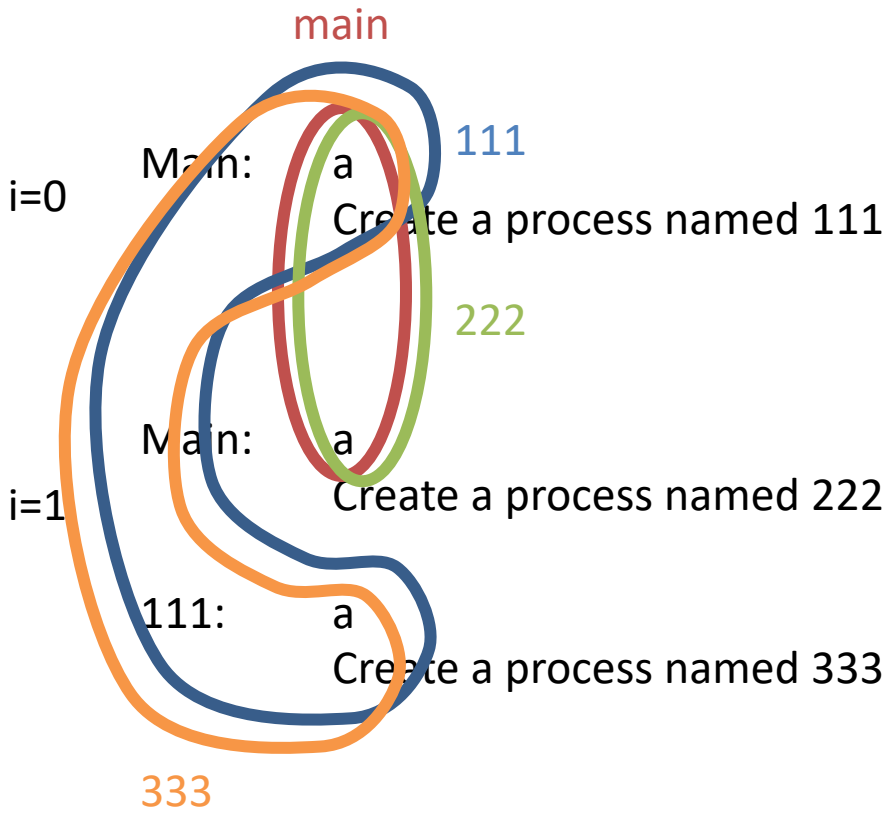
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a");
        pid = fork();
    }
    return 0;
}
```



Fork() example

- Printf (without /n) will not flush the buffer until the program finished

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int i;
    pid_t pid;
    for(i = 0; i < 2; i++){
        printf("a");
        pid = fork();
    }
    return 0;
}
```



A fork() bomb

- What does this code do?

```
while(1) {
        fork();
}
```

Agent smith

Test

Assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

```
main() {
    int i = 0;
    int rc = fork();
    i++;

    if (rc == 0) {
        rc = fork();
        i++;
    } else {
        i++;
    }
    printf("Hello!\n");
    printf("i is %d\n", i);
}
```

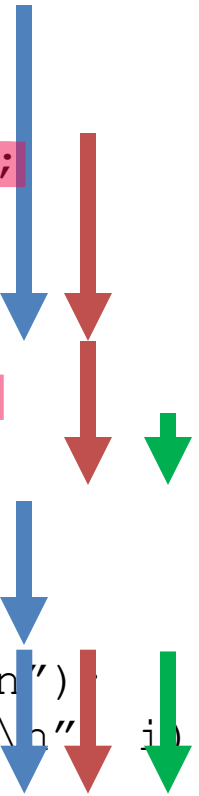
Assuming fork() never fails, how many times will the message "Hello!\n" be displayed? And explain your answer.

a) 2
b) 3
c) 4
d) 6
e) None of the above

Test

Assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

```
main() {
    int i = 0;
    int rc = fork();
    i++;

    if (rc == 0) {
        rc = fork();
        i++;
    } else {
        i++;
    }
    printf("Hello!\n");
    printf("i is %d\n", i);
}
```



Assuming fork() never fails, how many times will the message "Hello!\n" be displayed? And explain your answer.

a) 2
b) 3
c) 4
d) 6
e) None of the above

Test

Assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

What will be the largest value of "a" displayed by the program? And explain your answer.

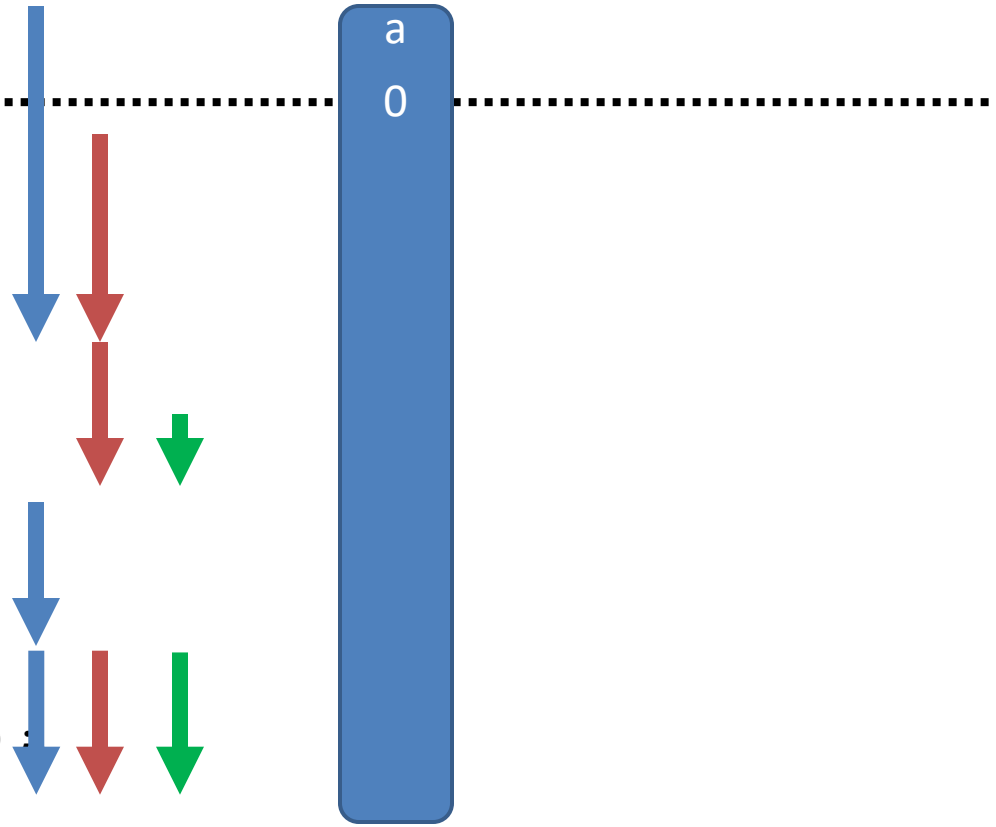
a) Due to race conditions, "a" may have different values on different runs of the program.
b) 2
c) 3
d) 5
e) None of the above

Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two … on a modern Linux machine (… irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

