CS 3502
Operating Systems

Process

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Outline

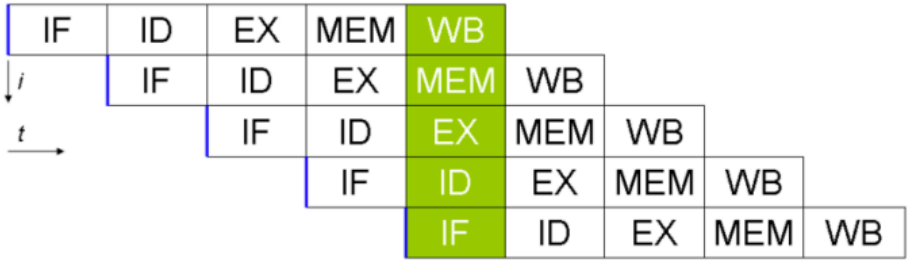
- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Orphan and Zombie process

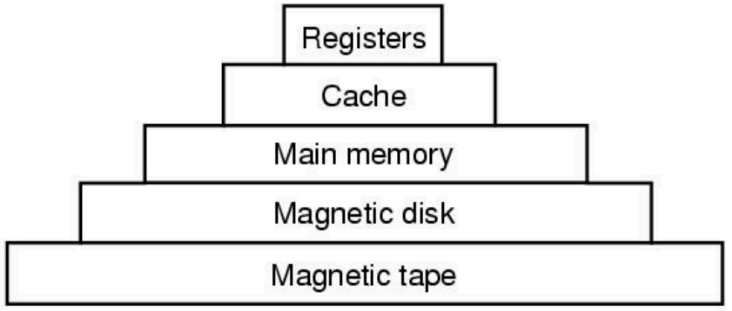
Process

- 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 …



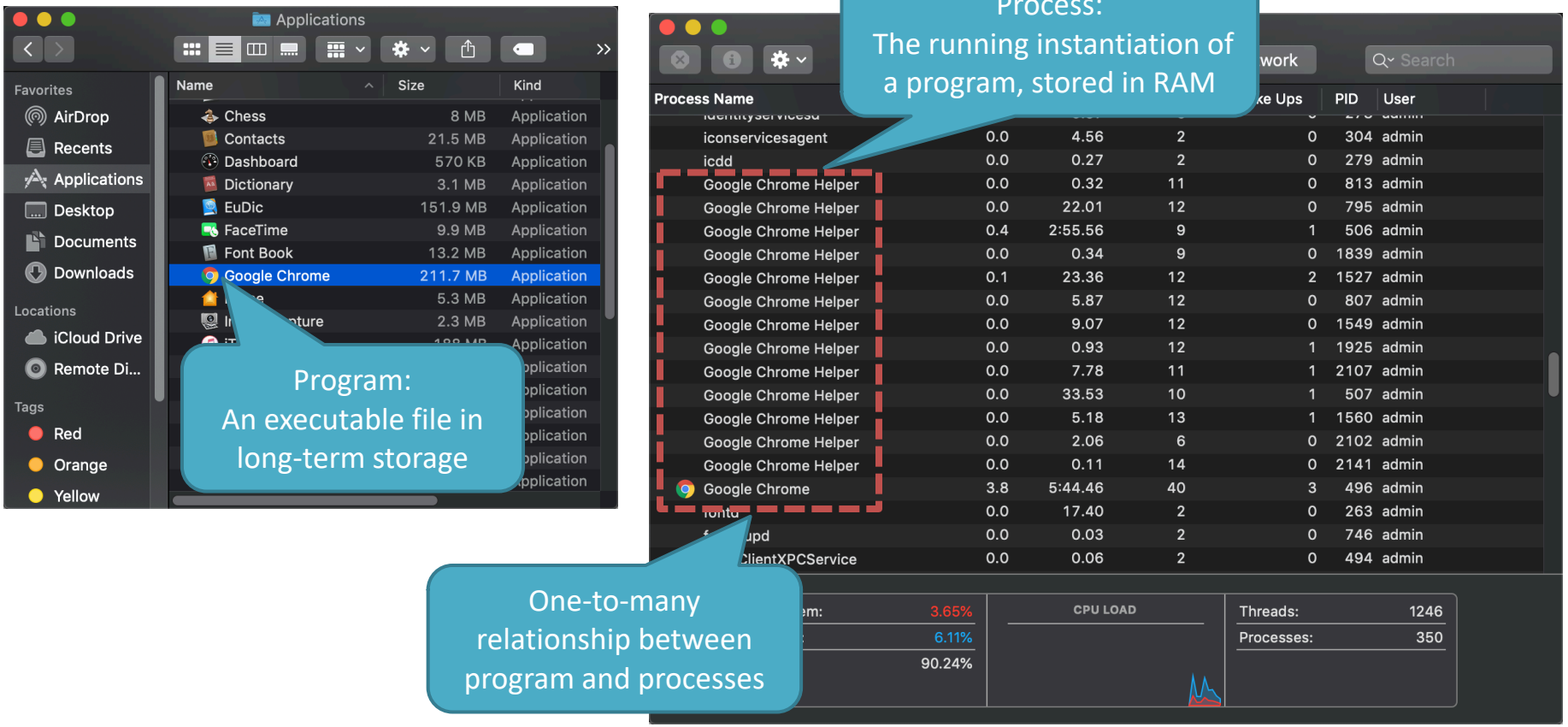


Program vs. Process

- 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

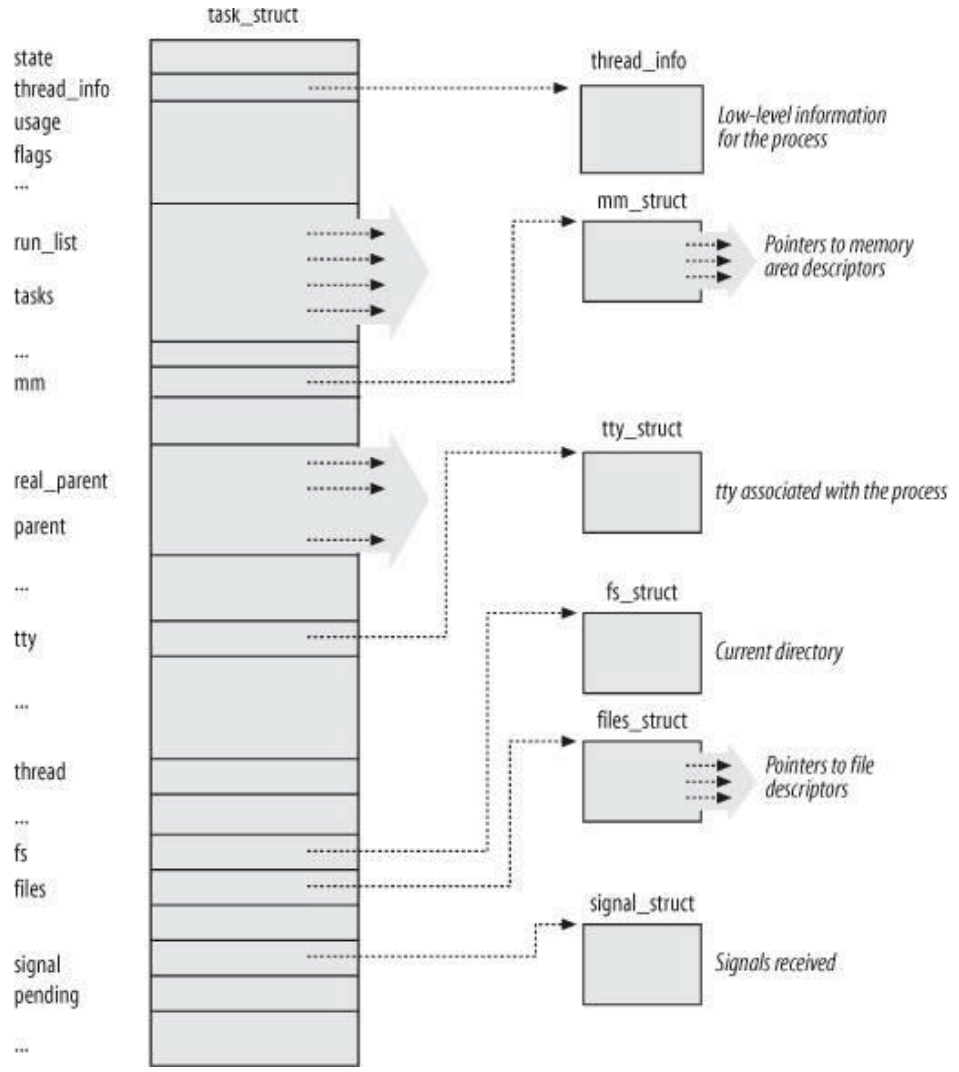
Process Descriptor

- 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/latest/source/include/linux/sched.h#L584

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

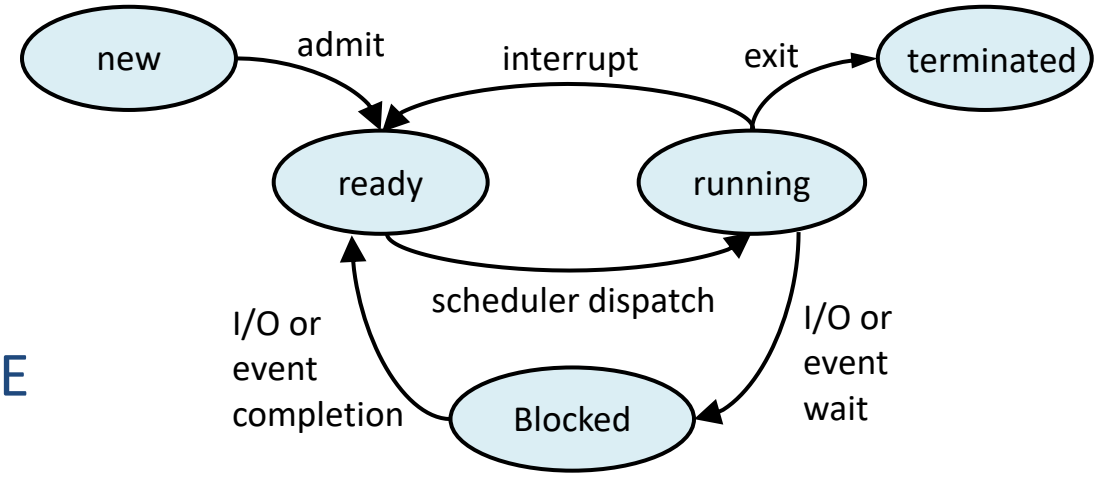


Process in Linux

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

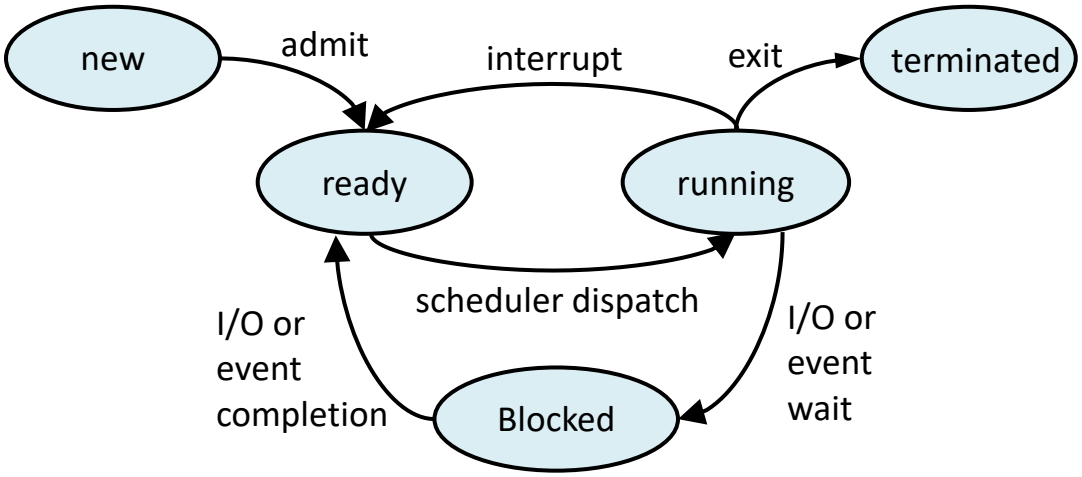
Linux PCB (5-state model to describe lifecycle of one process)

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



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

- What is difference between blocked and ready?

