

# 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

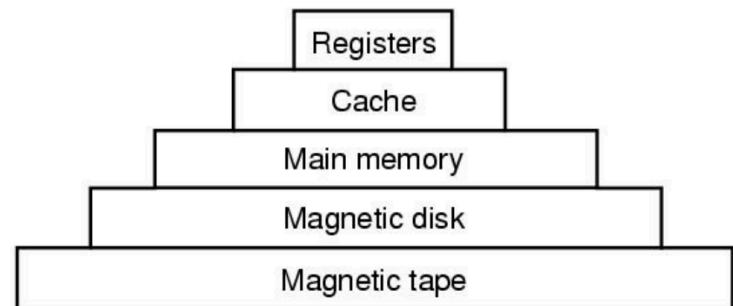
---

- 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 ...



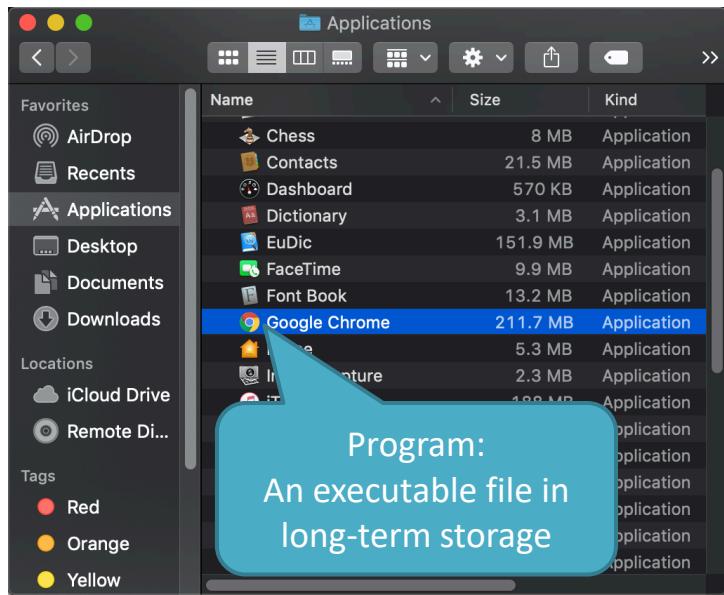
# Program vs. Process

---

- Program  $\neq$  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:  
An executable file in  
long-term storage

A screenshot of the Activity Monitor application. The title bar says "work" and "Search". The main table has columns: Process Name, CPU Usage, CPU Load, and Threads. A red dashed box highlights the "Process Name" column, which lists numerous "Google Chrome Helper" entries. A single "Google Chrome" entry is also visible at the bottom. The "CPU LOAD" section shows values: 3.65%, 6.11%, and 90.24%. The "Threads" section shows 1246 total threads and 350 processes.

Process Name	CPU Usage	CPU Load	Threads
identityservicesd	0.0	0.00	2
iconservicesagent	0.0	0.27	2
icdd	0.0	0.32	11
Google Chrome Helper	0.0	22.01	12
Google Chrome Helper	0.4	2:55.56	9
Google Chrome Helper	0.0	0.34	9
Google Chrome Helper	0.1	23.36	12
Google Chrome Helper	0.0	5.87	12
Google Chrome Helper	0.0	9.07	12
Google Chrome Helper	0.0	0.93	12
Google Chrome Helper	0.0	7.78	11
Google Chrome Helper	0.0	33.53	10
Google Chrome Helper	0.0	5.18	13
Google Chrome Helper	0.0	2.06	6
Google Chrome Helper	0.0	0.11	14
Google Chrome	3.8	5:44.46	40
ronta	0.0	17.40	2
upd	0.0	0.03	2
ClientXPCService	0.0	0.06	2

Process:  
The running instantiation of  
a program, stored in RAM

One-to-many  
relationship between  
program and processes



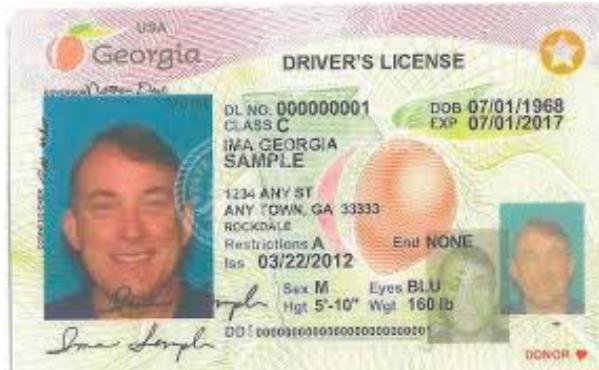
# Program vs. Process

<b>BASIS FOR COMPARISON</b>	<b>PROGRAM</b>	<b>PROCESS</b>
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

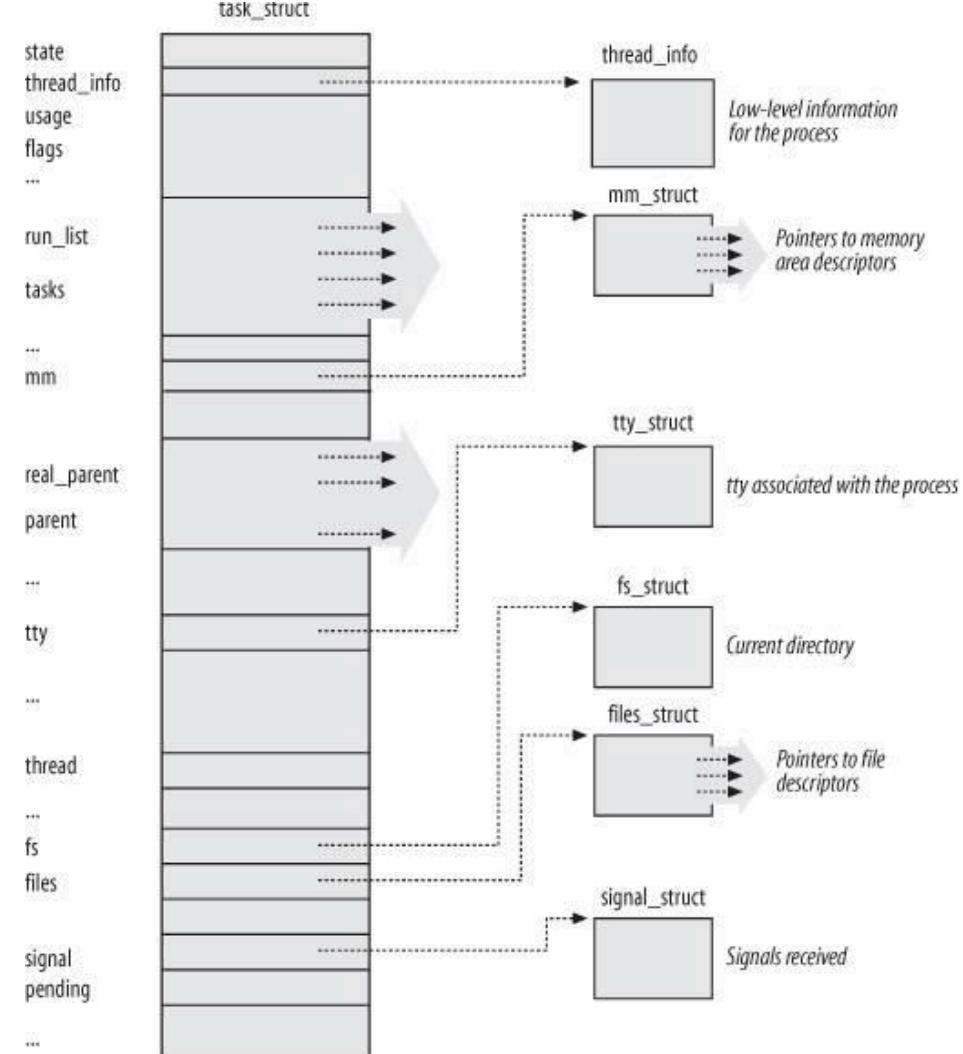


- Driving license
  - ID
  - Name
  - Address
  - Birth
  - Time
  - ...
- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - ...