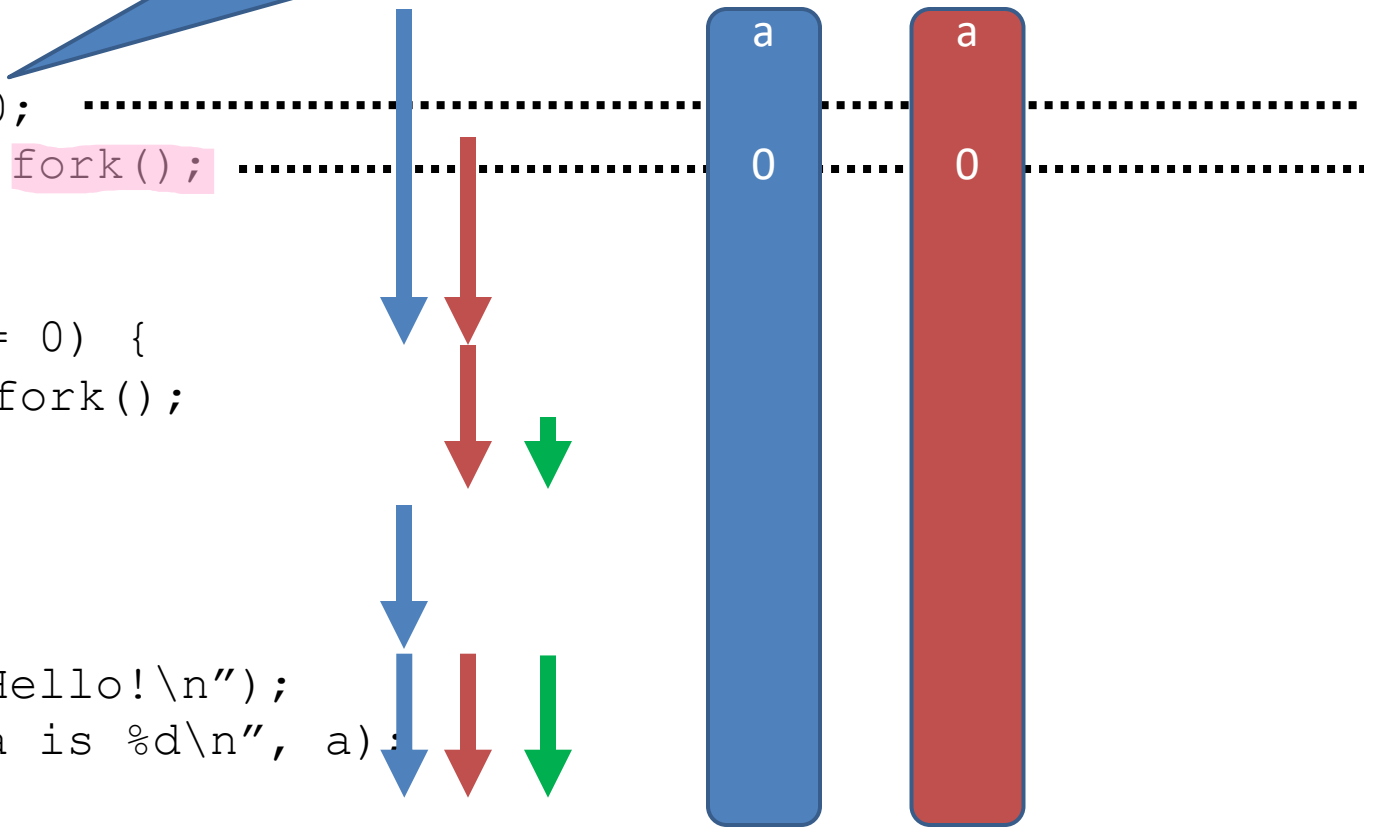


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two
on a modern Linux machine (irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

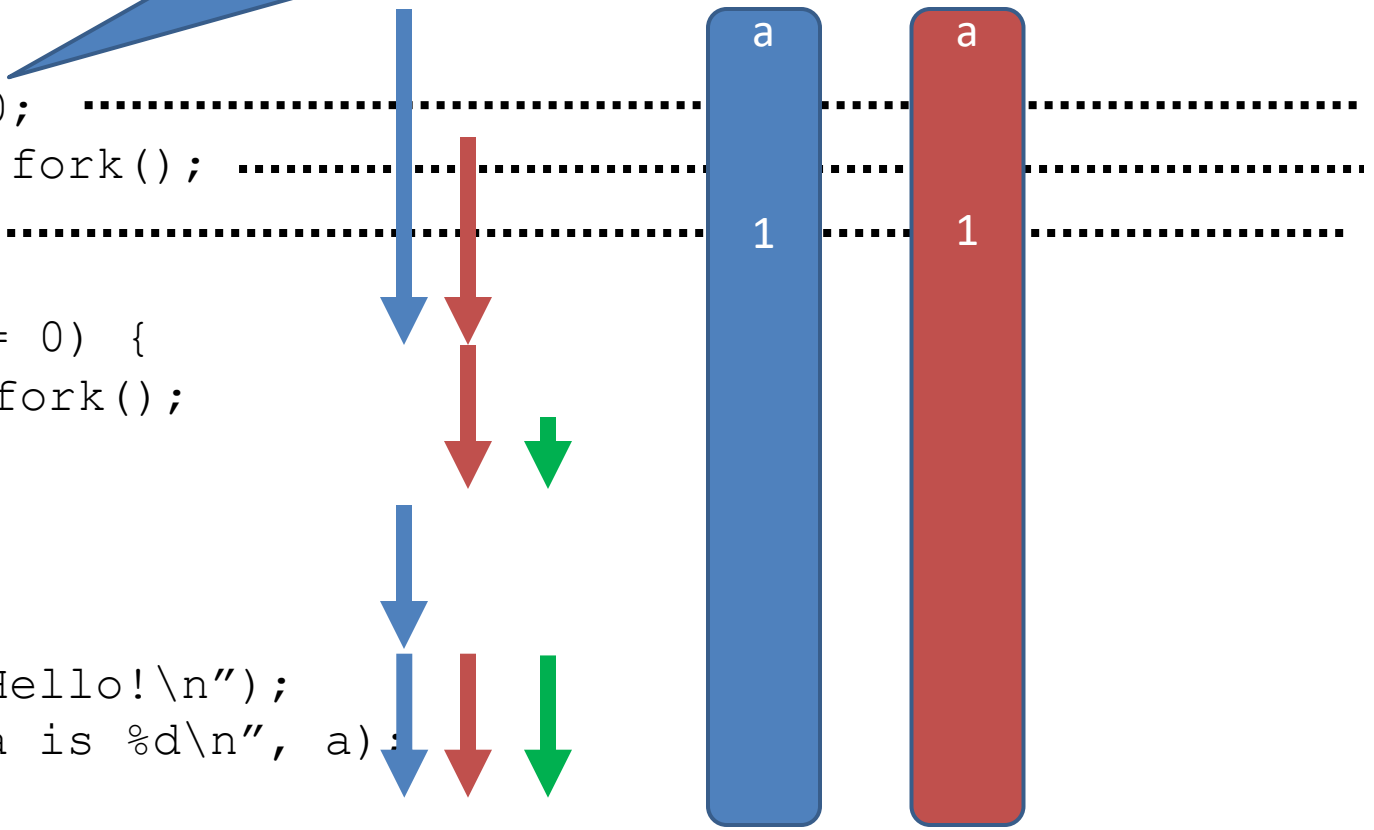


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two … on a modern Linux machine (… irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

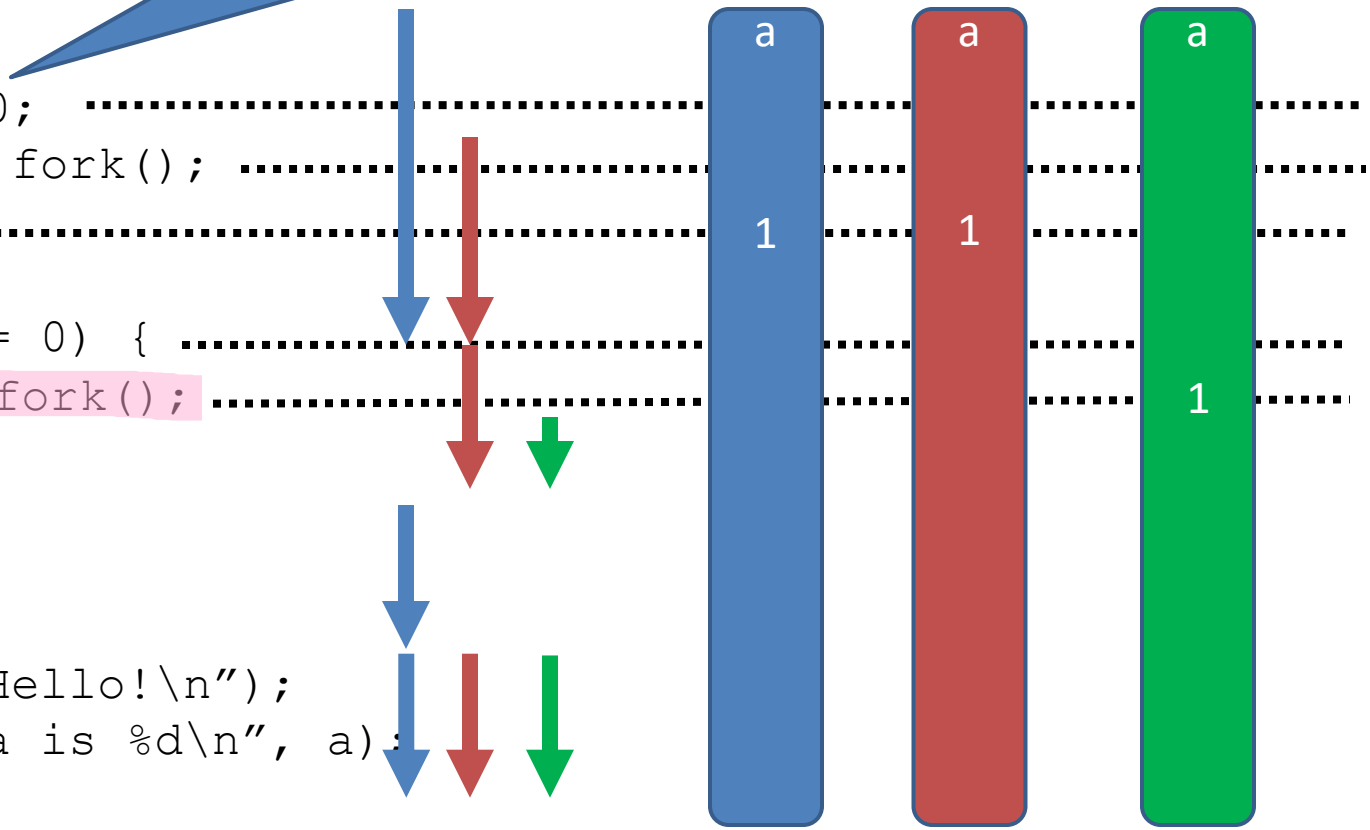


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two … on a modern Linux machine (… irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