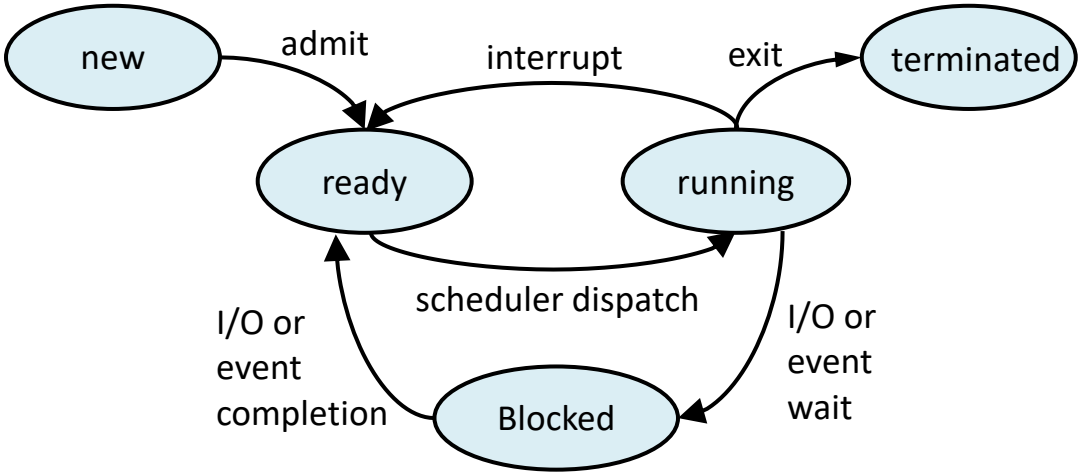


Blocked: unable to run, wait for an event/data

Ready: able and willing to run, wait for the CPU

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

- State
  - TASK_RUNNING
    - Running, ready



- How to differentiate the running and ready state?

Running: on the CPU, not in the queues
Ready: off the CPU, in the queues

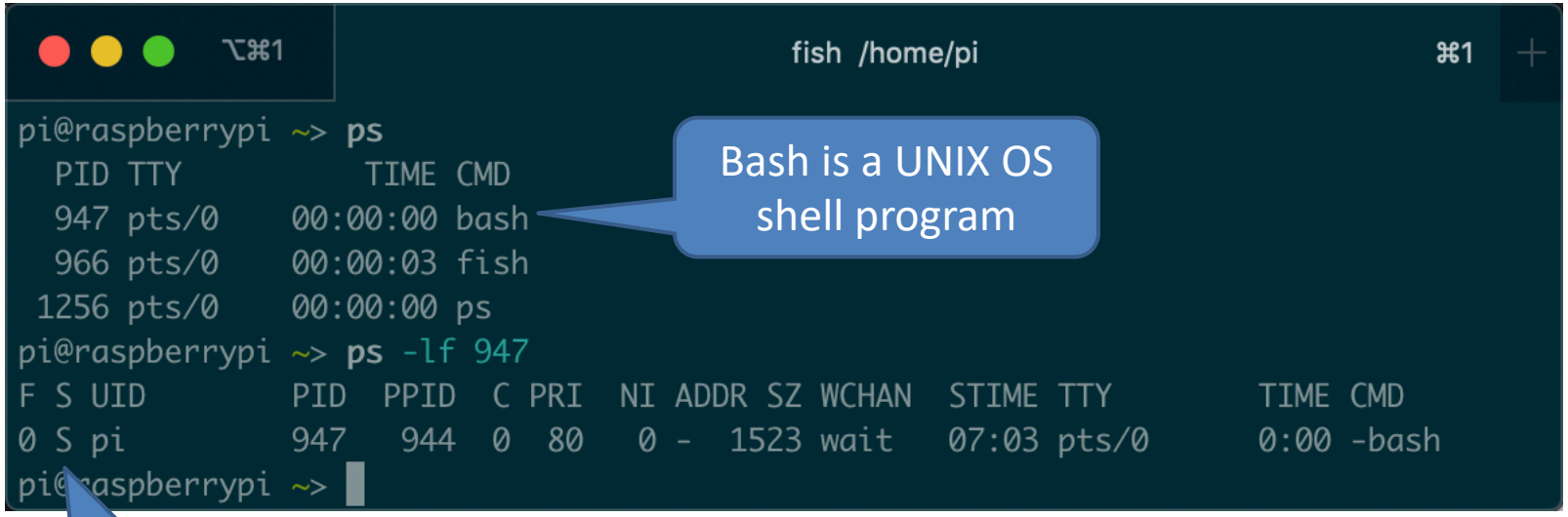
Linux Process Control Block (cont')

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

Take Linux 4.2 as an example

https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1345

$ 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

Process in Linux

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

Linux Process Control Block (cont')

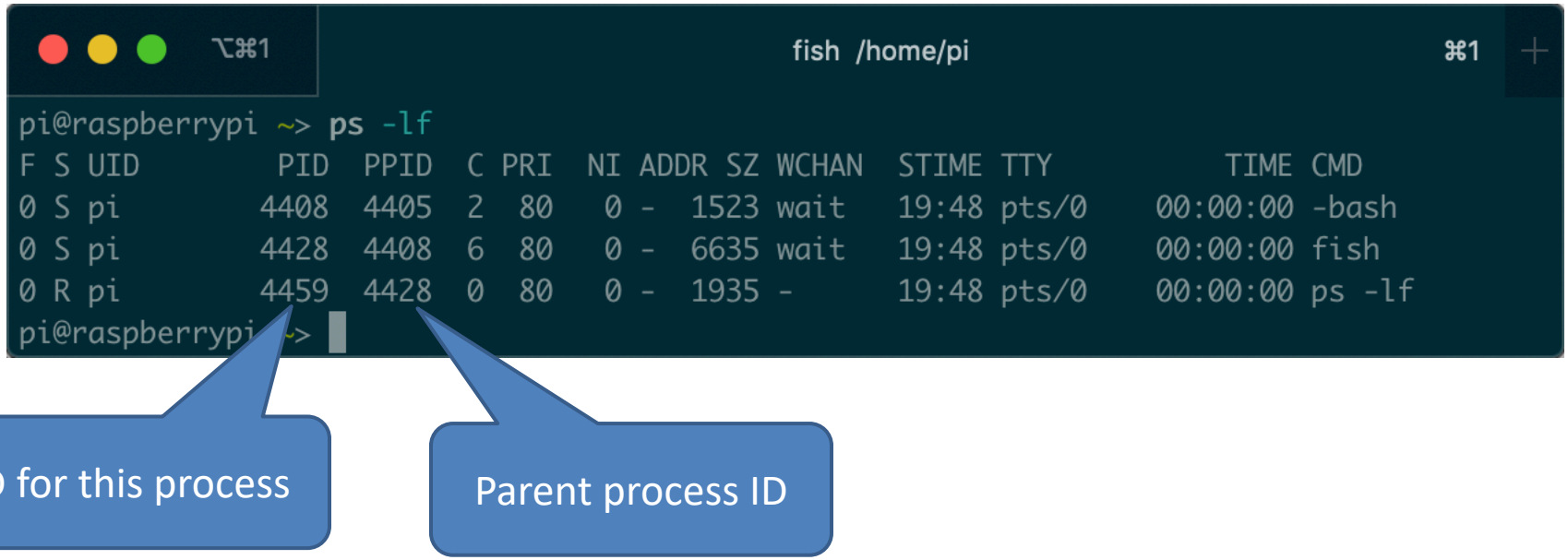
- Process Control Block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - …

Take Linux 4.2 as an example

https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1444

Linux Process Control Block (cont')

- Identifiers
  - pid: ID of the process



Linux Process Control Block (cont')

- How to get the pointer to a specific process ?
  - The **`current`** macro: point to current running process
  - current->pid

https://elixir.bootlin.com/linux/v4.2/source/arch/arc/mm/mmap.c#L34

  - getpid(): one system call (No. 39) https://filippo.io/linux-syscall-table/

https://www.geeksforgeeks.org/getppid-getpid-linux/

Linux Process Control Block (cont')

- How to get the pointer to a specific process ?

```
#include <iostream>
#include <unistd.h>
using namespace std;

// Driver Code
int main()
{
    int pid = fork();
    if (pid == 0)
        cout << "\nCurrent process id of Proce
             << getpid() << endl;
    return 0;
}
```

```
Current process id of Process : 4195
```

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

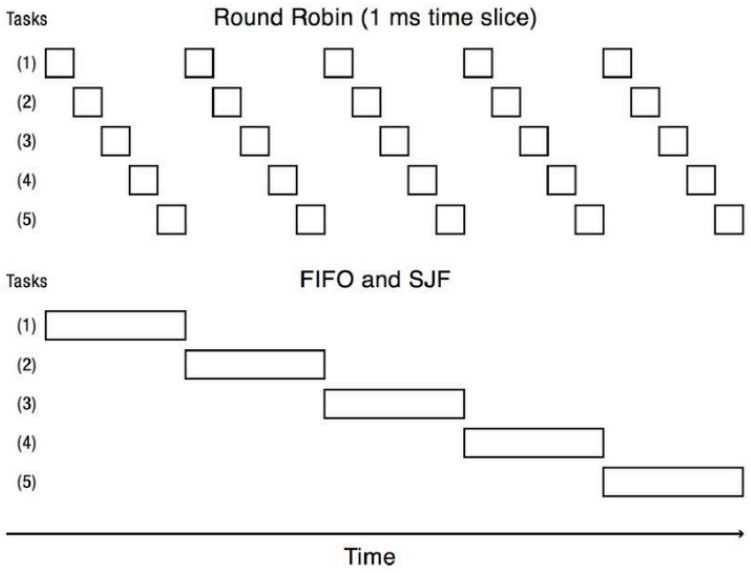
Linux Process Control Block (cont')

- 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

Linux Process Control Block (cont')