# 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

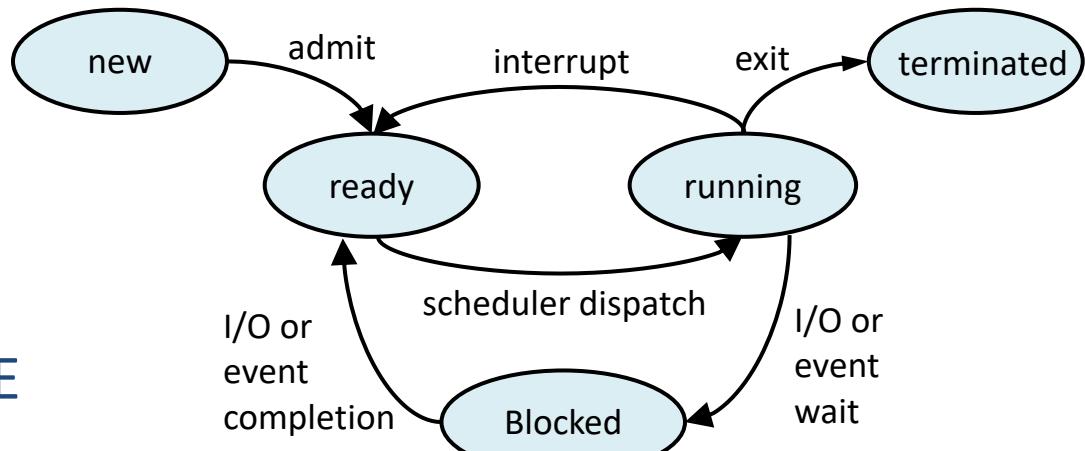
- Process specific context

- ...

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

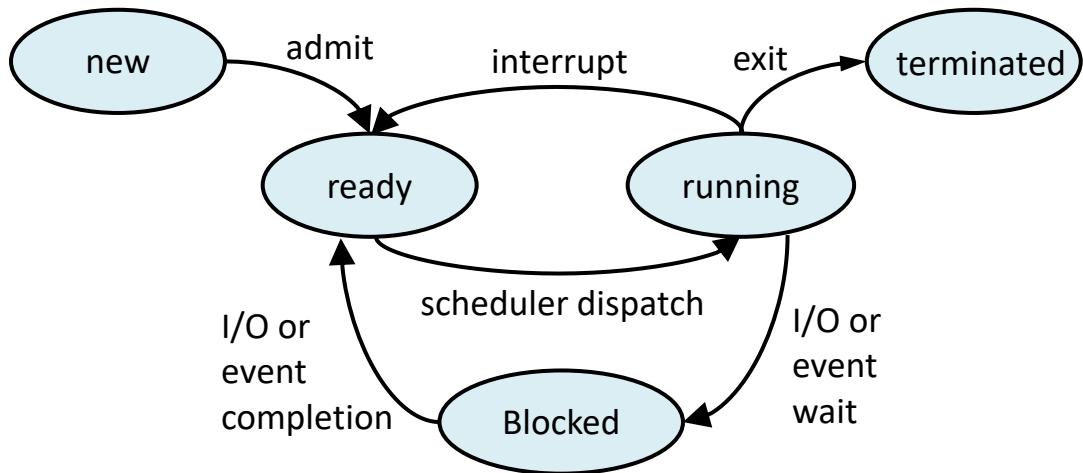
- State

- **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?

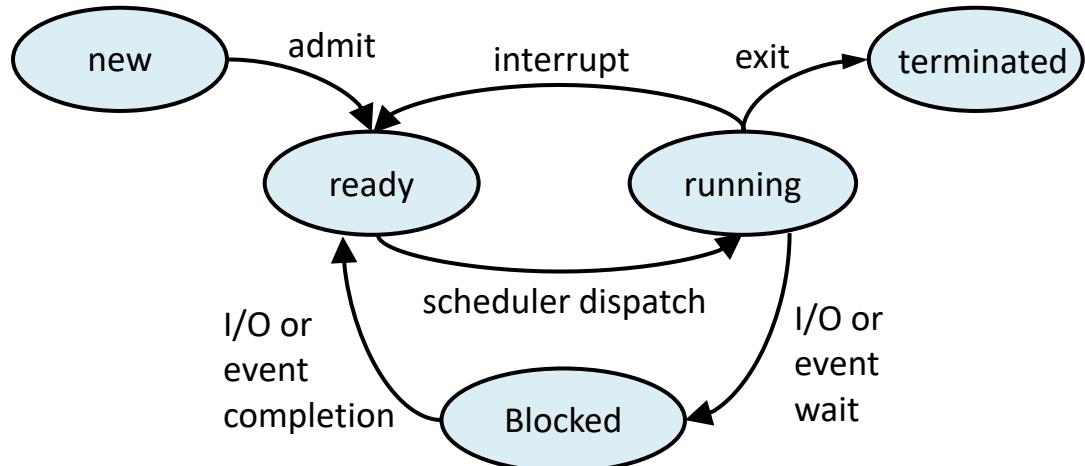


Blocked: unable to run, wait for an event/data

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 <https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1345>
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - ...



# \$ ps -lf : process information

```
pi@raspberrypi ~> ps
  PID TTY          TIME CMD
  947 pts/0        00:00:00 bash
  966 pts/0        00:00:03 fish
 1256 pts/0        00:00:00 ps
pi@raspberrypi ~> ps -lf 947
F S  UID          PID  PPIID C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S pi            947   944  0  80    0 - 1523 wait    07:03 pts/0        0:00 -bash
pi@raspberrypi ~>
```

Bash is a UNIX OS  
shell program

The state of the  
process

- 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

```
pi@raspberrypi ~> ps -lf
F S UID      PID  PPID   C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S pi        4408  4405   2 80    0 - 1523 wait    19:48 pts/0    00:00:00 -bash
0 S pi        4428  4408   6 80    0 - 6635 wait    19:48 pts/0    00:00:00 fish
0 R pi        4459  4428   0 80    0 - 1935 -       19:48 pts/0    00:00:00 ps -lf
pi@raspberrypi ~>
```