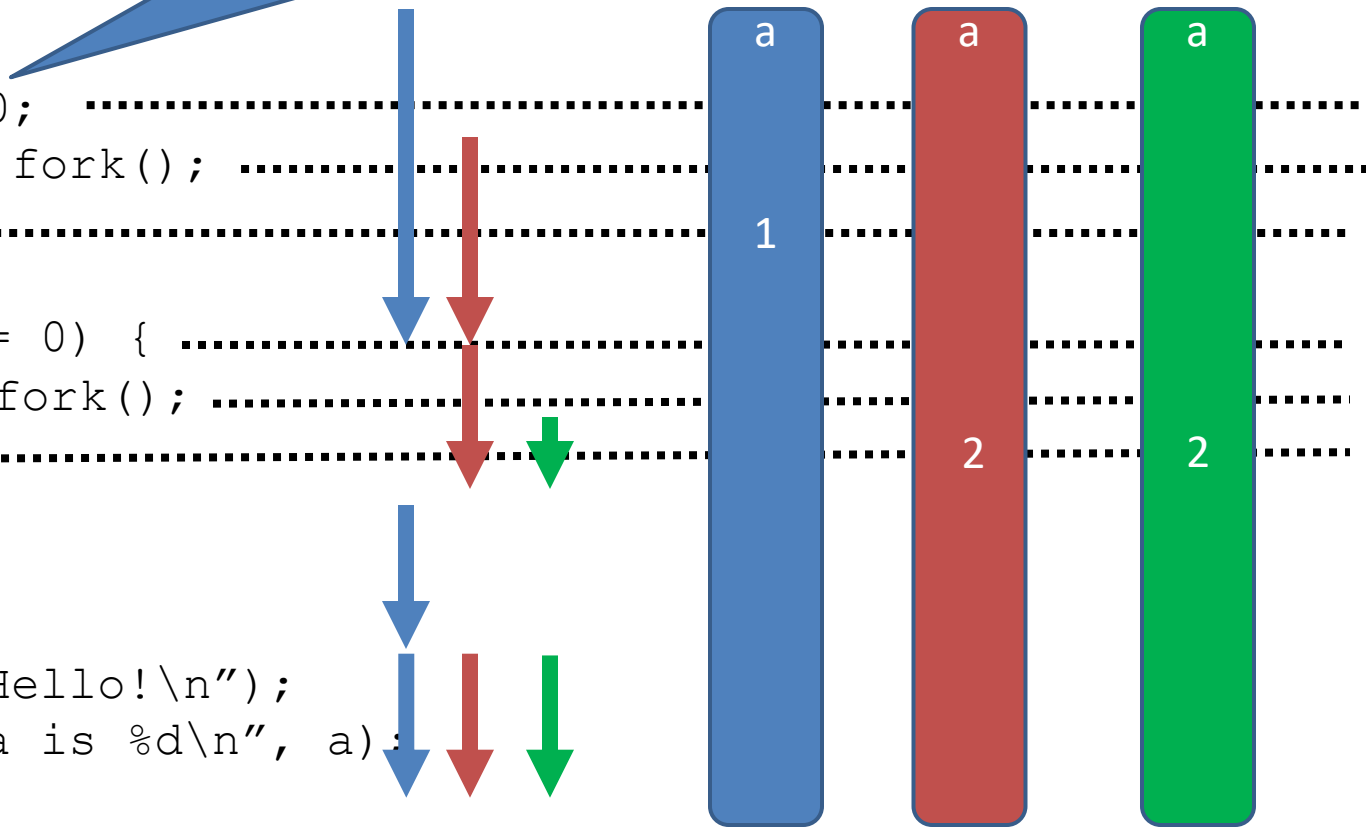


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two
on a modern Linux machine (irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

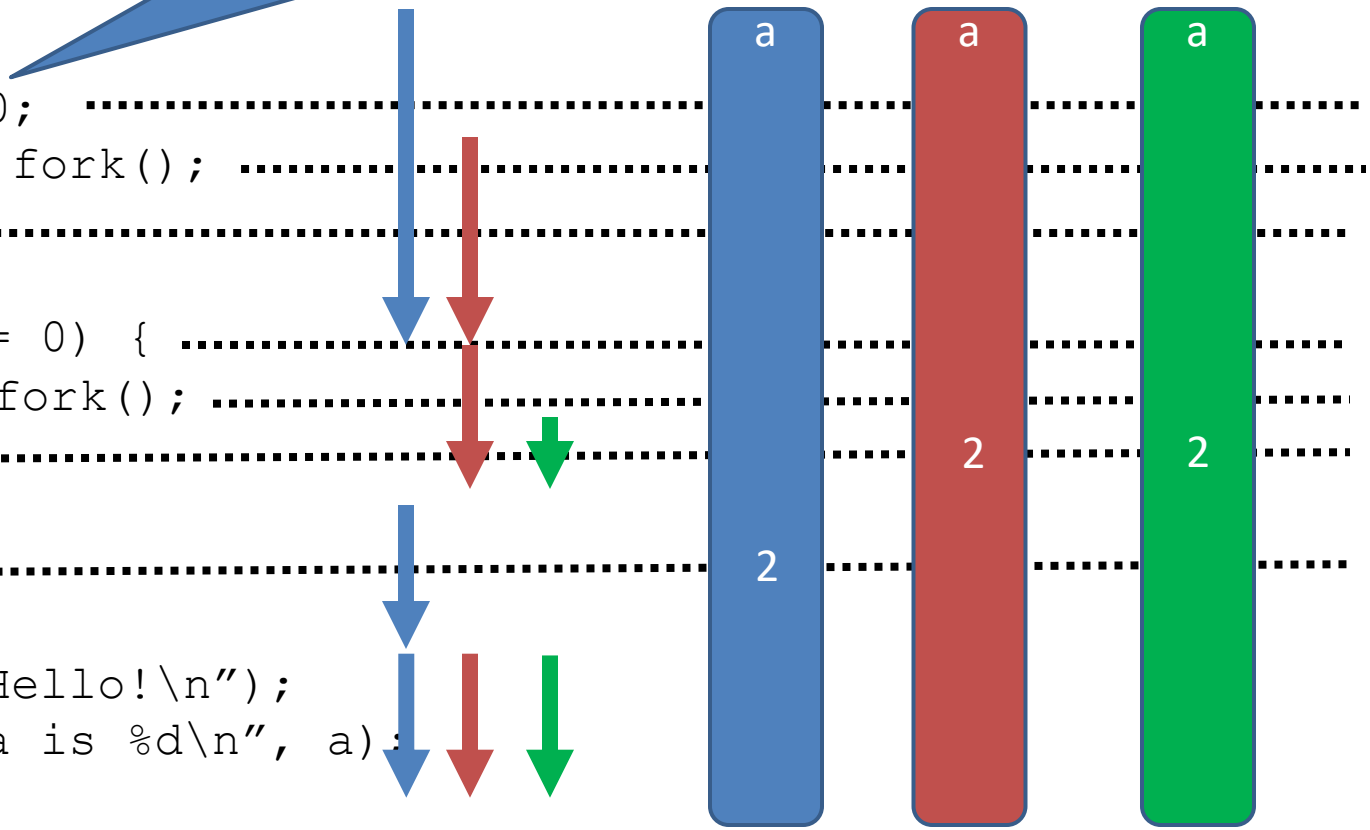


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two … on a modern Linux machine (… irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