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

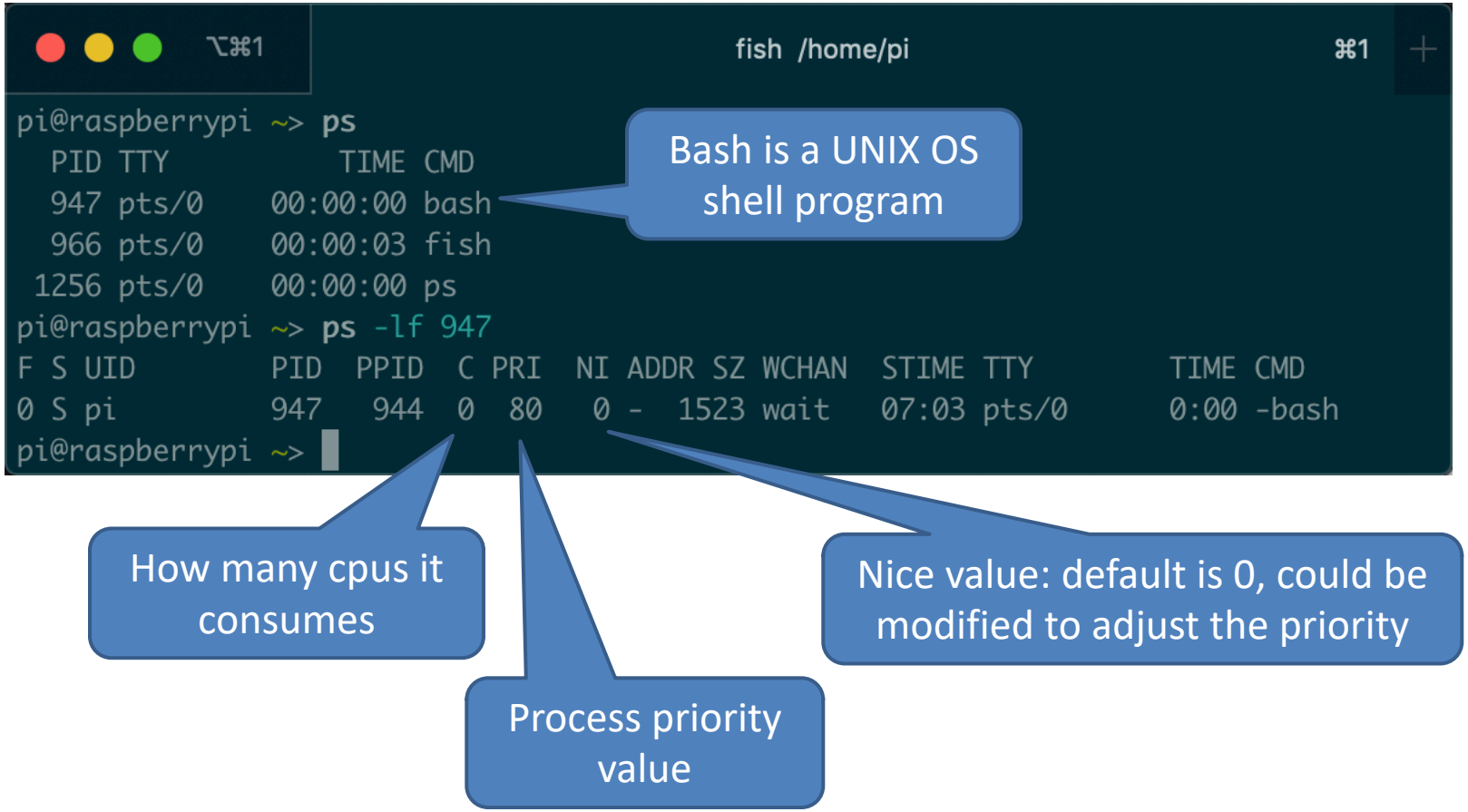
Take Linux 4.2 as an example

https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1399

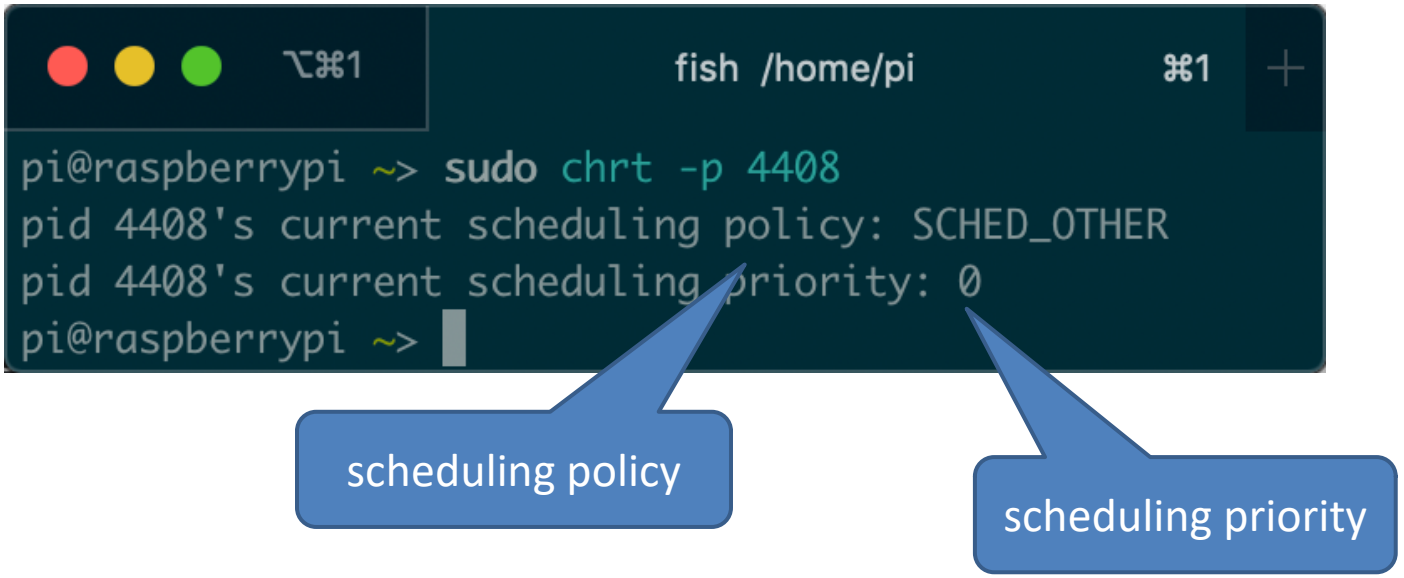
https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1362

https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1363

$ ps -lf : process information



$ chrt: process scheduling info

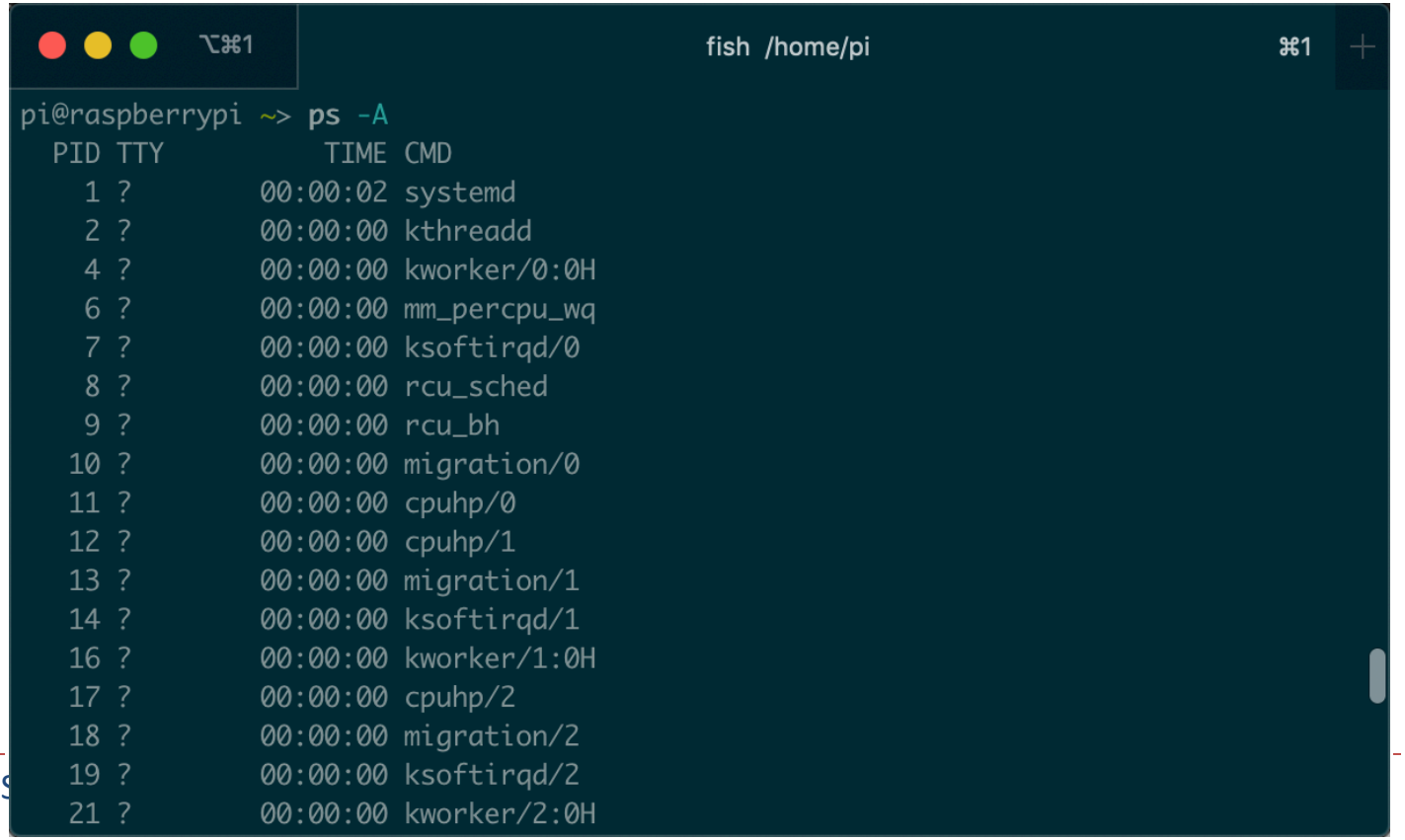


SCHED_OTHER	min/max priority : 0/0
SCHED_FIFO	min/max priority : 1/99
SCHED_RR	min/max priority : 1/99
SCHED_BATCH	min/max priority : 0/0