ID for this process

Parent process ID

# 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/arm/mm/mm\\_ap.c#L34](https://elixir.bootlin.com/linux/v4.2/source/arch/arm/mm/mm_ap.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:  $P_1 > P_2 = P_3 = P_4$ ,  $P_1$  can execute whenever it needs;
- (2) Normal priority:  $P_1 = P_2 = P_3 = P_4$ ,  $P_1$  execute depending on the scheduling algorithm;
- (3) Prio: dynamic priority, will change over the time

# Linux Process Control Block (cont')

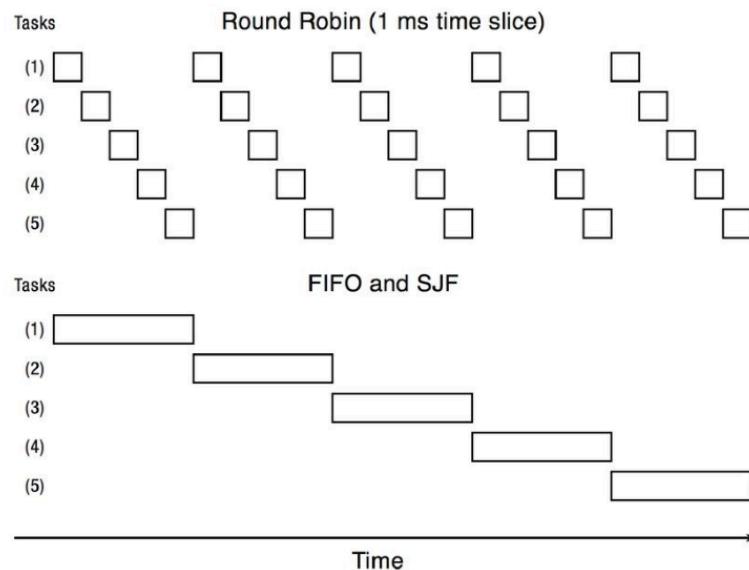
- Scheduling information
  - rt\_priority



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

```
pi@raspberrypi ~> ps
  PID TTY          TIME CMD
  947 pts/0        00:00:00 bash
  966 pts/0        00:00:03 fish
 1256 pts/0        00:00:00 ps
pi@raspberrypi ~> ps -lf 947
F S UID          PID  PPID C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S pi           947   944  0  80    0 - 1523 wait    07:03 pts/0        0:00 -bash
pi@raspberrypi ~>
```

Bash is a UNIX OS shell program

How many cpus it consumes

Nice value: default is 0, could be modified to adjust the priority

Process priority value

# \$ chrt: process scheduling info

```
● ● ● 1 fish /home/pi 1 +  
pi@raspberrypi ~> sudo chrt -p 4408  
pid 4408's current scheduling policy: SCHED_OTHER  
pid 4408's current scheduling priority: 0  
pi@raspberrypi ~>
```

scheduling policy

scheduling priority

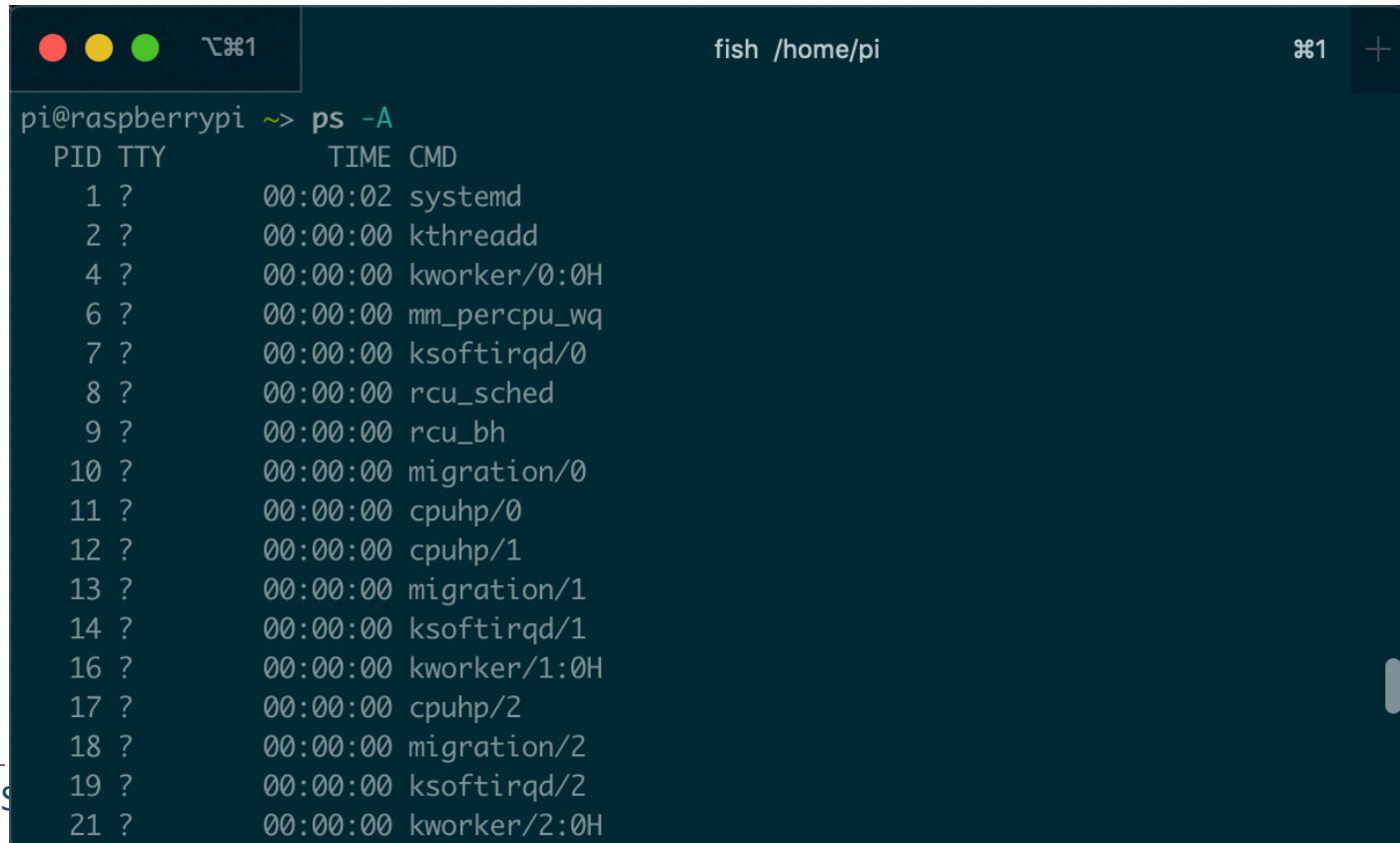
SCHED\_OTHER  
SCHED\_FIFO  
SCHED\_RR  
SCHED\_BATCH

min/max priority : 0/0  
min/max priority : 1/99  
min/max priority : 1/99  
min/max priority : 0/0