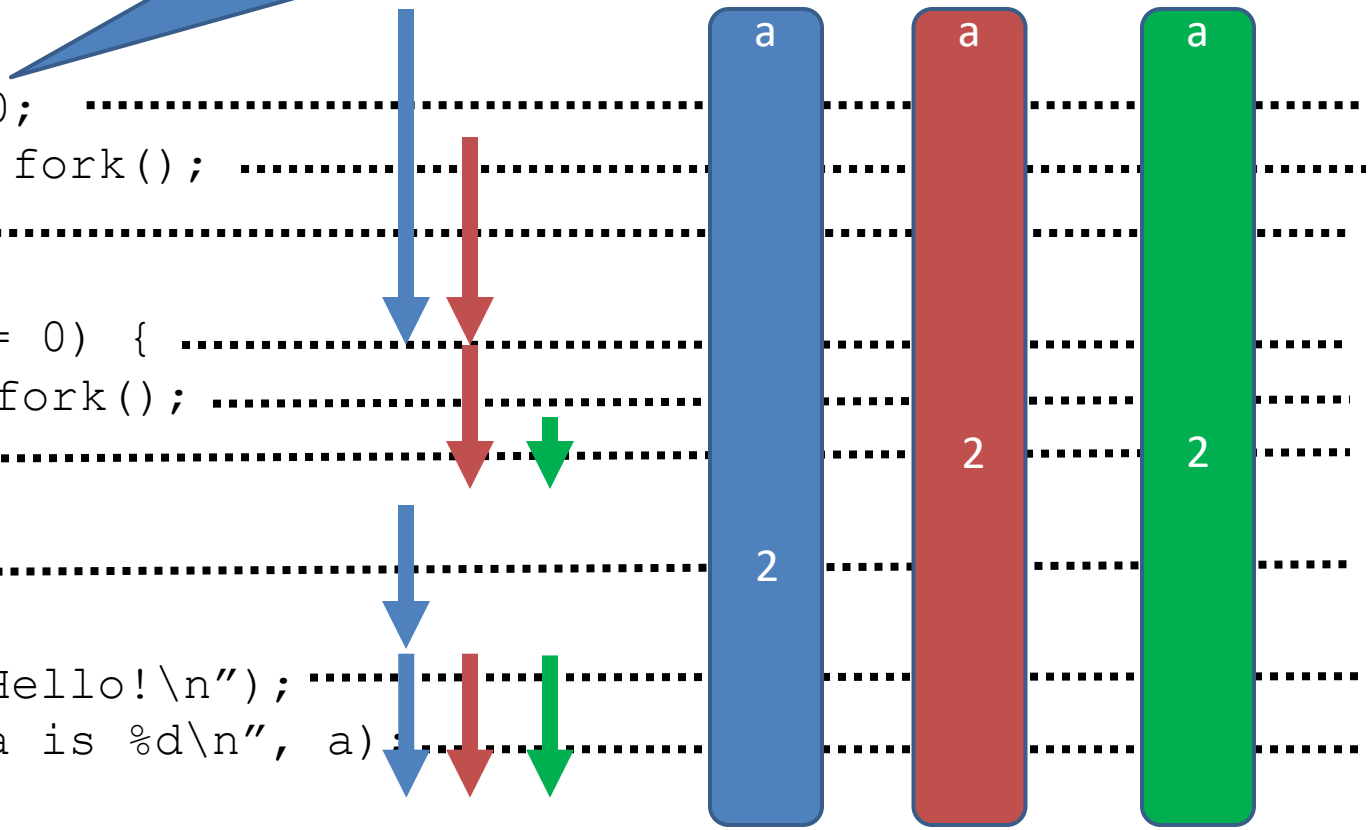


Test

Separate copies of a in each process:
Parent process: a = 0; a++; a++; → a=2.
Child process: a=0 (after fork()); a++; a++; → a=2.
Grandchild: a=1 (after fork()); a++; → a =2.

(2.1) For the next two … on a modern Linux machine (… irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```



Test

(2.1) For the next two questions, assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

```
main() {
    int a = 0;
    int rc = fork();
    a++;

    if (rc == 0) {
        rc = fork();
        a++;
    } else {
        a++;
    }
    printf("Hello!\n");
    printf("a is %d\n", a);
}
```

What will be the largest value of "a" displayed by the program? And explain your answer.

a) Due to race conditions, "a" may have different values on different runs of the program.

b) 2

c) 3

d) 5

e) None of the above

Exec() system call

- Replaces current process image with new program image.

- Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:
  - int execl(const char* path, const char* arg, …)
  - int execlp(const char* file, const char* arg, …)
  - int execle(const char* path, const char* arg, …, char* const envp[])
  - int execv(const char* path, const char* argv[])
  - int execvp(const char* file, const char* argv[])
  - int execvpe(const char* file, const char* argv[], char *const envp[])

Exec() system call

- Replaces current process image with new program image.

Wait() system call

https://github.com/kevinsuo/CS7172/blob/master/wait.c

- Helps the parent process
  - to know when a child completes
  - to check the return status of child

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
        pid_t cpid;
        if (fork()== 0) {
                printf("Child pid = %d\n", getpid());
        } else {
//              cpid = wait(NULL); /* returns a process ID of dead children */
                printf("Parent pid = %d\n", getpid());
        }

        return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./wait.o
Parent pid = 3425
Child pid = 3426
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
        pid_t cpid;
        if (fork()== 0) {
                printf("Child pid = %d\n", getpid());
        } else {
                cpid = wait(NULL); /* returns a process ID of dead children */
                printf("Parent pid = %d\n", getpid());
        }