More about command

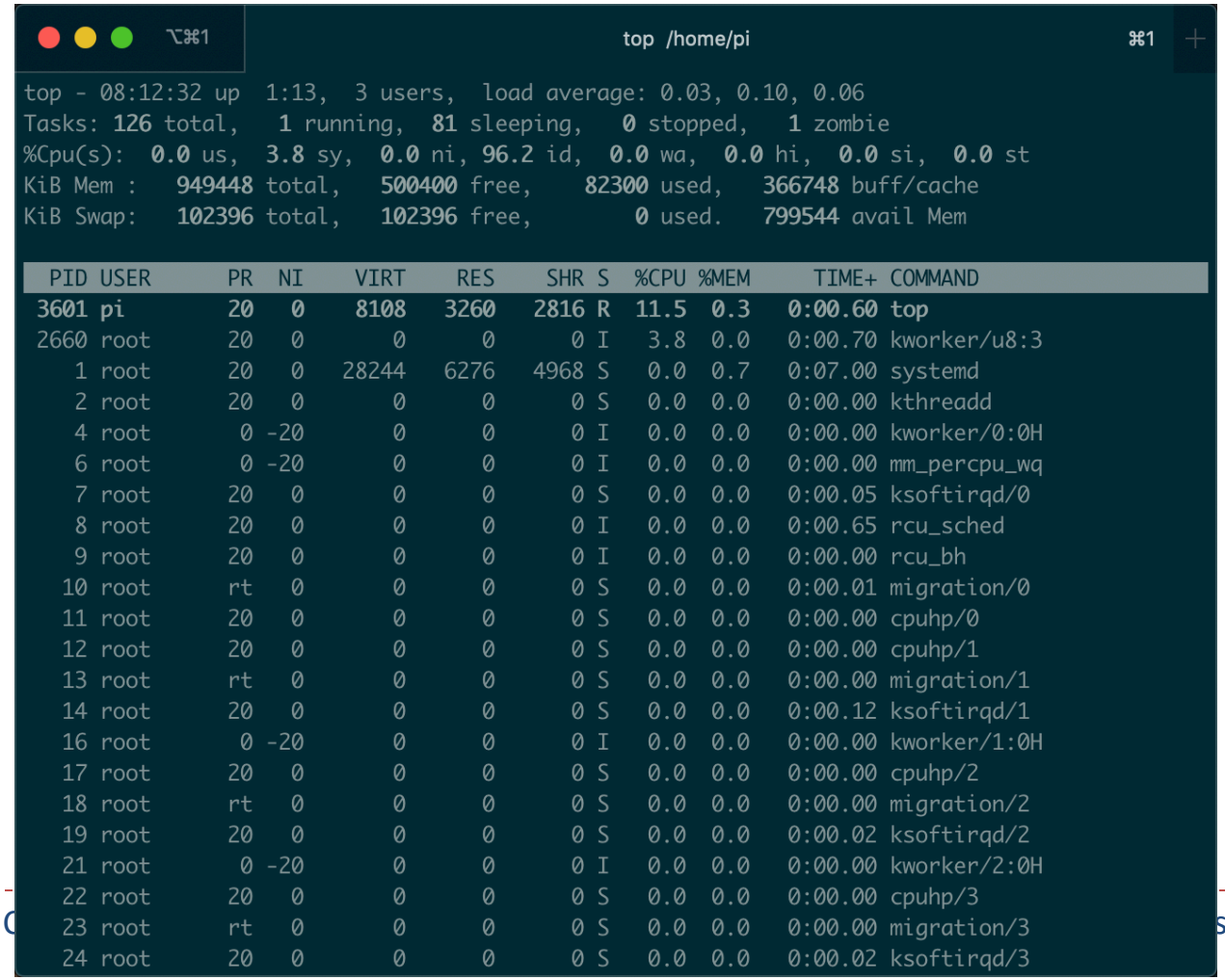
- $ ps -A or $ ps -ef

show all information for all process



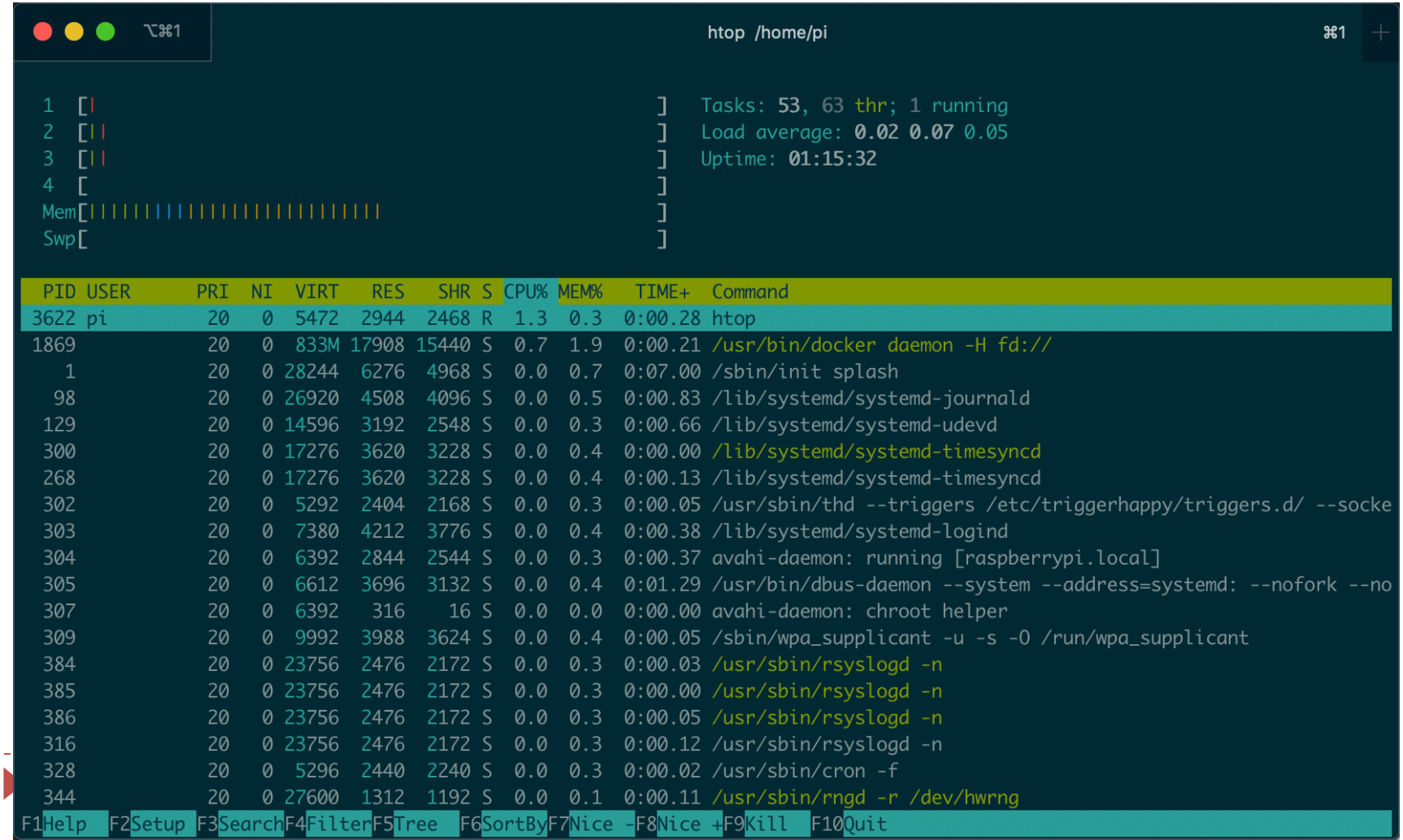
More about command

- $ top: displays processor activity



More about command

- $ htop: an advance top

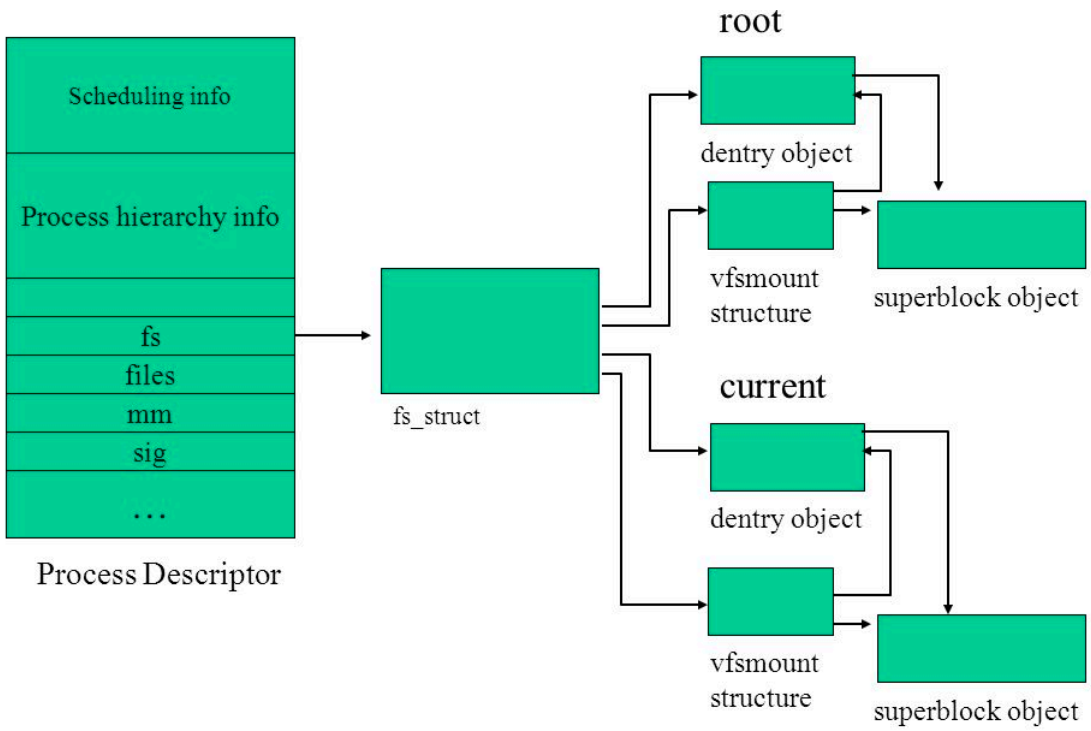


Process in Linux

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

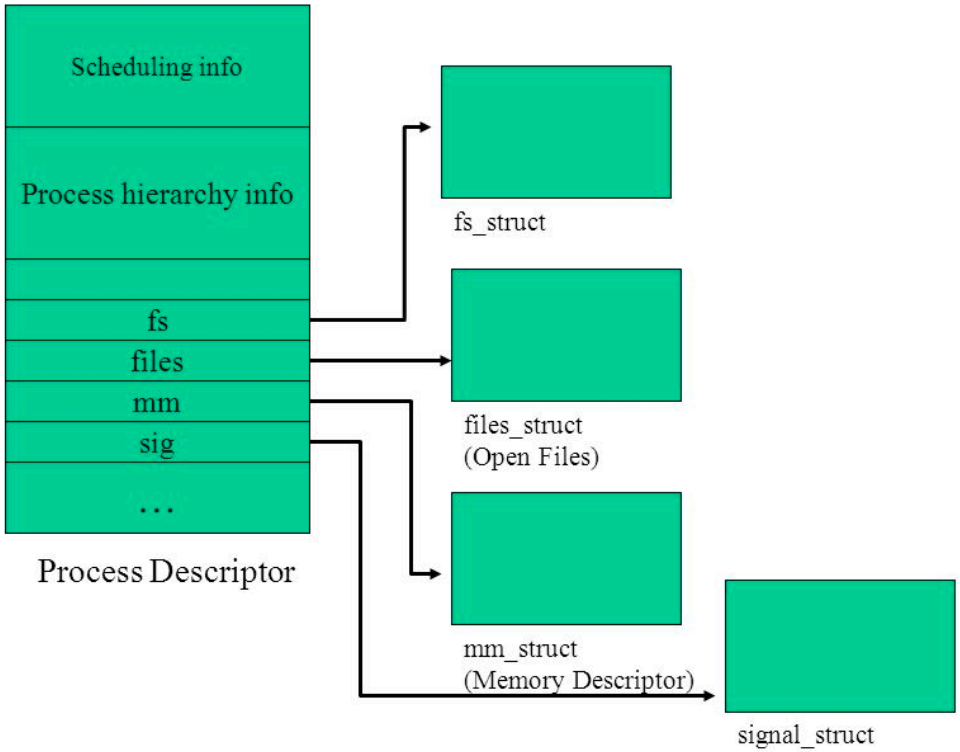
Linux Process Control Block (cont')

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

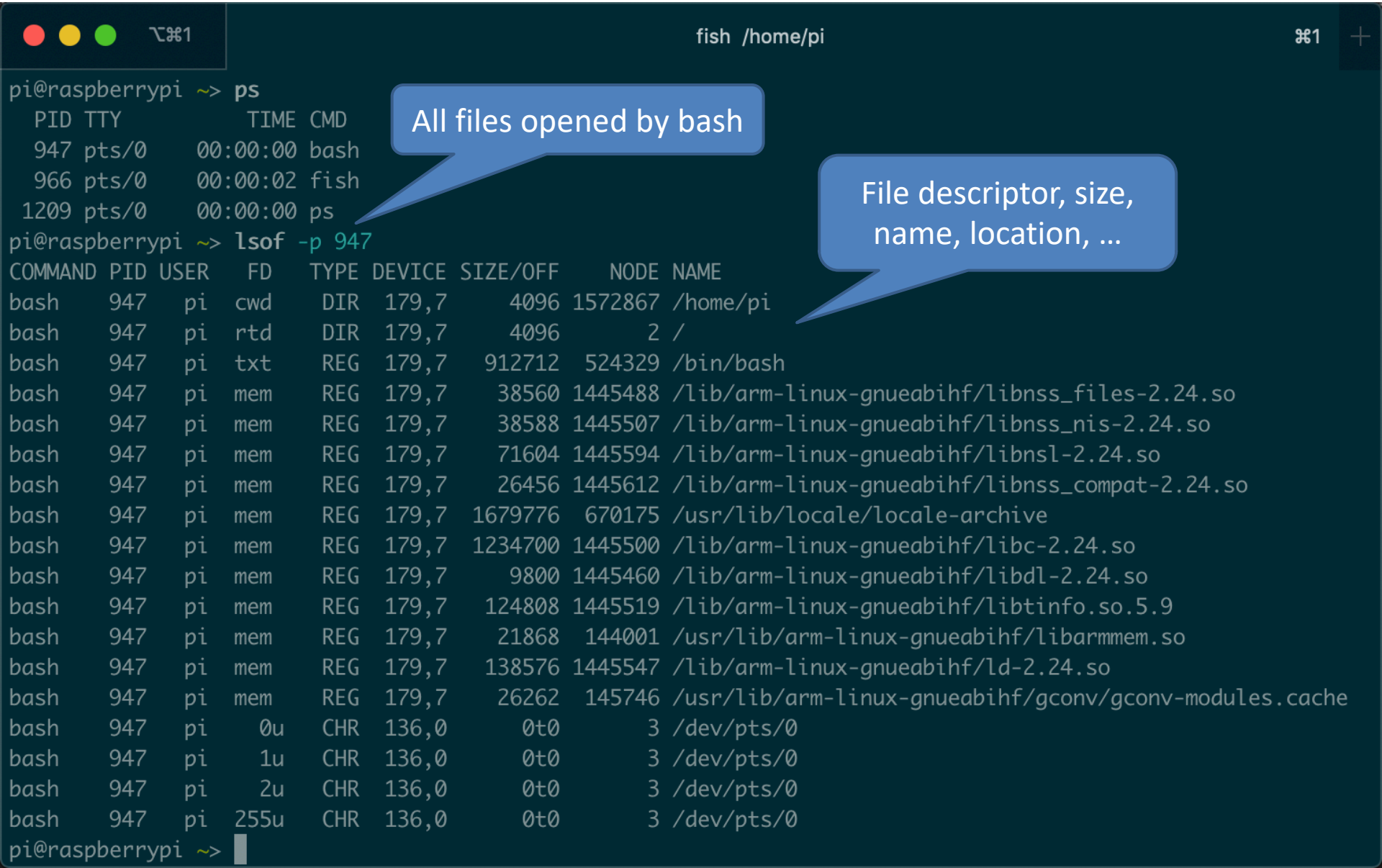


Linux Process Control Block (cont')