# More about command

- \$ ps -A or \$ ps -ef

show all information for all process

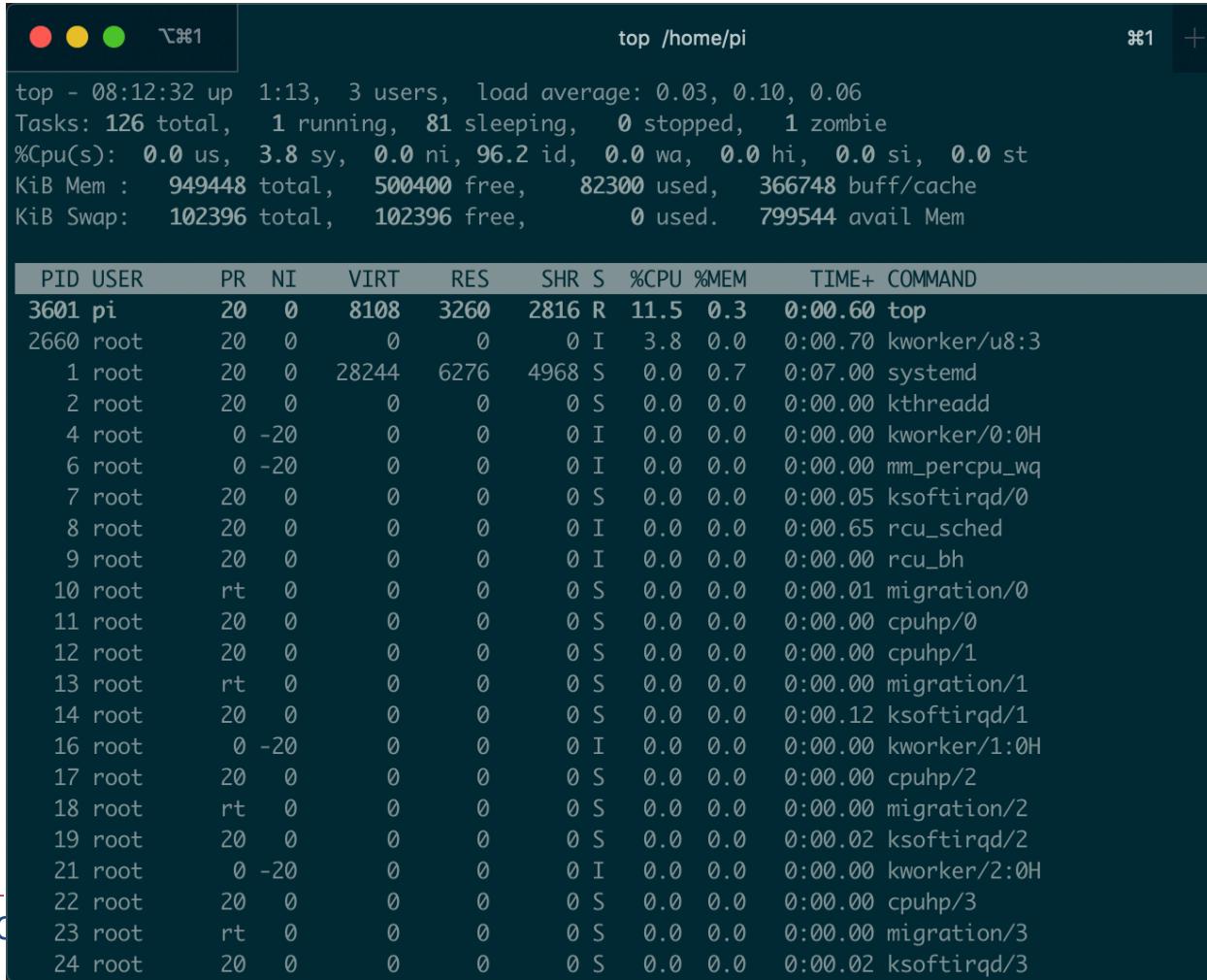


A screenshot of a terminal window titled "fish /home/pi". The window shows the command "pi@raspberrypi ~> ps -A" followed by a list of processes. The output is as follows:

PID	TTY	TIME	CMD
1	?	00:00:02	systemd
2	?	00:00:00	kthreadd
4	?	00:00:00	kworker/0:0H
6	?	00:00:00	mm_percpu_wq
7	?	00:00:00	ksoftirqd/0
8	?	00:00:00	rcu_sched
9	?	00:00:00	rcu_bh
10	?	00:00:00	migration/0
11	?	00:00:00	cpuhp/0
12	?	00:00:00	cpuhp/1
13	?	00:00:00	migration/1
14	?	00:00:00	ksoftirqd/1
16	?	00:00:00	kworker/1:0H
17	?	00:00:00	cpuhp/2
18	?	00:00:00	migration/2
19	?	00:00:00	ksoftirqd/2
21	?	00:00:00	kworker/2:0H

# More about command

- \$ top: displays processor activity



The screenshot shows the output of the top command on a Linux system. The title bar indicates "top /home/pi". The top section provides system statistics: 126 tasks total, 1 running, 81 sleeping, 0 stopped, 1 zombie. CPU usage is at 0.0 us, 3.8 sy, 0.0 ni, 96.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st. Memory usage shows 949448 KiB total memory, 500400 KiB free, 82300 KiB used, 366748 KiB buff/cache, and 102396 KiB total swap, 102396 KiB free, 0 KiB used, 799544 KiB avail Mem.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3601	pi	20	0	8108	3260	2816	R	11.5	0.3	0:00.60	top
2660	root	20	0	0	0	0	I	3.8	0.0	0:00.70	kworker/u8:3
1	root	20	0	28244	6276	4968	S	0.0	0.7	0:07.00	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
7	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd/0
8	root	20	0	0	0	0	I	0.0	0.0	0:00.65	rcu_sched
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.01	migration/0
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/1
13	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/1
14	root	20	0	0	0	0	S	0.0	0.0	0:00.12	ksoftirqd/1
16	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/1:0H
17	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/2
18	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/2
19	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/2
21	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/2:0H
22	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/3
23	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/3
24	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/3

# More about command

- \$ htop: an advance top

The screenshot shows the htop terminal application running on a Linux system. The title bar indicates it's running as user pi. The interface includes a header with system status (Tasks: 53, Load average: 0.02 0.07 0.05, Uptime: 01:15:32) and a footer with navigation keys.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3622	pi	20	0	5472	2944	2468	R	1.3	0.3	0:00.28	htop
1869		20	0	833M	17908	15440	S	0.7	1.9	0:00.21	/usr/bin/docker daemon -H fd://
1		20	0	28244	6276	4968	S	0.0	0.7	0:07.00	/sbin/init splash
98		20	0	26920	4508	4096	S	0.0	0.5	0:00.83	/lib/systemd/systemd-journald
129		20	0	14596	3192	2548	S	0.0	0.3	0:00.66	/lib/systemd/systemd-udevd
300		20	0	17276	3620	3228	S	0.0	0.4	0:00.00	/lib/systemd/systemd-timesyncd
268		20	0	17276	3620	3228	S	0.0	0.4	0:00.13	/lib/systemd/systemd-timesyncd
302		20	0	5292	2404	2168	S	0.0	0.3	0:00.05	/usr/sbin/thd --triggers /etc/triggerhappy/triggers.d/ --socket
303		20	0	7380	4212	3776	S	0.0	0.4	0:00.38	/lib/systemd/systemd-logind
304		20	0	6392	2844	2544	S	0.0	0.3	0:00.37	avahi-daemon: running [raspberrypi.local]
305		20	0	6612	3696	3132	S	0.0	0.4	0:01.29	/usr/bin/dbus-daemon --system --address=systemd: --nofork --no
307		20	0	6392	316	16	S	0.0	0.0	0:00.00	avahi-daemon: chroot helper
309		20	0	9992	3988	3624	S	0.0	0.4	0:00.05	/sbin/wpa_supplicant -u -s -0 /run/wpa_supplicant
384		20	0	23756	2476	2172	S	0.0	0.3	0:00.03	/usr/sbin/rsyslogd -n
385		20	0	23756	2476	2172	S	0.0	0.3	0:00.00	/usr/sbin/rsyslogd -n
386		20	0	23756	2476	2172	S	0.0	0.3	0:00.05	/usr/sbin/rsyslogd -n
316		20	0	23756	2476	2172	S	0.0	0.3	0:00.12	/usr/sbin/rsyslogd -n
328		20	0	5296	2440	2240	S	0.0	0.3	0:00.02	/usr/sbin/cron -f
344		20	0	27600	1312	1192	S	0.0	0.1	0:00.11	/usr/sbin/rngd -r /dev/hwrng