        return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./wait.o
Child pid = 3395
Parent pid = 3394
```

Few other useful syscalls

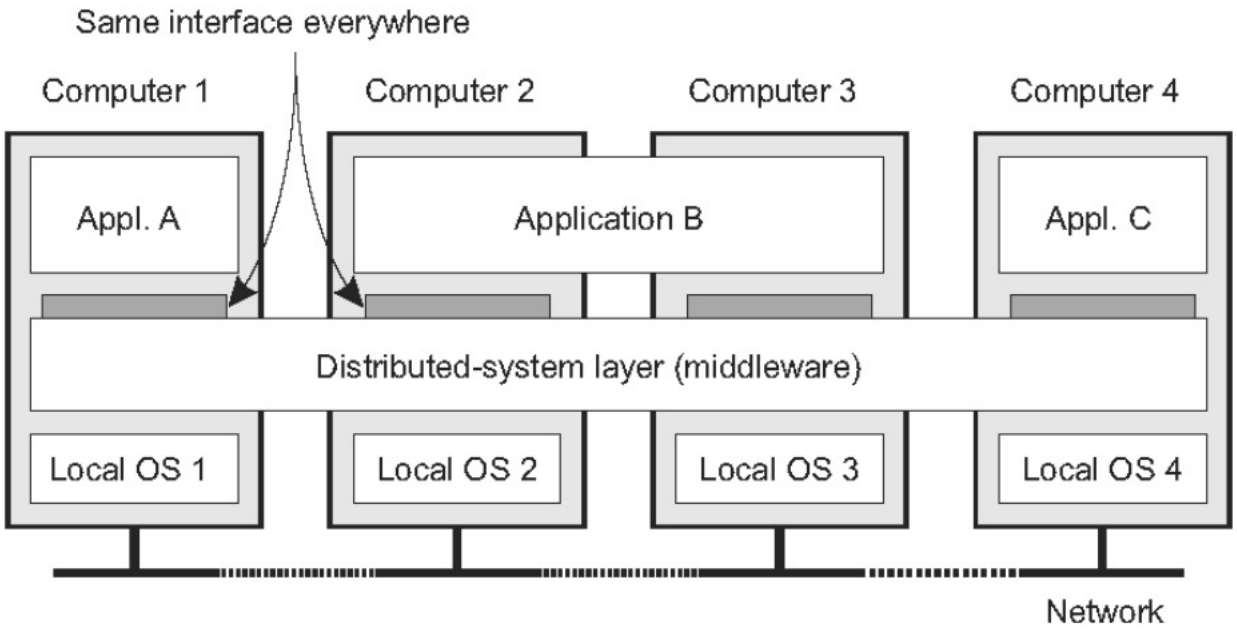
- sleep(seconds)
  - suspend execution for certain time
- exit(status)
  - Exit the program.
  - Status is retrieved by the parent using wait().
  - 0 for normal status, non-zero for error
- kill(pid_t pid, int sig)
  - Kill certain process

Outline

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Process on Distributed OSes

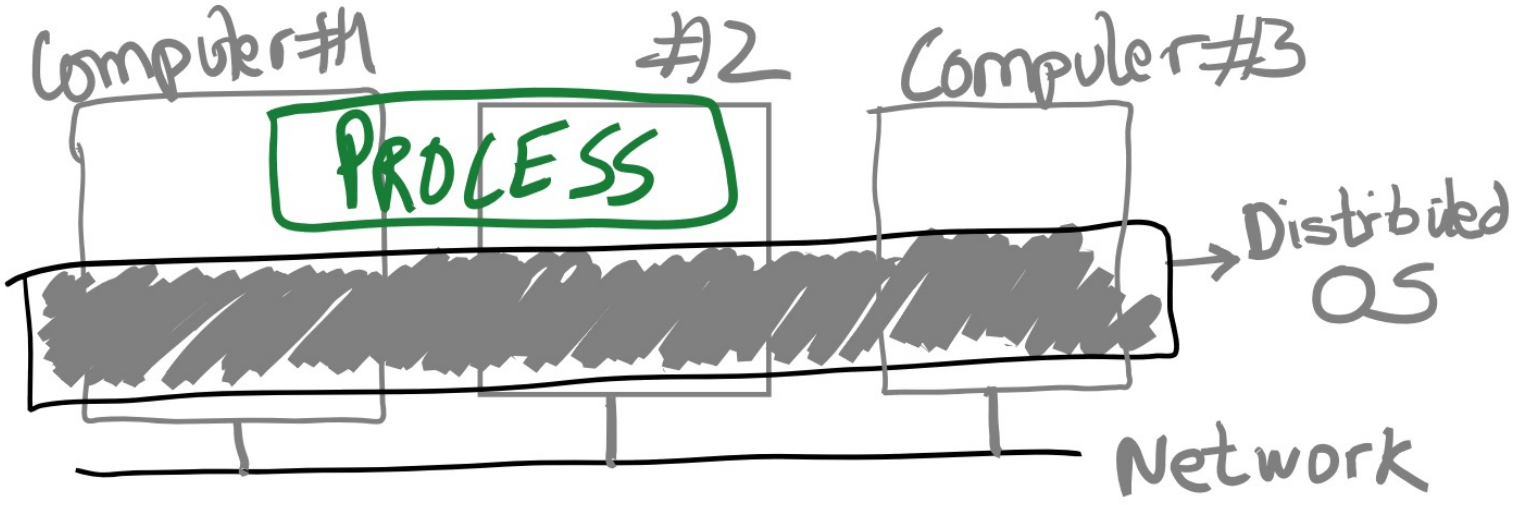
The OS of Distributed Systems

- Commonly used components and functions for distributed applications



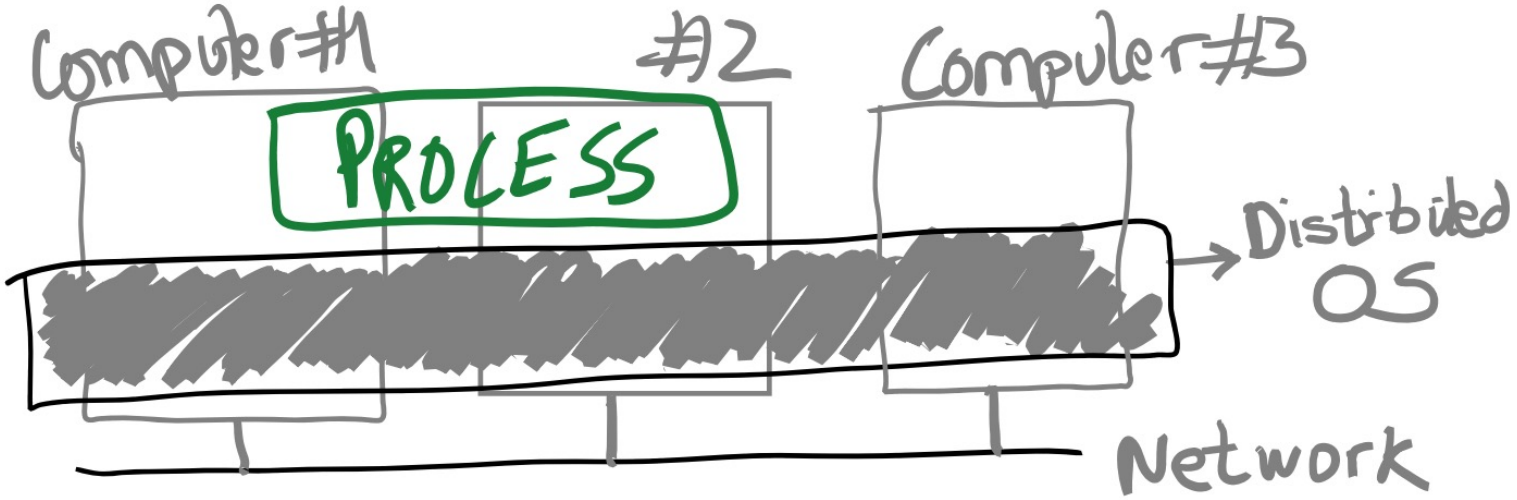
The OS of Distributed Systems

- An OS that spans multiple computers
- Same OS services, functionality, and abstractions as single-machine OS



Discussion

- What could be the challenges for distributed OS?

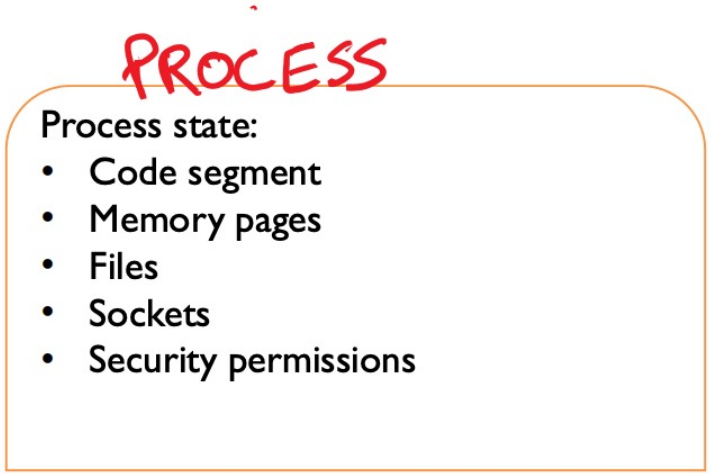


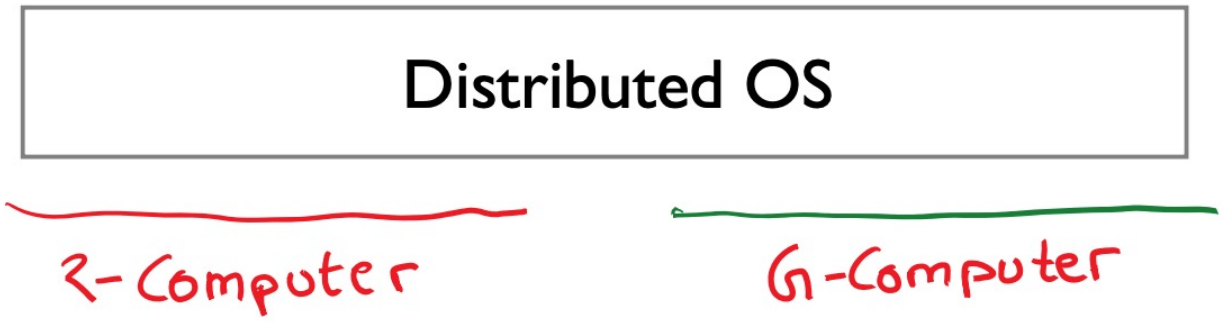
Distributed OS Challenges

- Providing the process **abstraction** and resource **virtualization** is hard
- Resource virtualization must be **transparent**
  - But in distributed settings, there's always a distinction between local and remote resources
- In a single-machine OS, processes don't care where their resources are coming from:
  - Which CPU cores, when they are scheduled, which physical memory pages they use, etc.
- In fact, providing abstract, virtual resources is one of the main OS services

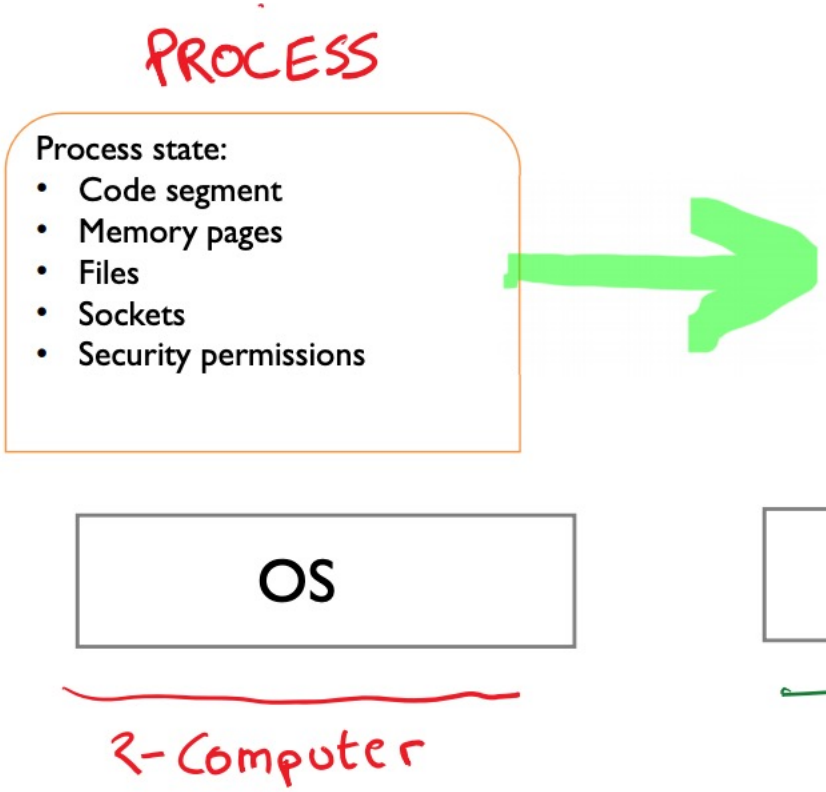
Transparency Issues In Distributed OS

- Where does code run?
- Which memory is used?
  Local vs. remote
- How are files accessed?



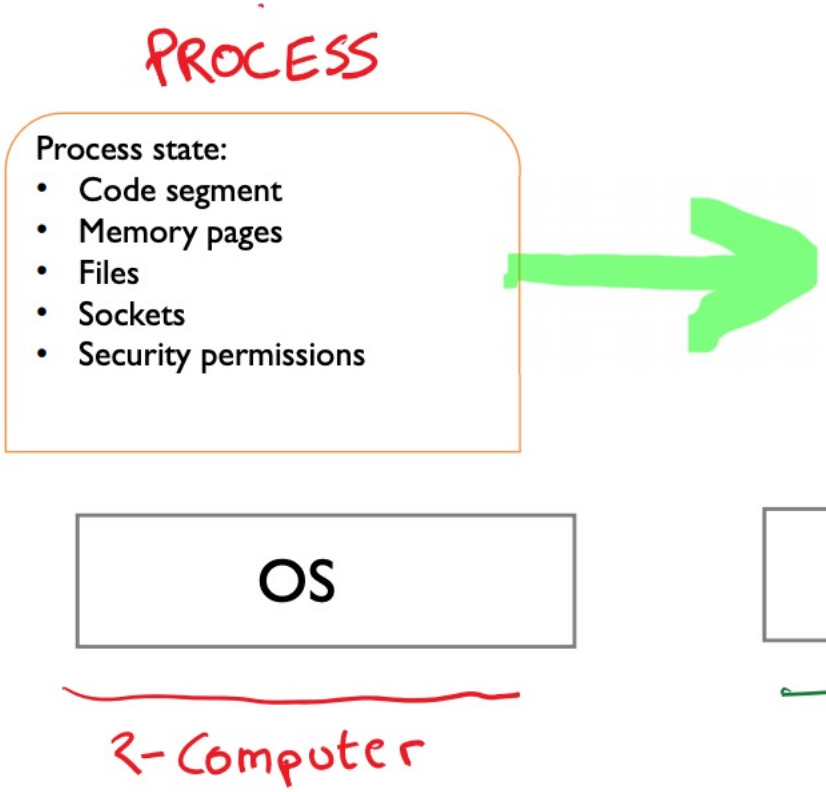


Process Migration



- Move all process state from one computer to another