- 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

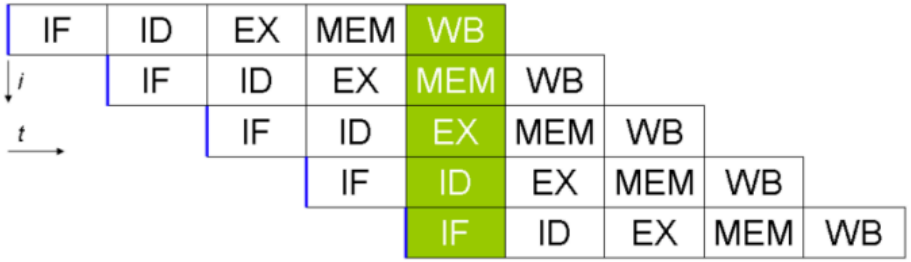


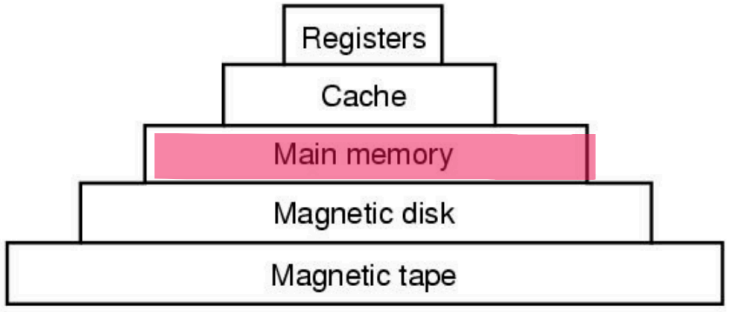
Process in Linux

- 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





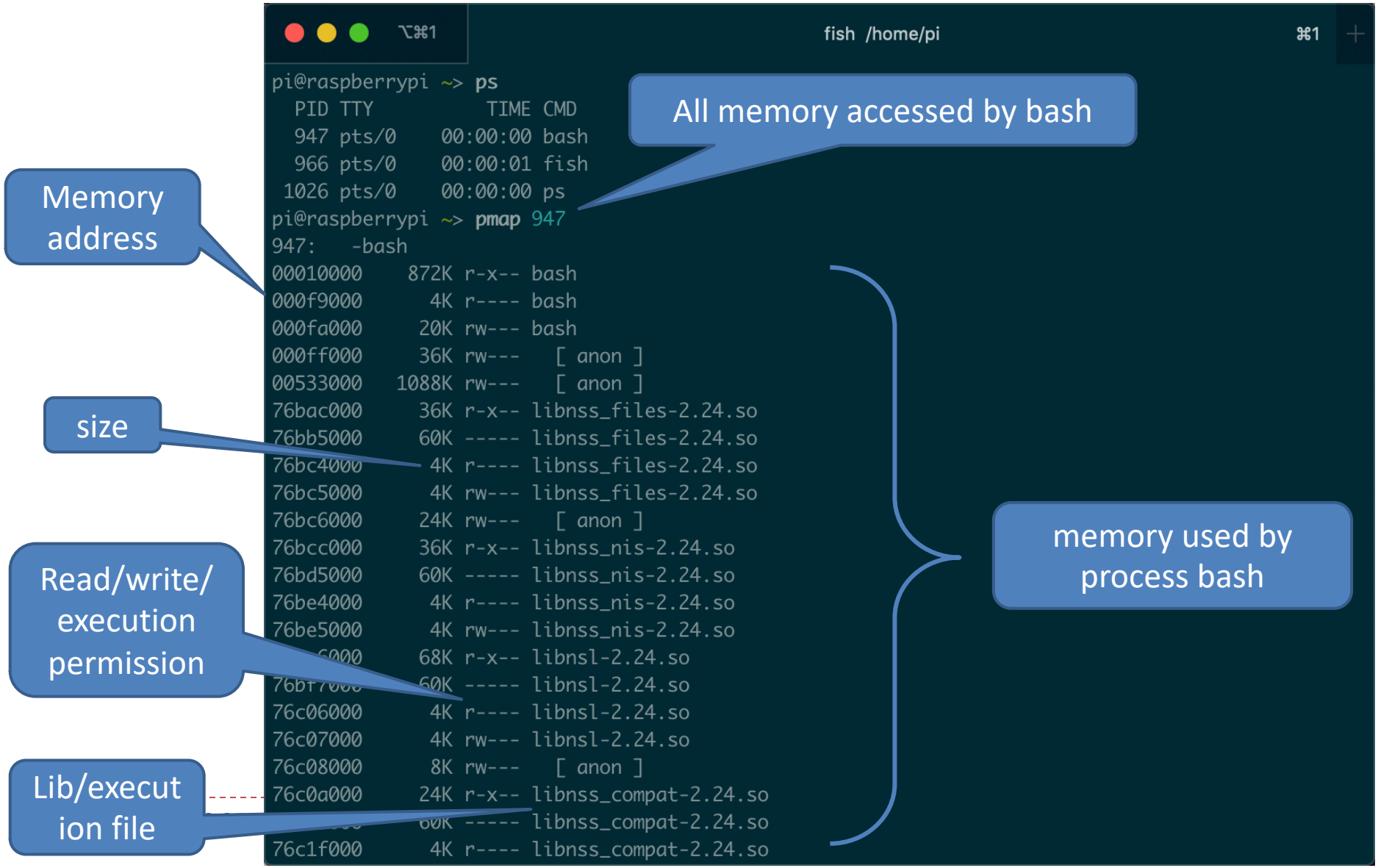
Linux Process Control Block (cont')