# Process in Linux

---

- Process control block (PCB)

- State
- Identifiers
- Scheduling info
- File system 
- Virtual memory
- Process specific context
- ...



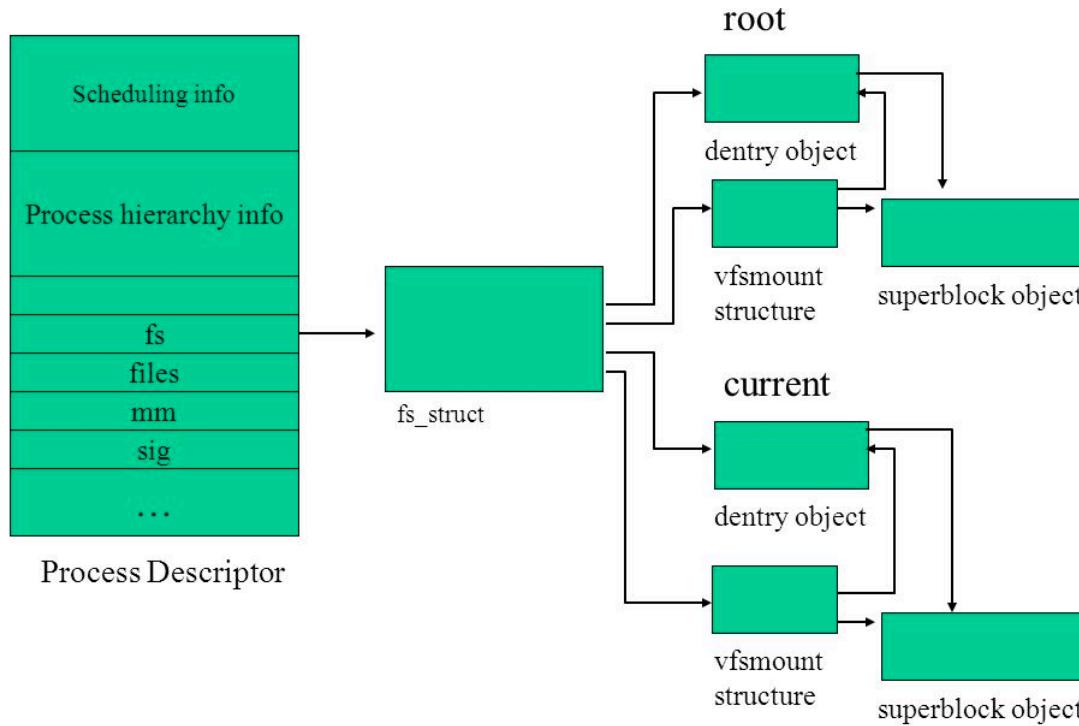
# Linux Process Control Block (cont')

- Files

- **fs\_struct**

<https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1525>

- ▶ file system information: root directory, current directory



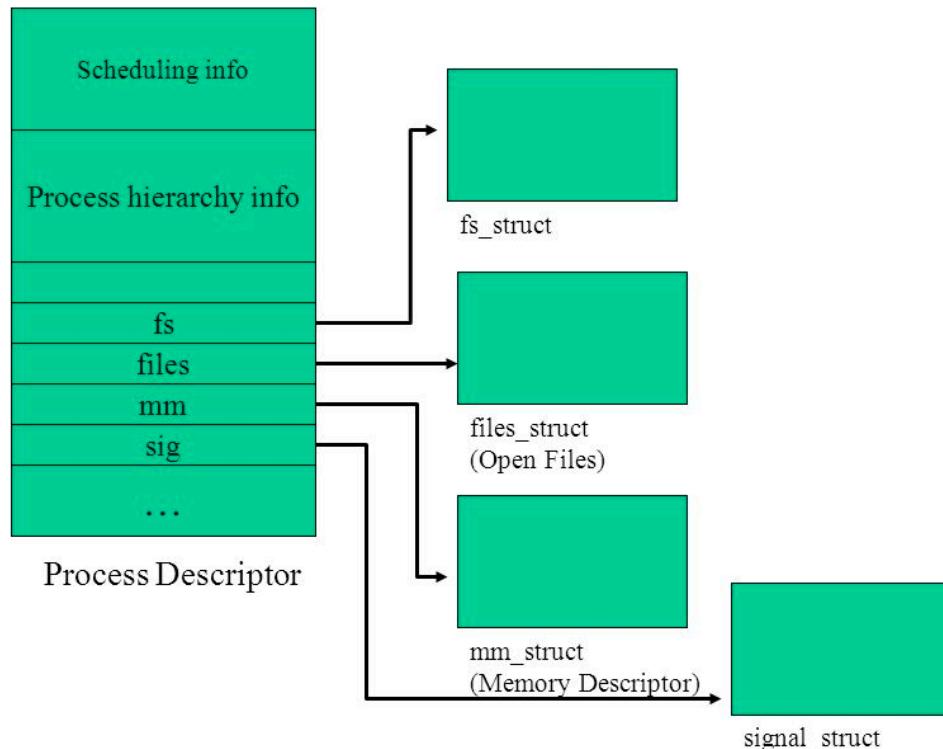
# Linux Process Control Block (cont')