- Process state can be vast
- Also entangled with other process states
  - ✓ Shared files?
  - ✓ IPC (pipes etc)

Process Migration



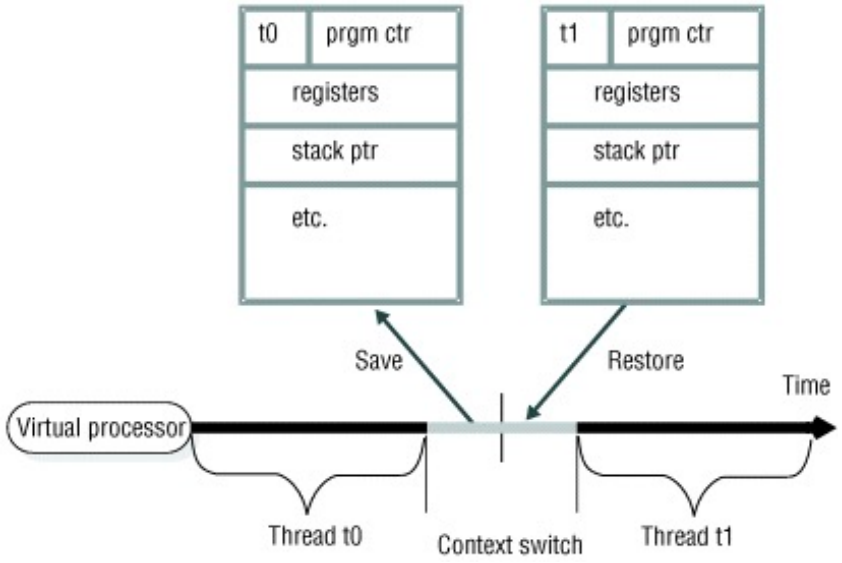
- Migrate some state? Other state, if required, is accessed over the network?
- Example: migrate only fraction of pages. Other pages are copied over the network on access?
- Access remote hardware devices (GPUs)?

Conclusion

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork, etc.
- Process on Distributed OSes

Context Switch

- A context switch is a procedure that a computer's CPU follows to change from one task (or process) to another while ensuring that the tasks do not conflict.

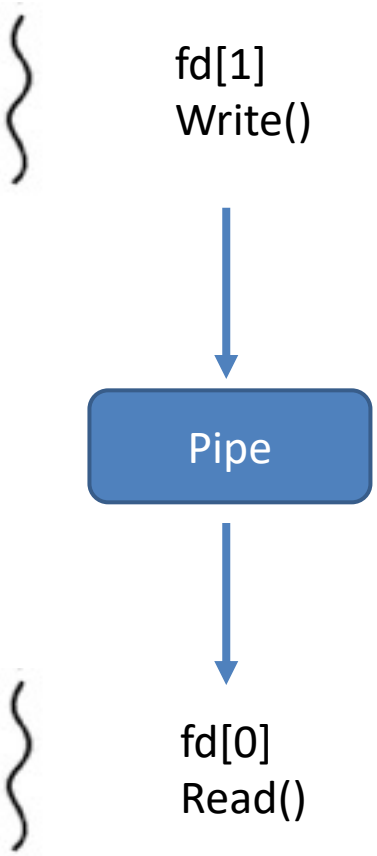


Context Switch

https://github.com/kevinsuo/CS7172/blob/master/cs.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <sched.h>
#include <sys/types.h>
#include <unistd.h>      //pipe()

int main()
{
    int x, i, fd[2], p[2];
    char send    = 's';
    char receive;
    pipe(fd);
    pipe(p);
    struct timeval tv;
    struct sched_param param;
    param.sched_priority = 0;

    while ((x = fork()) == -1);
    if (x==0) {
        sched_setscheduler(getpid(), SCHED_FIFO, &param);
        gettimeofday(&tv, NULL);
        printf("Before Context Switch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
        for (i = 0; i < 10000; i++) {
            read(fd[0], &receive, 1);
            write(p[1], &send, 1);
        }
        exit(0);
    }
    else {
        sched_setscheduler(getpid(), SCHED_FIFO, &param);
        for (i = 0; i < 10000; i++) {
            write(fd[1], &send, 1);
            read(p[0], &receive, 1);
        }
        gettimeofday(&tv, NULL);
        printf("After Context SWitch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
    }
    return 0;
}
```