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

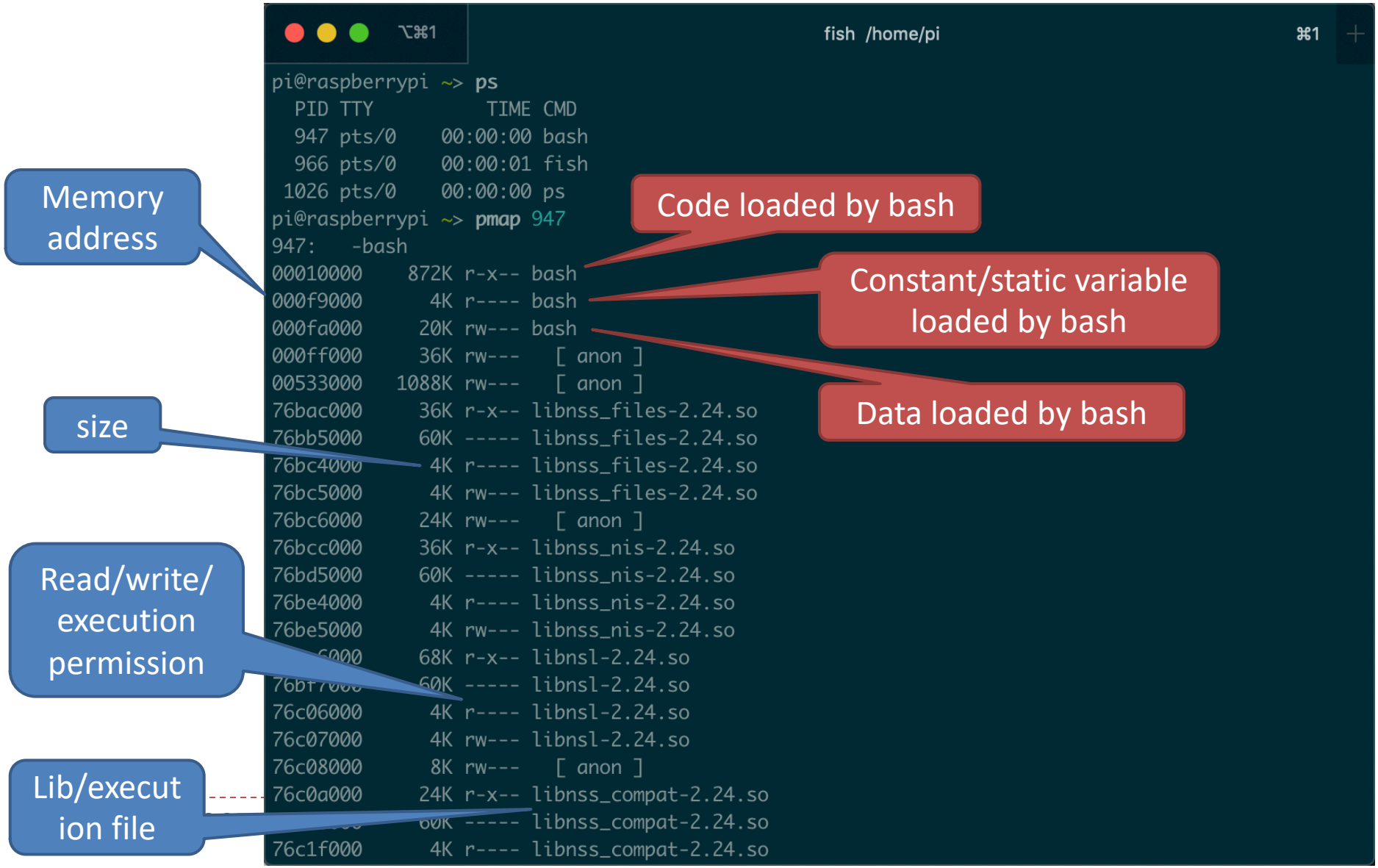
Take Linux 4.2 as an example

https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1408

$pmap: memory mapping



$pmap: memory mapping



$pmap: memory mapping

Bash in memory

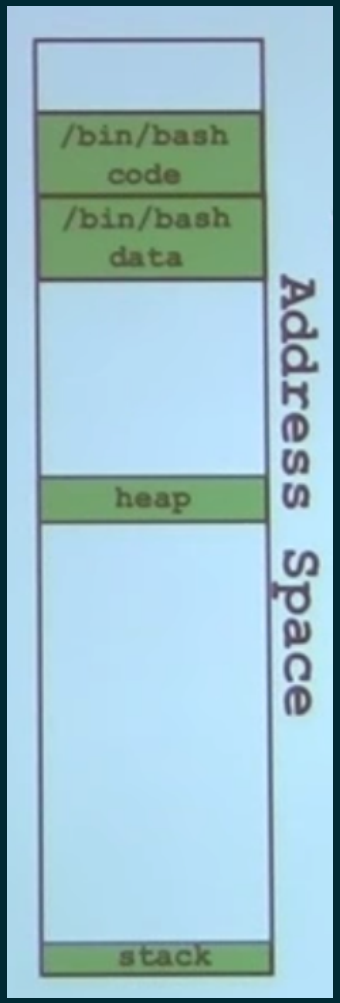
Memory address

size

Read/write/ execution permission

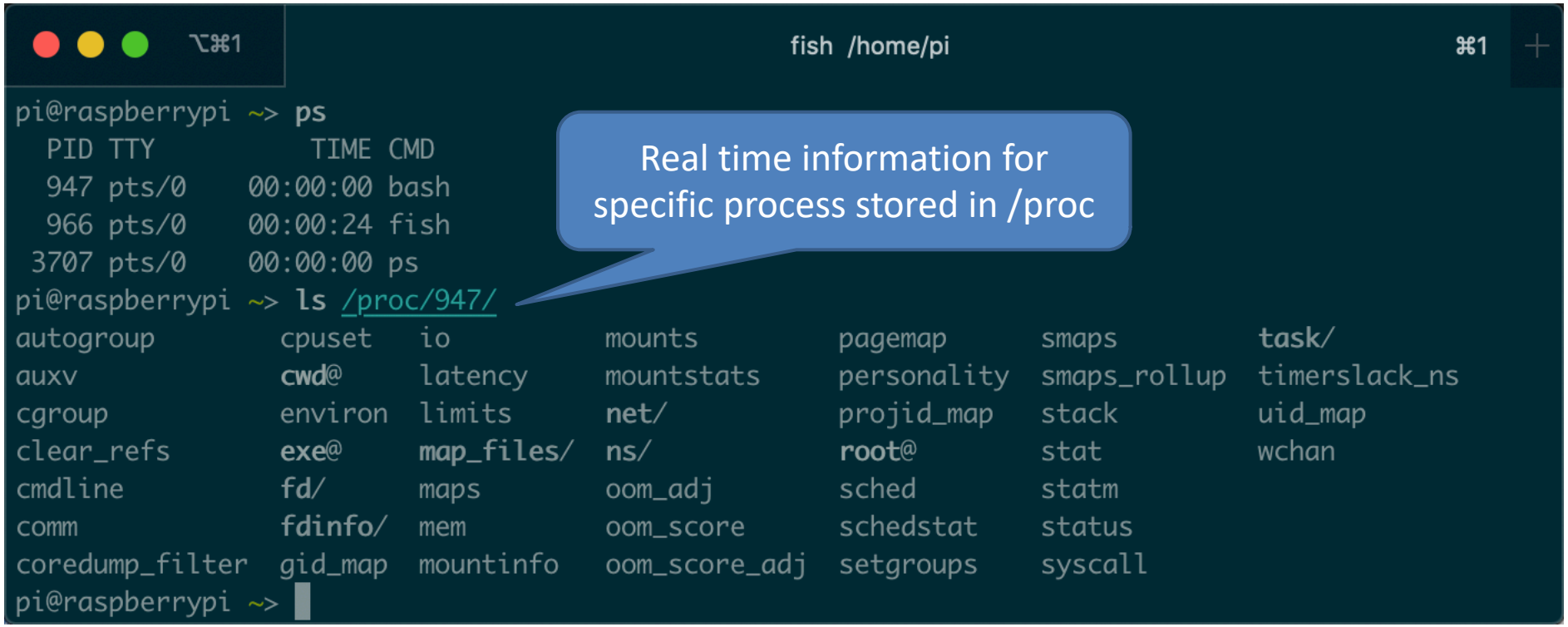
Lib/execut ion file

```
pi@raspberrypi ~> ps
  PID TTY          TIME CMD
  947 pts/0    00:00:00 bash
  966 pts/0    00:00:01 fish
 1026 pts/0    00:00:00 ps
pi@raspberrypi ~> pmap 947
947:   -bash
00010000    872K r-x-- bash
000f9000      4K r---- bash
000fa000     20K rw--- bash
000ff000     36K rw---   [ anon ]
00533000   1088K rw---   [ anon ]
76bac000     36K r-x-- libnss_files-2.24.so
76bb5000     60K ----- libnss_files-2.24.so
76bc4000      4K r---- libnss_files-2.24.so
76bc5000      4K rw--- libnss_files-2.24.so
76bc6000     24K rw---   [ anon ]
76bcc000     36K r-x-- libnss_nis-2.24.so
76bd5000     60K ----- libnss_nis-2.24.so
76be4000      4K r---- libnss_nis-2.24.so
76be5000      4K rw--- libnss_nis-2.24.so
              68K r-x-- libnsl-2.24.so
              60K ----- libnsl-2.24.so
76c06000      4K r---- libnsl-2.24.so
76c07000      4K rw--- libnsl-2.24.so
76c08000      8K rw---   [ anon ]
76c0a000     24K r-x-- libnss_compat-2.24.so
              60K ----- libnss_compat-2.24.so
76c1f000      4K r---- libnss_compat-2.24.so
```

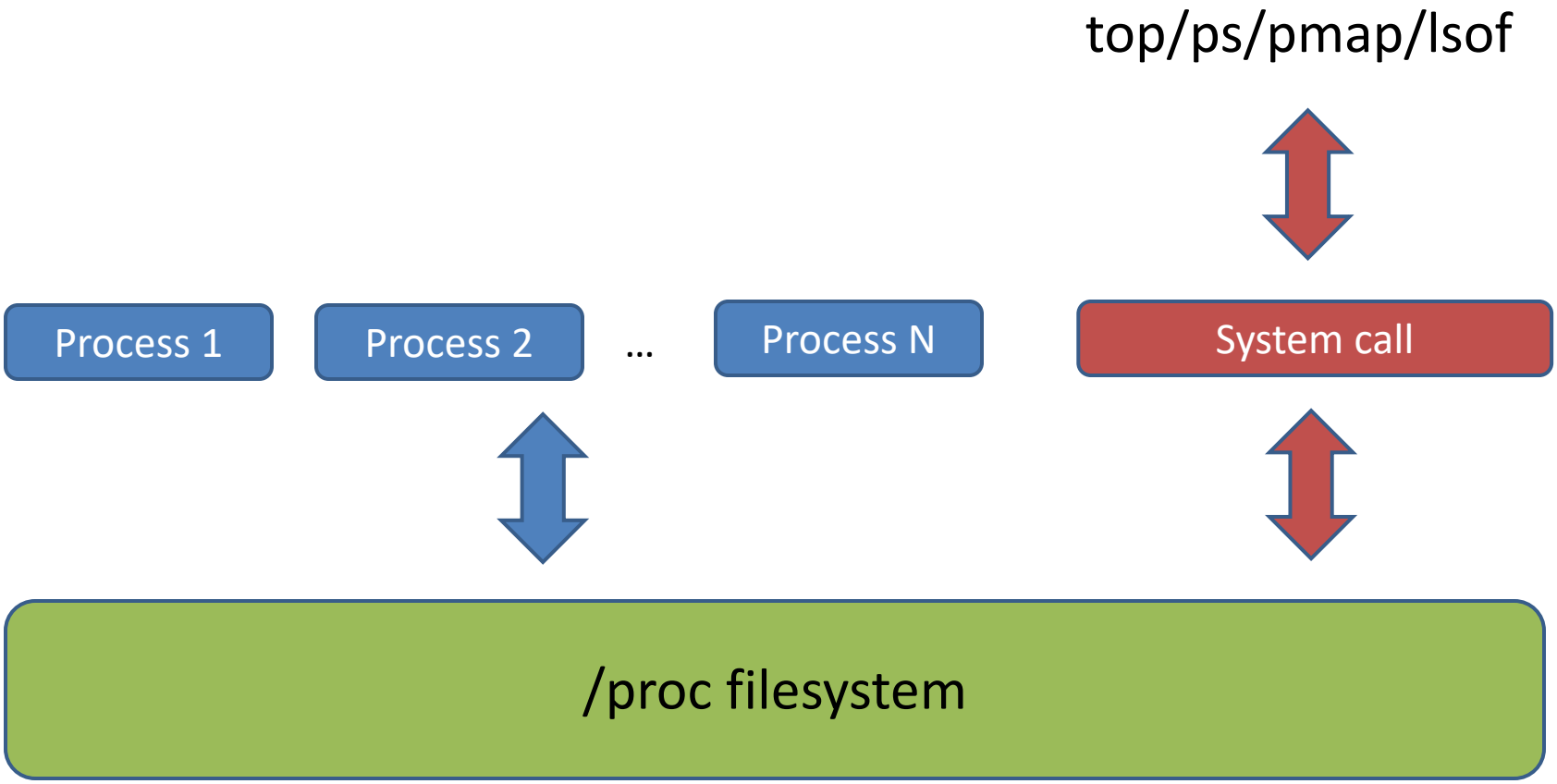


Aside: the /proc/ filesystem

- How do top/ps/pmap/lsof and other process command gather information?
- Linux reuses the file abstraction for this purpose



Aside: the /proc/ filesystem

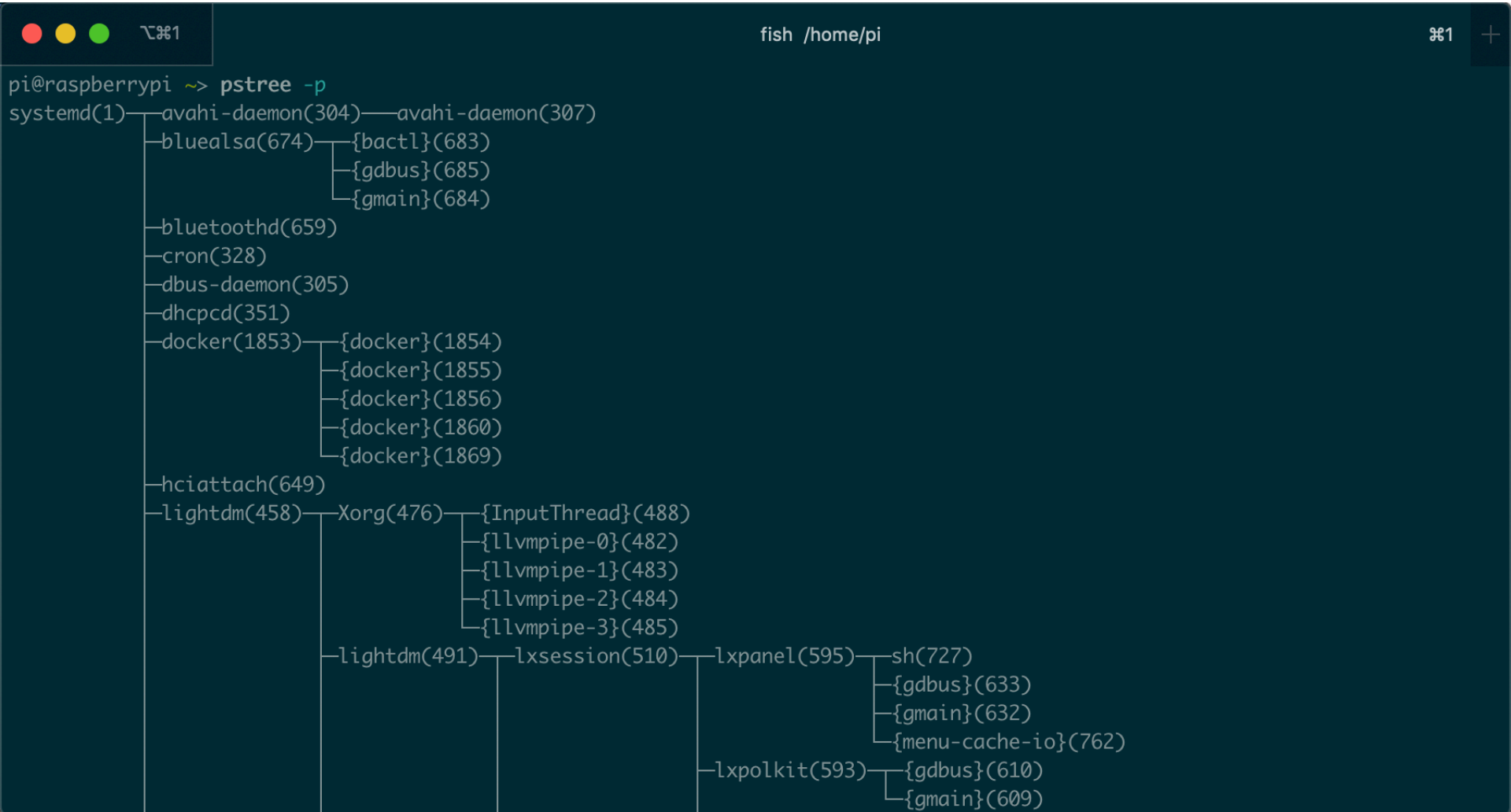


Outline

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Orphan and Zombie process

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/latest/source/init/main.c#L551

- 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/CS3502/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() system call

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

Fork() system call

https://github.com/kevinsuo/CS3502/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!!