- 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

```
pi@raspberrypi ~> ps
  PID TTY      TIME CMD
 947 pts/0    00:00:00 bash
 966 pts/0    00:00:02 fish
1209 pts/0    00:00:00 ps
pi@raspberrypi ~> lsof -p 947
COMMAND PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    947  pi cwd   DIR  179,7     4096 1572867 /home/pi
bash    947  pi rtd   DIR  179,7     4096      2 /
bash    947  pi txt   REG  179,7  912712  524329 /bin/bash
bash    947  pi mem   REG  179,7  38560 1445488 /lib/arm-linux-gnueabihf/libnss_files-2.24.so
bash    947  pi mem   REG  179,7  38588 1445507 /lib/arm-linux-gnueabihf/libnss_nis-2.24.so
bash    947  pi mem   REG  179,7  71604 1445594 /lib/arm-linux-gnueabihf/libnsl-2.24.so
bash    947  pi mem   REG  179,7  26456 1445612 /lib/arm-linux-gnueabihf/libnss_compat-2.24.so
bash    947  pi mem   REG  179,7 1679776 670175 /usr/lib/locale/locale-archive
bash    947  pi mem   REG  179,7 1234700 1445500 /lib/arm-linux-gnueabihf/libc-2.24.so
bash    947  pi mem   REG  179,7   9800 1445460 /lib/arm-linux-gnueabihf/libdl-2.24.so
bash    947  pi mem   REG  179,7 124808 1445519 /lib/arm-linux-gnueabihf/libtinfo.so.5.9
bash    947  pi mem   REG  179,7   21868 144001 /usr/lib/arm-linux-gnueabihf/libarmmem.so
bash    947  pi mem   REG  179,7 138576 1445547 /lib/arm-linux-gnueabihf/ld-2.24.so
bash    947  pi mem   REG  179,7  26262 145746 /usr/lib/arm-linux-gnueabihf/gconv/gconv-modules.cache
bash    947  pi  0u  CHR  136,0     0t0      3 /dev/pts/0
bash    947  pi  1u  CHR  136,0     0t0      3 /dev/pts/0
bash    947  pi  2u  CHR  136,0     0t0      3 /dev/pts/0
bash    947  pi 255u  CHR  136,0     0t0      3 /dev/pts/0
pi@raspberrypi ~>
```

fish /home/pi

All files opened by bash

File descriptor, size, name, location, ...

# Process in Linux

---

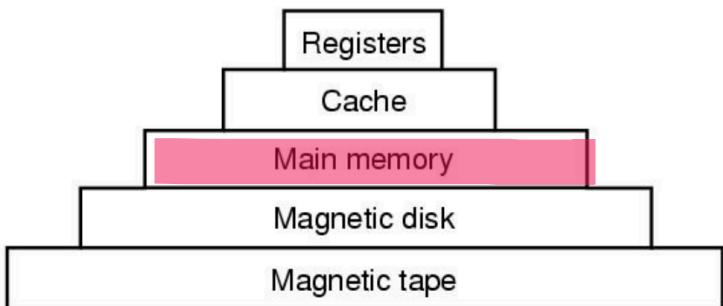
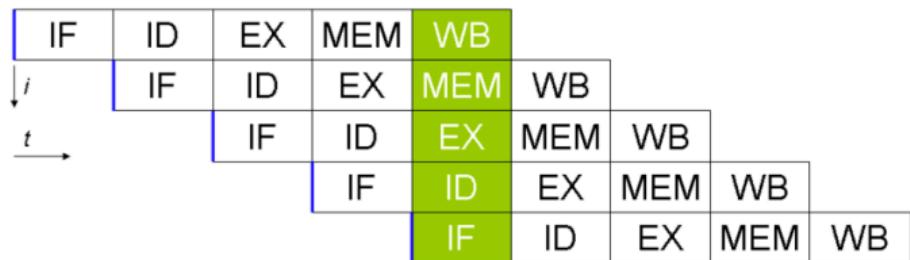
- Process control block (PCB)

- 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 <https://elixir.bootlin.com/linux/v4.2/source/include/linux/sched.h#L1408>
  - Process specific context
  - ...



# \$pmap: memory mapping

Memory address

size

Read/write/  
execution  
permission

Lib/execut  
ion file

All memory accessed by bash

memory used by  
process bash

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
76bf6000	68K	r-x--	libnsl-2.24.so
76bt7000	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
76c0b000	60K	----	libnss_compat-2.24.so
76c1f000	4K	r----	libnss_compat-2.24.so

# \$pmap: memory mapping

Memory address

size

Read/write/  
execution  
permission

Lib/execut  
ion file

Code loaded by bash

Constant/static variable  
loaded by bash

Data loaded by bash

```
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
76bf6000   68K r-x-- libnsl-2.24.so
76b77000   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
76c1b000   60K ----- libnss_compat-2.24.so
76c1f000    4K r---- libnss_compat-2.24.so
```