When A writes pipe, B will be blocked



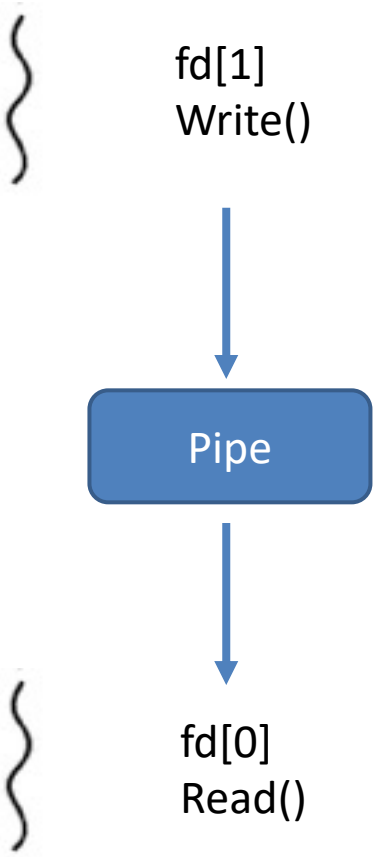
When B reads pipe, A will be blocked

Context Switch

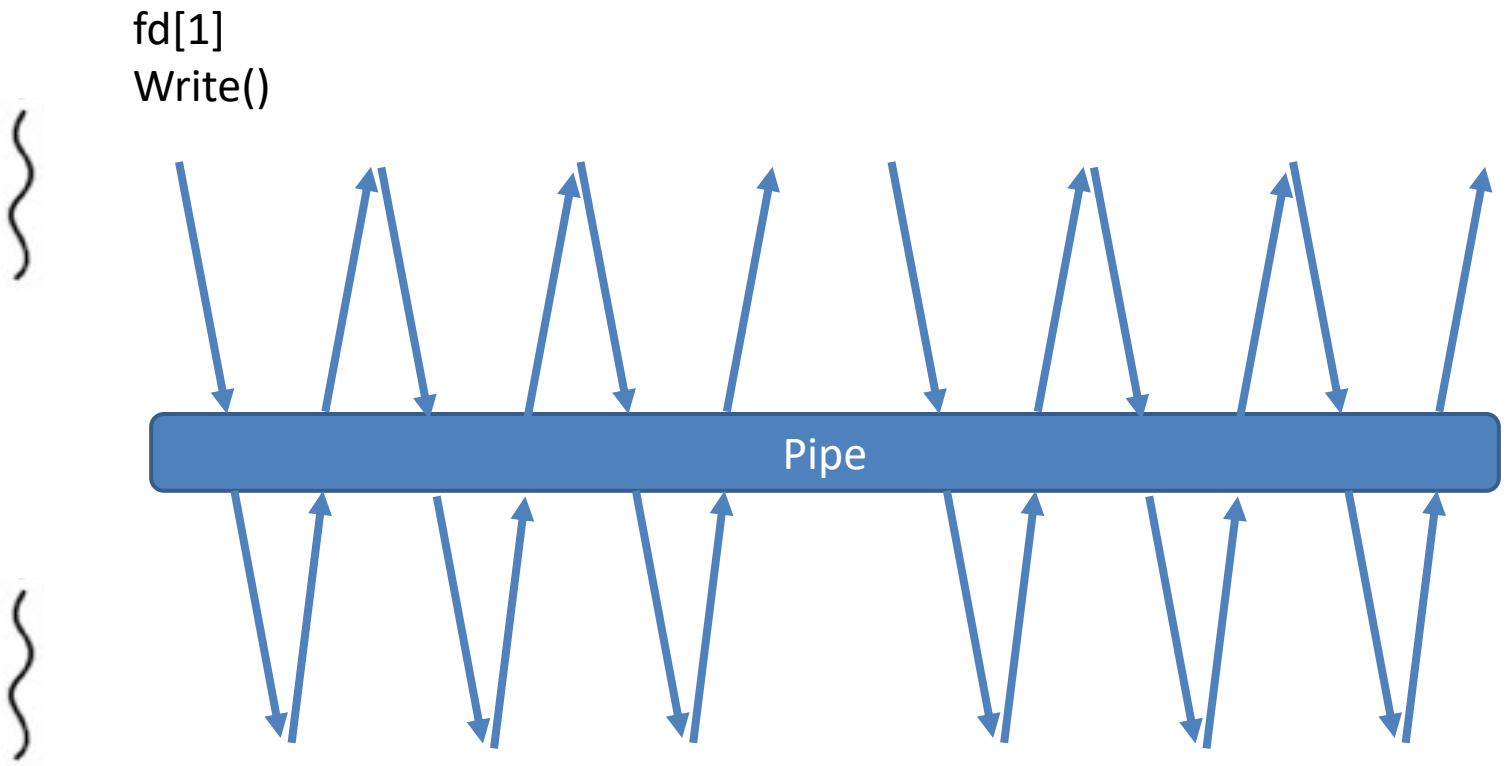
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <sched.h>
#include <sys/types.h>
#include <unistd.h>      //pipe()

int main()
{
    int x, i, fd[2], p[2];
    char send    = 's';
    char receive;
    pipe(fd);
    pipe(p);
    struct timeval tv;
    struct sched_param param;
    param.sched_priority = 0;