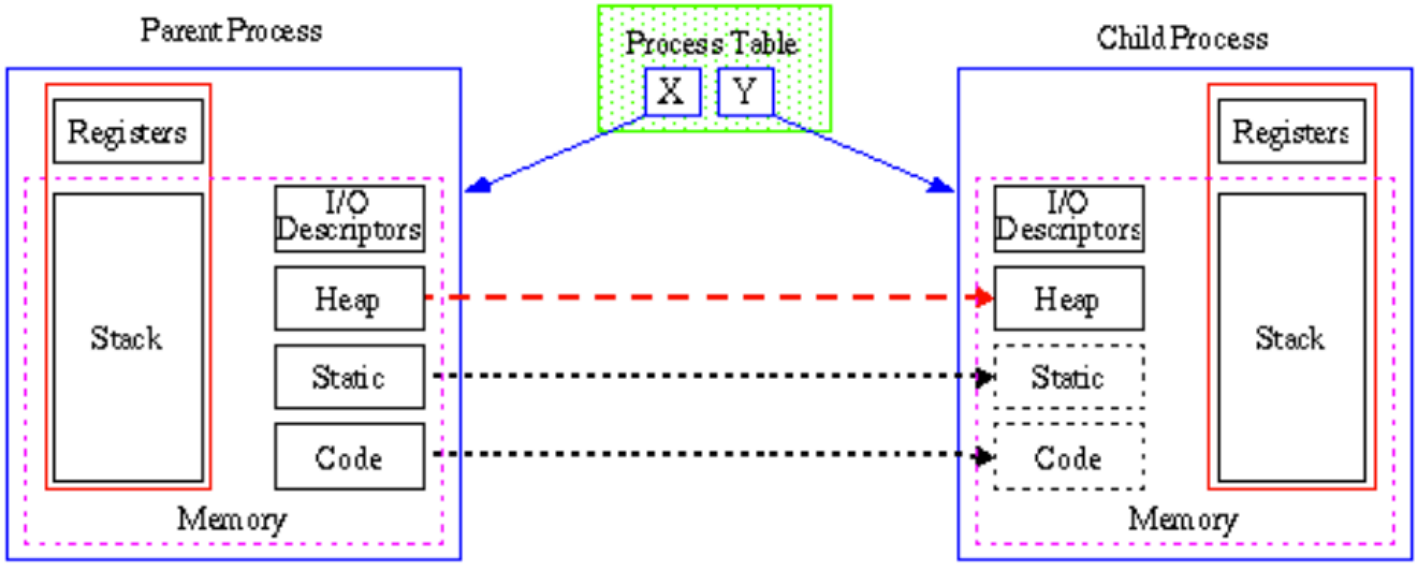
```
pi@raspberrypi ~/Downloads> ./test.o
The parent pid is 2510, pid:2511
The child pid is 2511, pid:0
```

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

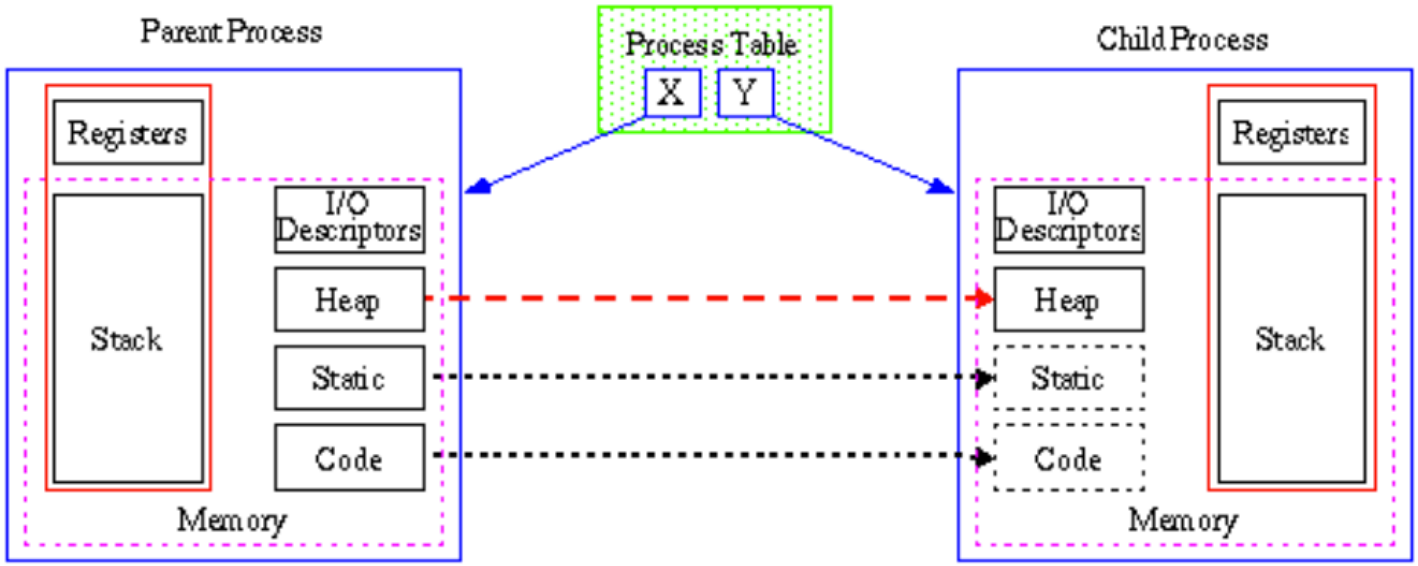
- When a parent process uses fork() to create a child process, the two processes have
  - the **same** program text.
  - but **separate** copies of the data, stack, and heap segments.



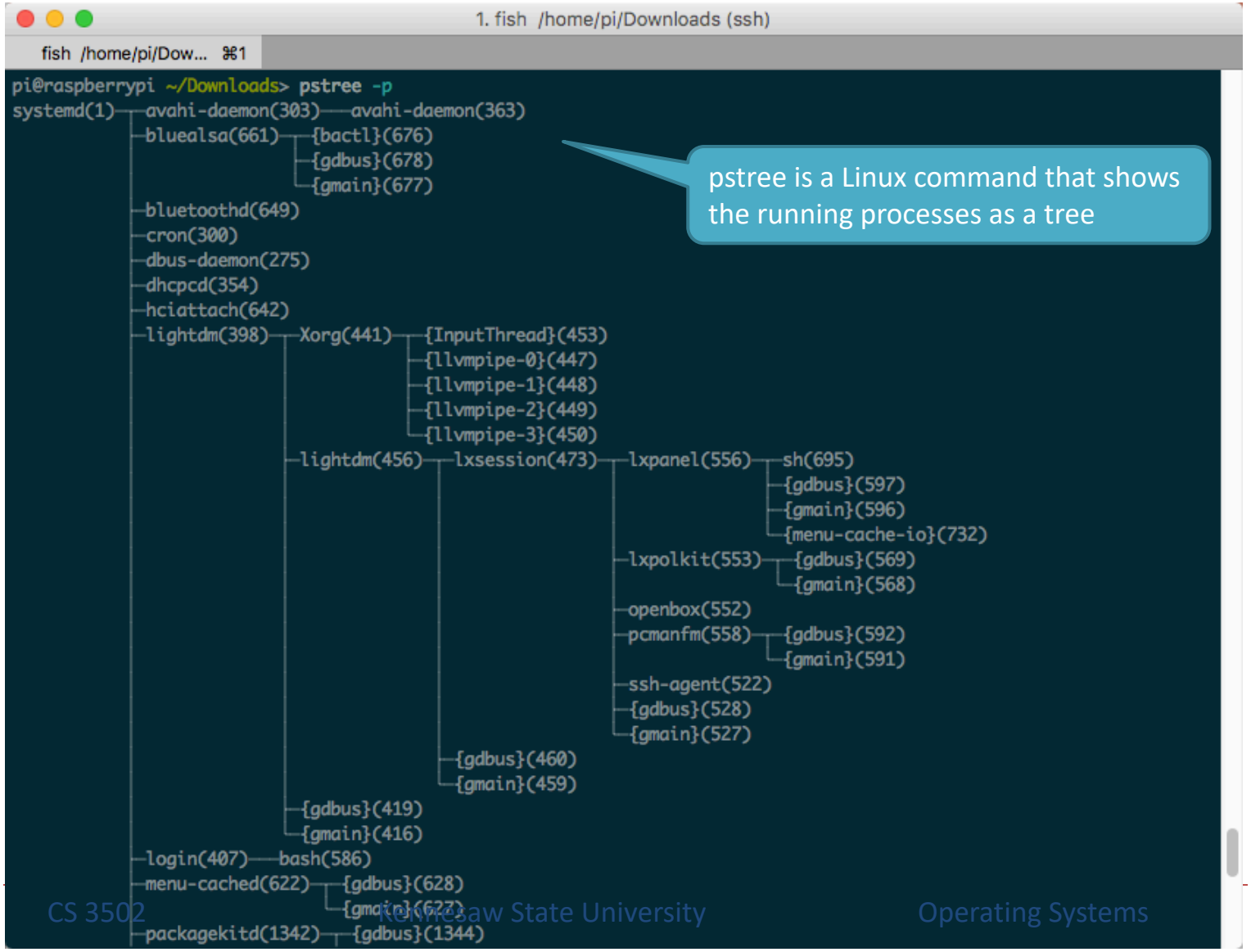
Parent process and child process

- 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.