# Bash in memory

## \$pmap: memory mapping

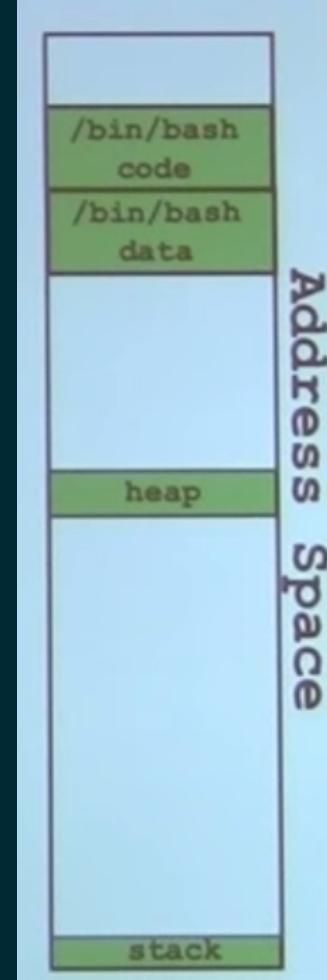
Memory address

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
76bf6000   68K r-x-- libnsl-2.24.so
76bt7000   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
76c0b000   60K ----- libnss_compat-2.24.so
76c1f000     4K r---- libnss_compat-2.24.so
```



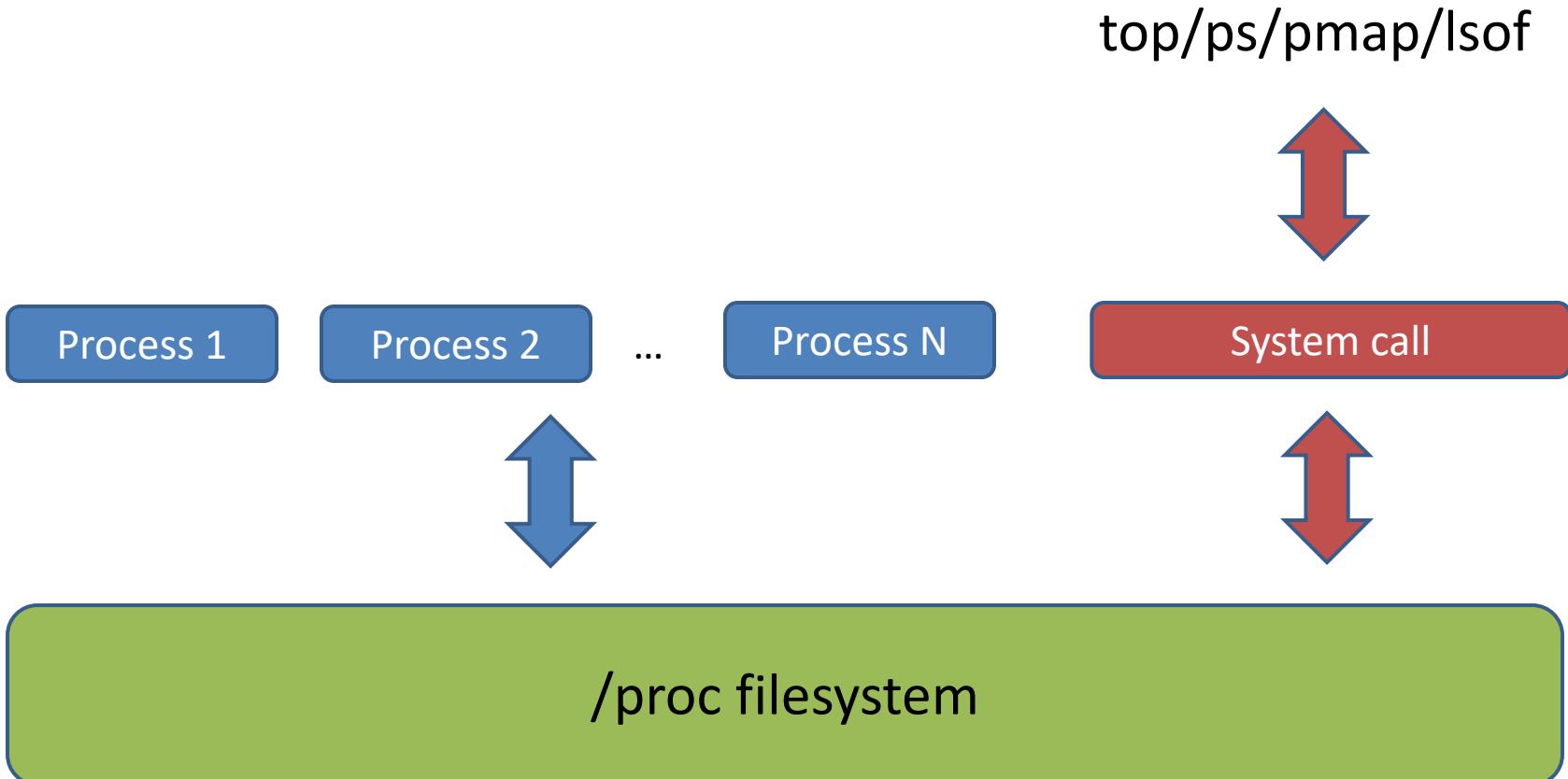
The diagram illustrates the memory layout for the /bin/bash process. It shows the code and data segments of the binary at the top, followed by the heap (green) and stack (green) regions at the bottom. Red arrows point from the pmap output to specific memory blocks in the diagram, indicating the correspondence between the memory dump and the actual memory structure.

# Aside: the /proc/ filesystem

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

```
● ● ● ✎ 1 fish /home/pi  
pi@raspberrypi ~> ps  
PID TTY          TIME CMD  
947 pts/0        00:00:00 bash  
966 pts/0        00:00:24 fish  
3707 pts/0       00:00:00 ps  
pi@raspberrypi ~> ls /proc/947/  
autogroup      cpuset   io        mounts      pagemap      smaps      task/  
auxv          cwd@    latency   mountstats  personality  smaps_rollup  timerslack_ns  
cgroup         environ  limits    net/       projid_map  stack       uid_map  
clear_refs     exe@    map_files/ ns/        root@       stat        wchan  
cmdline        fd/     maps      oom_adj    sched       statm  
comm           fdinfo/ mem      oom_score  schedstat  status  
coredump_filter gid_map  mountinfo oom_score_adj setgroups  syscall  
pi@raspberrypi ~> |  
  
Real time information for  
specific process stored in /proc
```

# 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

```
pi@raspberrypi ~> pstree -p
fish /home/pi
pi@raspberrypi ~> pstree -p
systemd(1)─avahi-daemon(304)─avahi-daemon(307)
                     └─bluealsa(674)─{bactl}(683)
                           ├─{gdbus}(685)
                           └─{gmain}(684)
                     └─bluetoothd(659)
                     └─cron(328)
                     └─dbus-daemon(305)
                     └─dhcpcd(351)
                     └─docker(1853)─{docker}(1854)
                           ├─{docker}(1855)
                           ├─{docker}(1856)
                           ├─{docker}(1860)
                           └─{docker}(1869)
                     └─hciattach(649)
                     └─lightdm(458)─Xorg(476)─{InputThread}(488)
                           ├─{llvmpipe-0}(482)
                           ├─{llvmpipe-1}(483)
                           ├─{llvmpipe-2}(484)
                           └─{llvmpipe-3}(485)
                     └─lightdm(491)─lxsession(510)─lxpanel(595)─sh(727)
                           ├─{gdbus}(633)
                           ├─{gmain}(632)
                           └─{menu-cache-io}(762)
                     └─lxpolkit(593)─{gdbus}(610)
                           └─{gmain}(609)
```

# 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

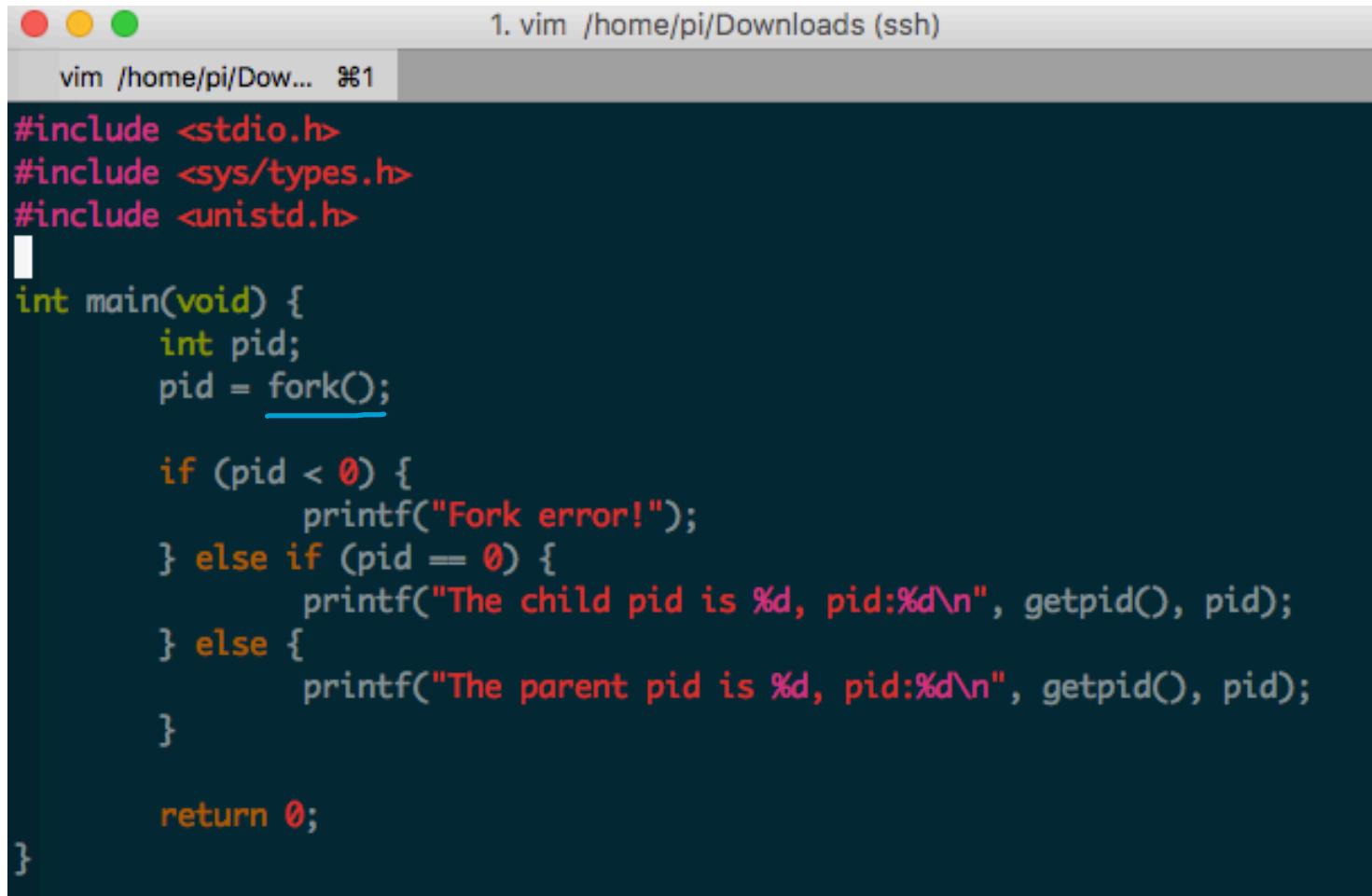
---

- 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>