    while ((x = fork()) == -1);
    if (x==0) {
        sched_setscheduler(getpid(), SCHED_FIFO, &param);
        gettimeofday(&tv, NULL);
        printf("Before Context Switch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
        for (i = 0; i < 10000; i++) {
            read(fd[0], &receive, 1);
            write(p[1], &send, 1);
        }
        exit(0);
    }
    else {
        sched_setscheduler(getpid(), SCHED_FIFO, &param);
        for (i = 0; i < 10000; i++) {
            write(fd[1], &send, 1);
            read(p[0], &receive, 1);
        }
        gettimeofday(&tv, NULL);
        printf("After Context SWitch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
    }
    return 0;
}
```

Run 10k times of read and write

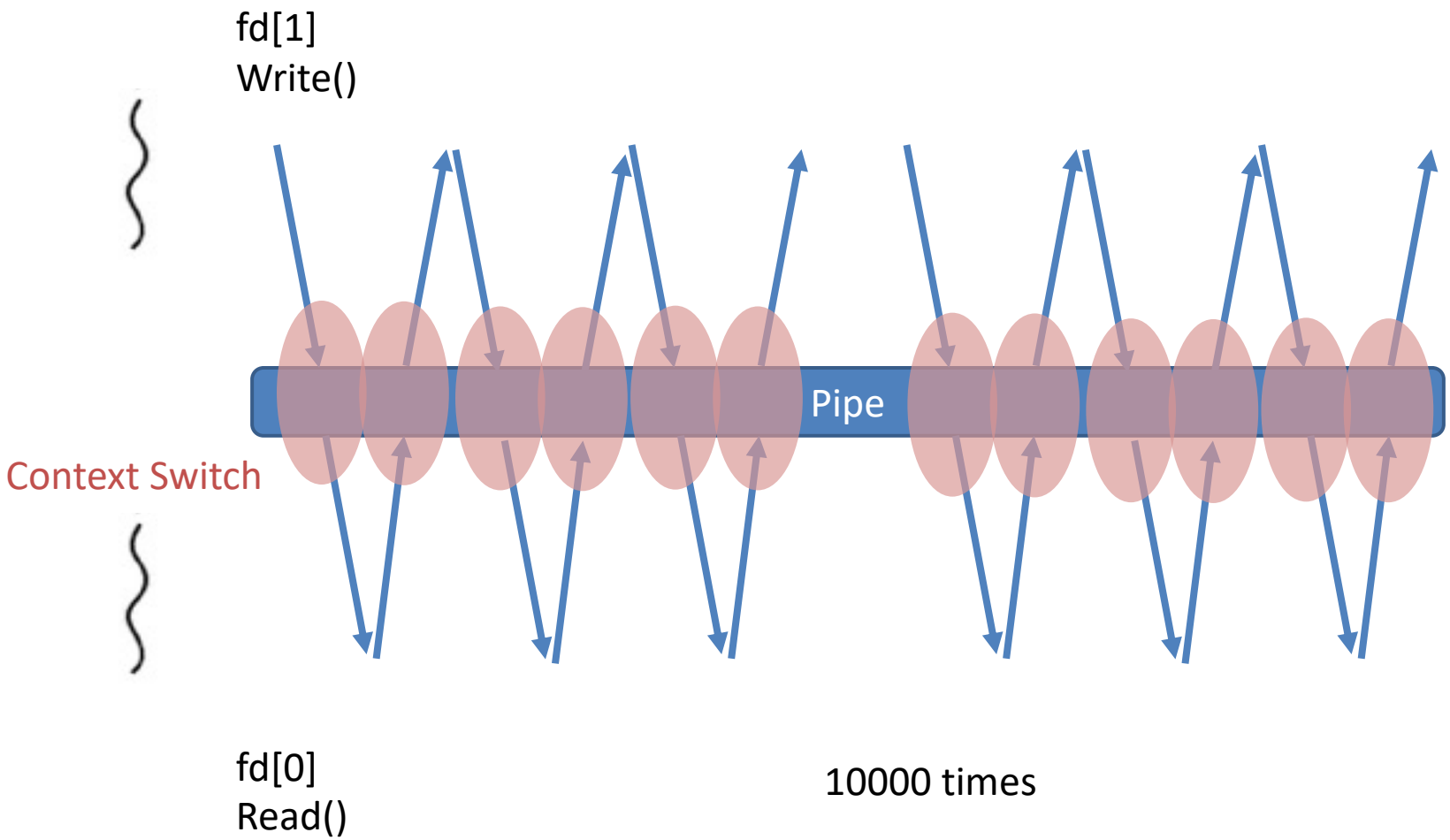
Run 10k times of read and write



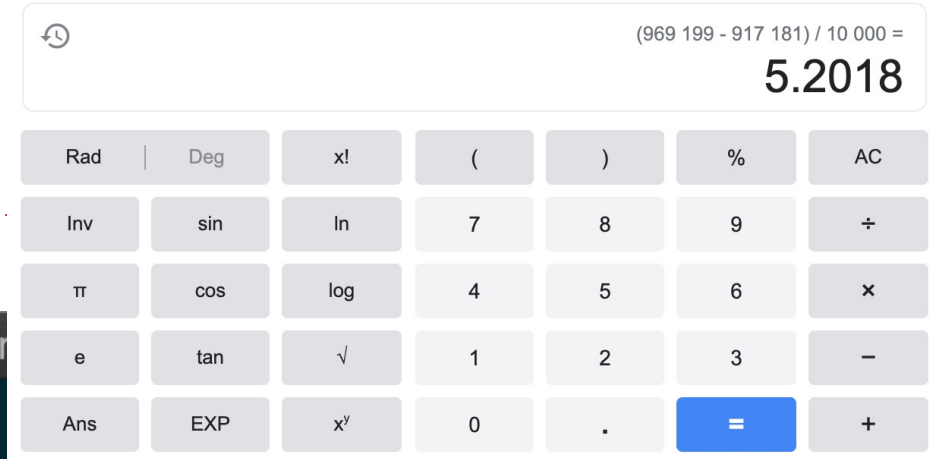
Context Switch

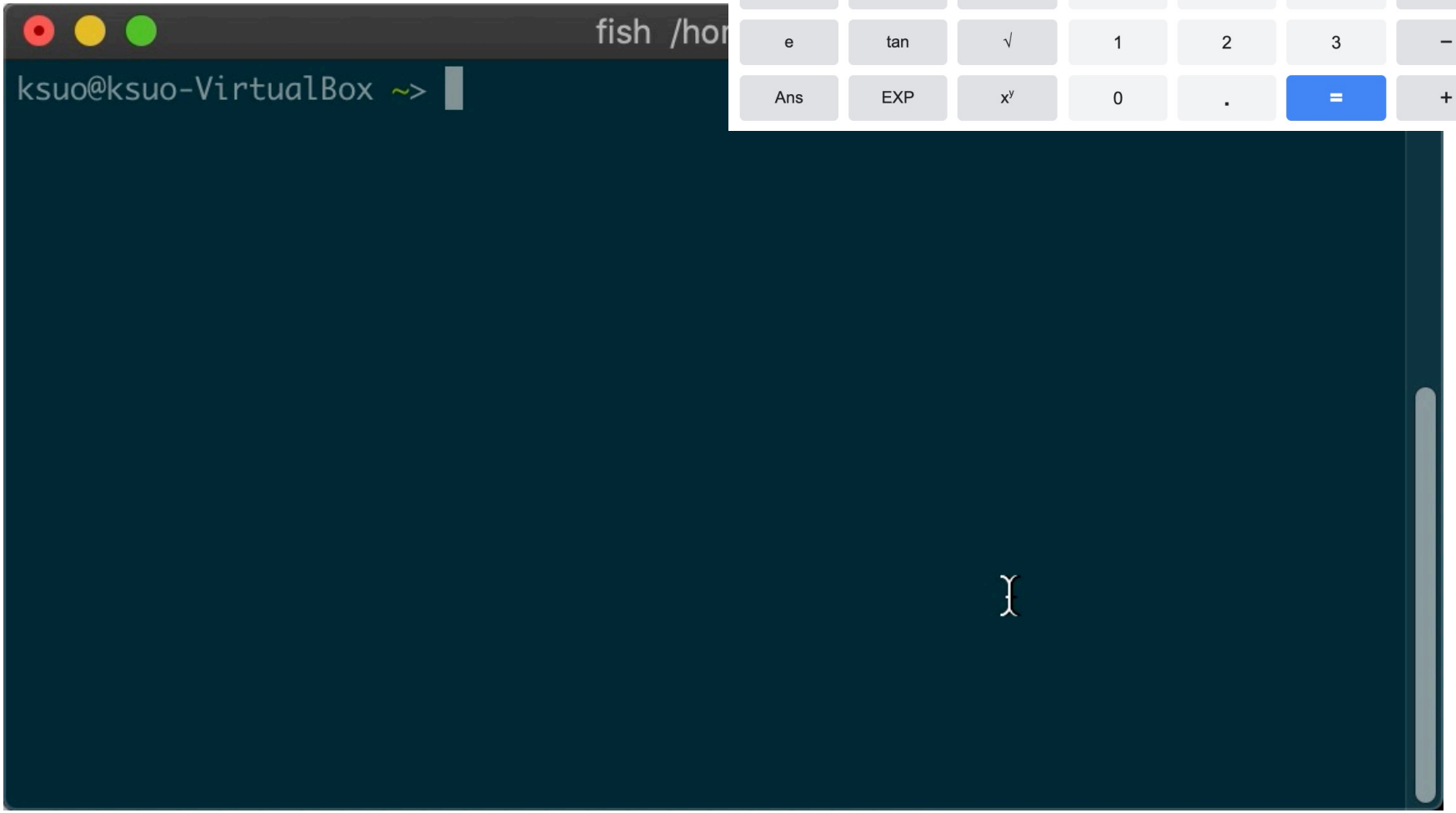


Context Switch



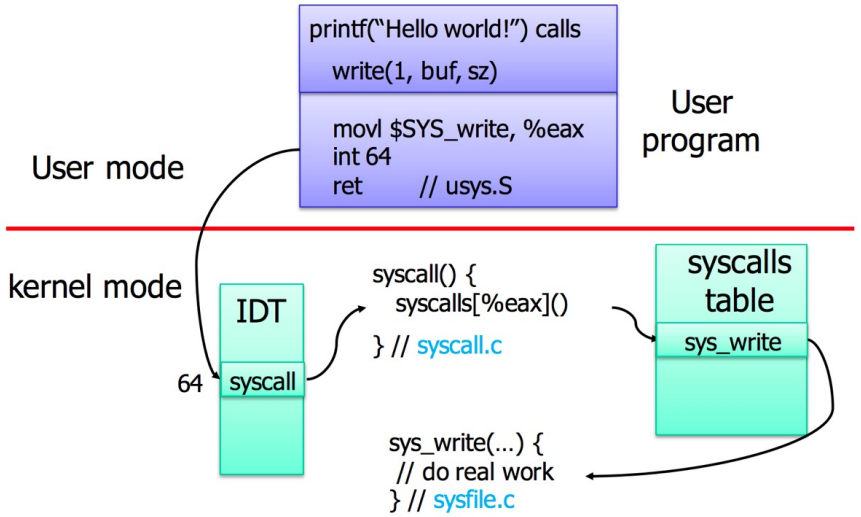
Context Switch



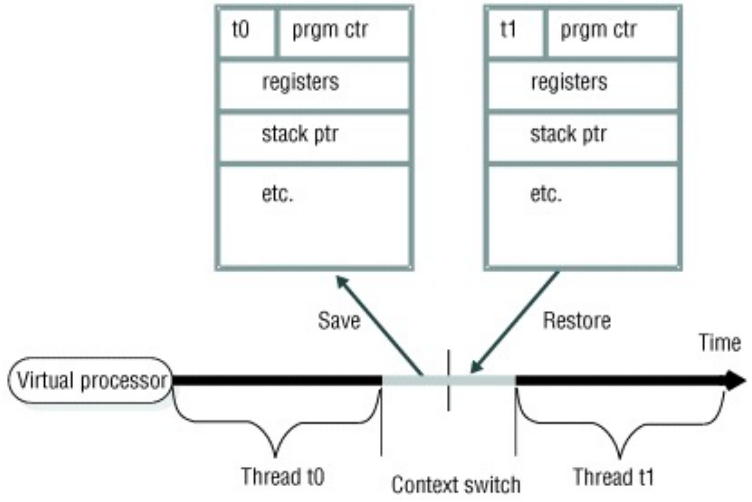


Context Switch

- The general system call time consumption is around 200ns

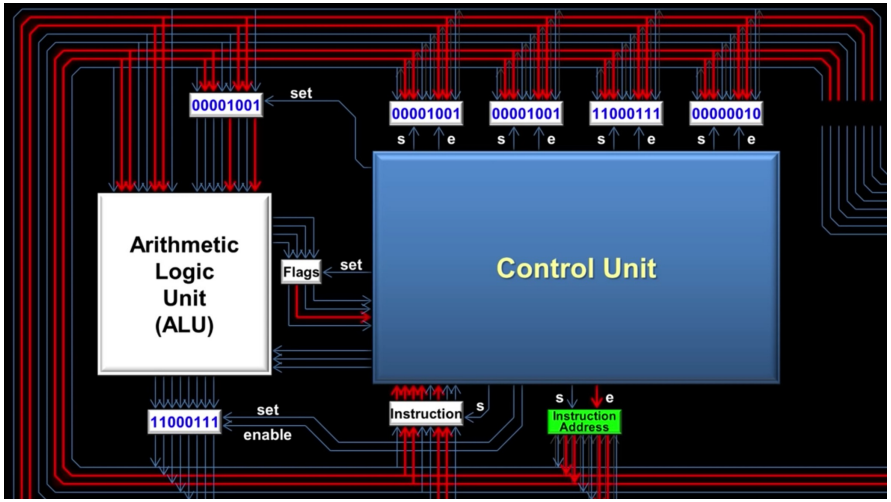


- Context switch is 5us >> 200ns



The Cost of Context Switch

- Direct cost
  - Context switch on memory pages
  - Context switch on kernel heap and stack
  - Save data into registers, ip(instruction pointer), bp(base pointer), sp(stack poinger)
  - Flush TLB
  - Code execution in kernel for scheduling
  - …



The Cost of Context Switch

- Indirect cost
  - Cache miss
  - Reload code and data into L1, L2, L3 cache