Process tree in Linux



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

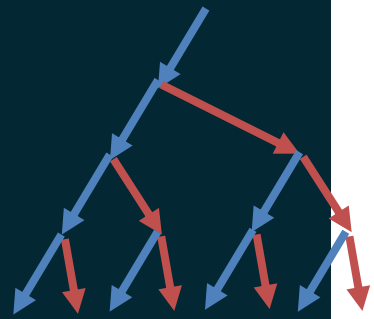
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        fork();

        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!
```

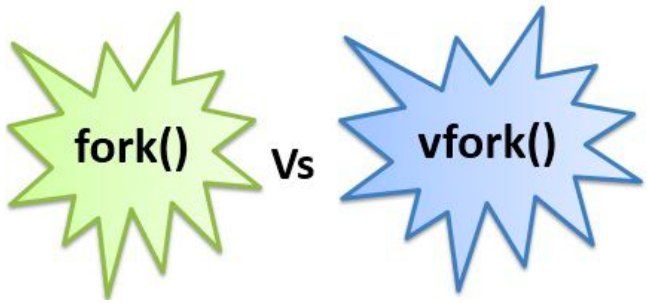
Issues with fork()

Copying all that state is **expensive**!

- Especially when the next thing that a process frequently does is start load a new binary which destroys most of the state fork() has carefully copied!

- Several solutions to this problem:
  - Optimize existing semantics -- through **copy-on-write**, a clever memory-management optimization we will discuss later.
  - Change the semantics -- **vfork** (), which will fail if the child does anything other than immediately load a new executable.
    - Does not copy the address space!

vfork()



- fork(): child process **copies data segment** and **code segment** from parent process
- vfork(): child process **shares data segment** from parent process

- fork(): the **order** of execution between parent and child process is **unknown**
- vfork(): the **order** of execution between parent and child process is **child** process **first**

A fork() bomb

- What does this code do?

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

Agent smith

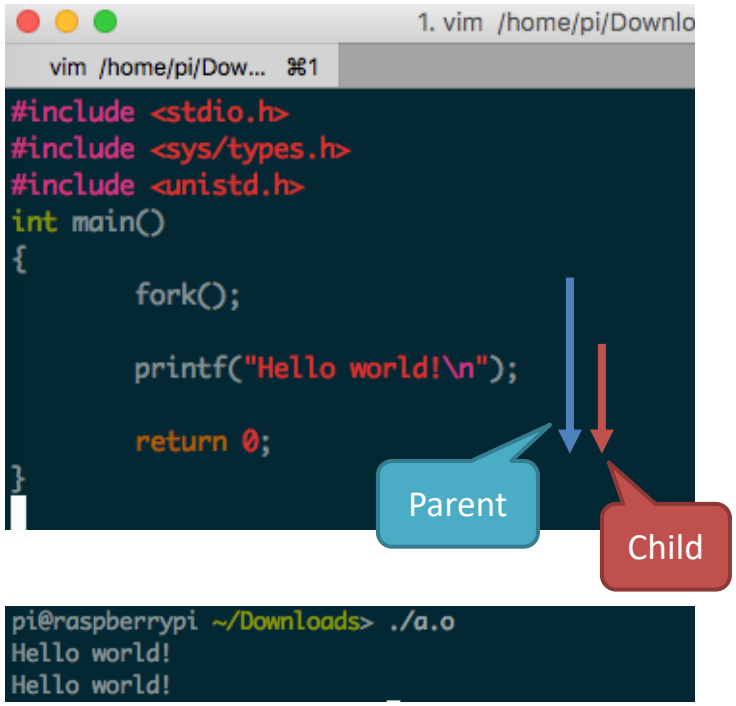
Exec() system call

- 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/CS3502/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

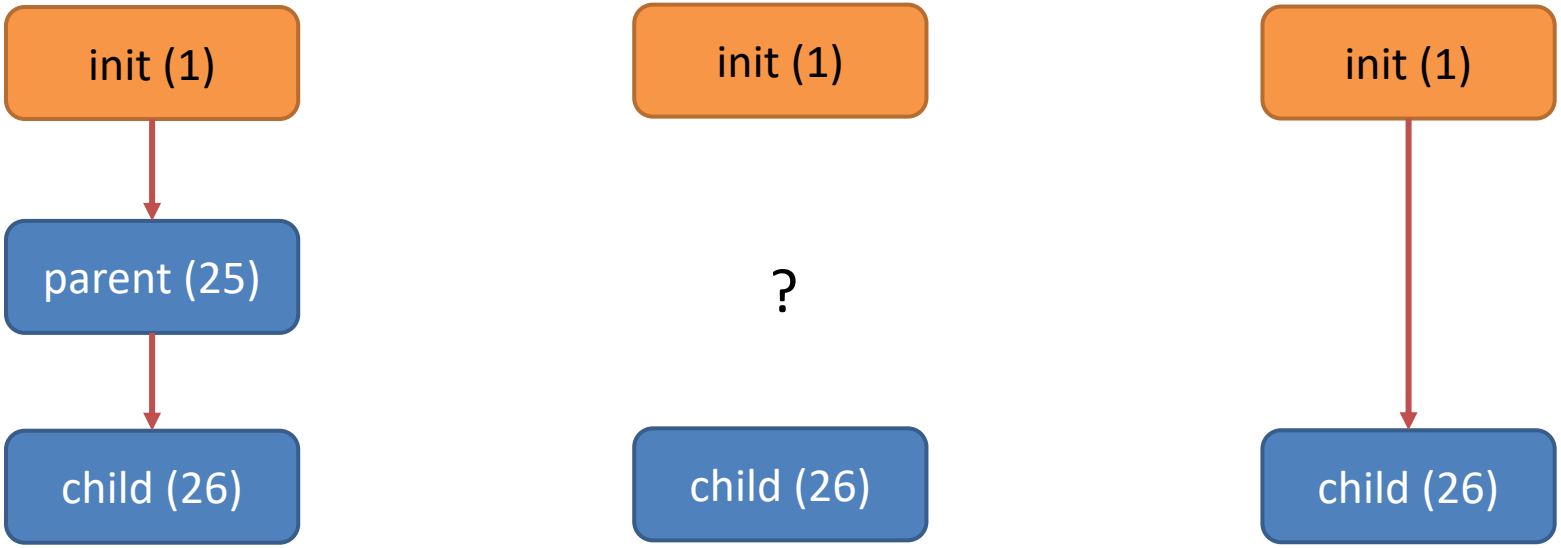
Outline

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Orphan and Zombie process

Orphan process

https://github.com/kevinsuo/CS3502/blob/master/orphan.c

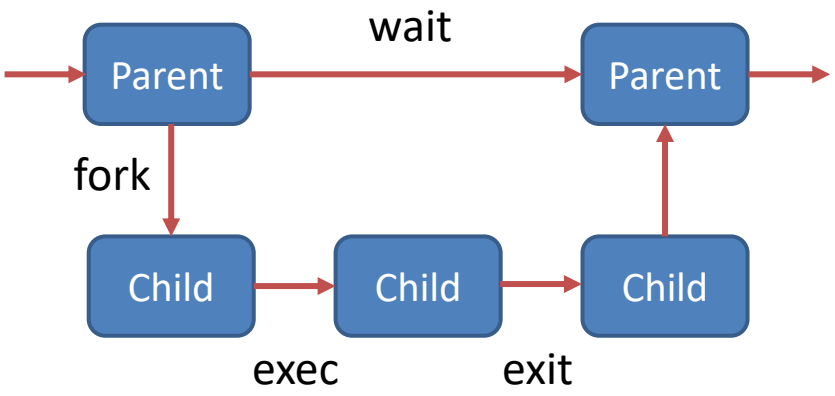
- When a parent process dies, child process becomes an orphan process.
- The init process (pid = 1) becomes the parent of the orphan processes.
- Parent stops first

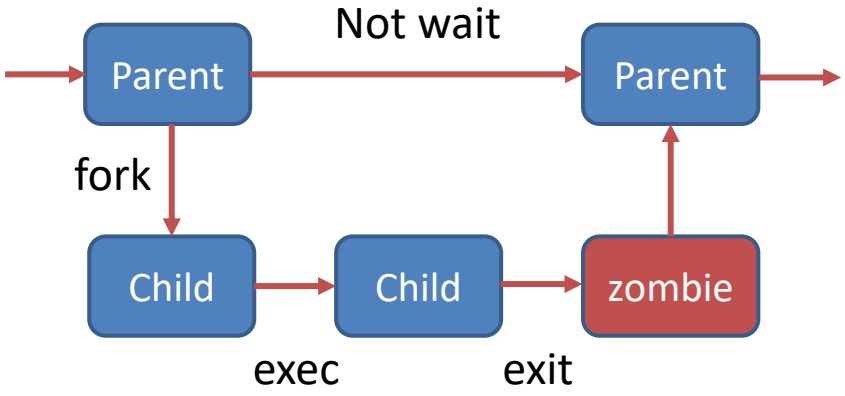


Zombie process

https://github.com/kevinsuo/CS3502/blob/master/zombie.c

- When a child dies, a SIGCHLD signal is sent to the parent.
- If parent doesn't wait() on the child, and child exit()s, it becomes a zombie (status "Z" seen with ps).
- Child process stops first





Orphan process vs Zombie process

- The zombie process will consume system resources. If it is a lot, it will seriously affect the performance of the server.

- The orphan process does not occupy system resources, and is ultimately hosted by the init process and is also released by the init process.

Orphan process vs Zombie process

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

int main()
{
        pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0) {
                printf("parent is stopped!\n");
                exit(0);
        }
        // Child process
        else {
                sleep(10);
                printf("child is stopped!");
        }

        return 0;
}
```

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

int main()
{
        pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0) {
                printf("parent process pid:%d\n", getpid());
                sleep(100);
                printf("parent is stopped!\n");
        }
        // Child process
        else {
                printf("child process pid:%d\n", getpid());
                exit(0);
        }

        return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./orphan.o
parent is stopped!
pi@raspberrypi ~/Downloads> child is stopped!
```

After 10 seconds

```
pi@raspberrypi ~/Downloads> ./zombie.o
parent process pid:3857
child process pid:3858
```

The child process is zombie process

```
pi@raspberrypi ~> ps aux | grep Z
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
pi         695  0.0  0.0      0     0 ?        Z    14:46   0:00 [sh] <defunct>
pi        3858  0.0  0.0      0     0 pts/1    Z+   17:44   0:00 [zombie.o] <defunct>
pi        3880  0.0  0.0   4372   536 pts/0    S+   17:44   0:00 grep --color=auto Z
```

Orphan process vs Zombie process

```
pi@raspberrypi ~/Downloads> ./zombie.o
parent process pid:3857
child process pid:3858
```

The child process is zombie process

```
pi@raspberrypi ~> ps aux | grep Z
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
pi         695  0.0  0.0      0     0 ?        Z    14:46   0:00 [sh] <defunct>
pi        3858  0.0  0.0      0     0 pts/1    Z+   17:44   0:00 [zombie.o] <defunct>
pi        3880  0.0  0.0   4372   536 pts/0    S+   17:44   0:00 grep --color=auto Z
```

The child process is zombie process

Avoid Zombie processes

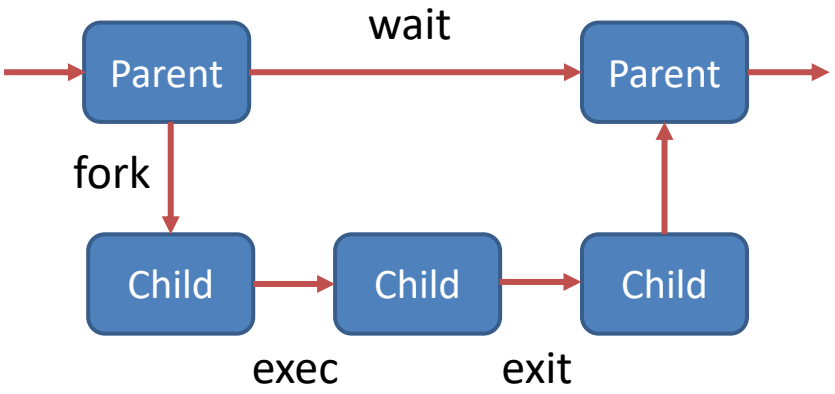
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0) {
                printf("parent process pid:%d\n", getpid());
                wait(NULL);
                sleep(100);
                printf("parent is stopped!\n");
        }
        // Child process
        else {
                printf("child process pid:%d\n", getpid());
                exit(0);
        }

        return 0;
}
```

To Avoid Zombie process




```
pi@raspberrypi ~/Downloads> ./zombie.o
parent process pid:3998
child process pid:3999
```

The child process is not zombie process

```
pi@raspberrypi ~> ps aux | grep Z
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
pi         695  0.0  0.0      0     0 ?        Z    14:46   0:00 [sh] <defunct>
pi        4004  0.0  0.0   4372   556 pts/0    S+   17:58   0:00 grep --color=auto Z
```

Conclusion

- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Orphan and Zombie process

Project 1

Assignment 2: Extend your new system call to print out the calling process's information (25 points)

Follow the instructions we discussed above and implement another system call print_self. This system call identifies the calling process at the user-level and print out various information of the process.

Implement the print_self system call and print out the following information of the calling process:

- Process id, running state, and program name
- Start time and virtual runtime
- Its parent processes until init (first system process)

HINT: The macro current returns a pointer to the task_struct of the current running process.

Please use diff command to highlight your modification:
$ diff -u original_file.c modified_file.c > result.txt

Assignment 3: Extend your new system call to print out the information of an arbitrary process identified by its PID (25 points)

Implement another system call print_other to print the information for an arbitrary process. The system call takes a process pid as its argument and outputs the above information of this process.