```
#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>

```
#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;
}

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

# Fork() system call

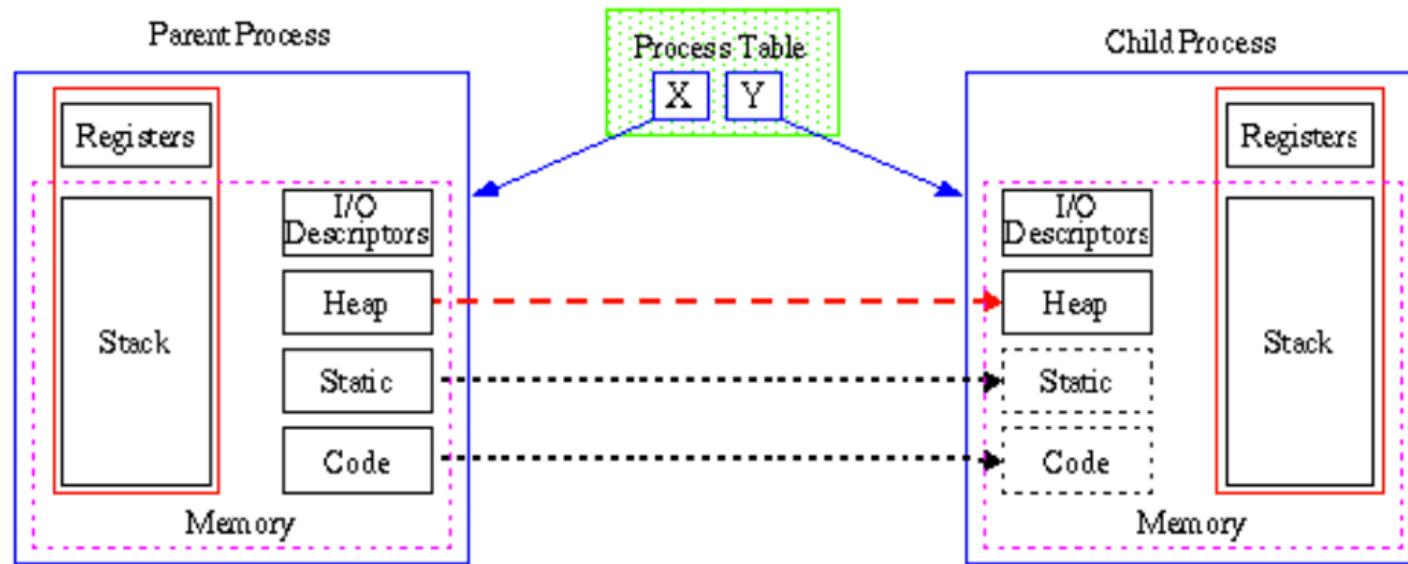
---

- 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.



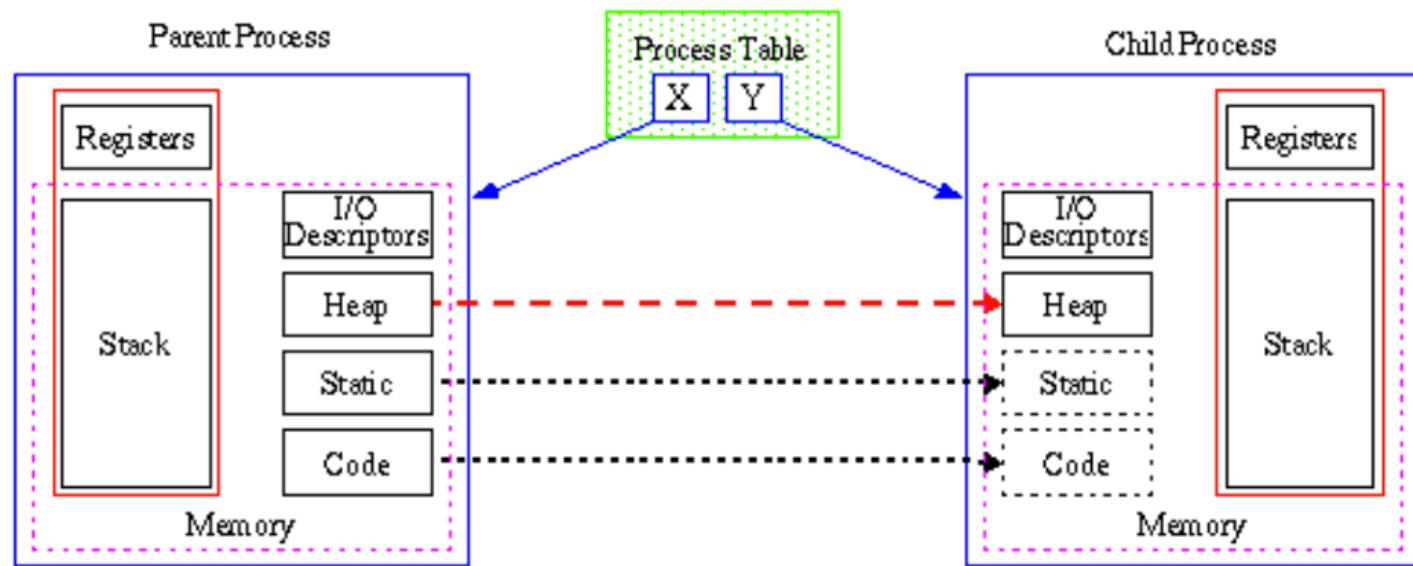
# Parent process and child process

- 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

```
1. fish /home/pi/Downloads (ssh)
fish /home/pi/Dow... %1
pi@raspberrypi ~/Downloads> pstree -p
systemd(1)─ avahi-daemon(303)─ avahi-daemon(363)
          └─ bluealsa(661)─ {bactl}(676)
                         ├ {gdbus}(678)
                         └ {gmain}(677)
          ├ bluetoothd(649)
          ├ cron(300)
          ├ dbus-daemon(275)
          ├ dhcpcd(354)
          ├ hciattach(642)
          ├ lightdm(398)─ Xorg(441)─ {InputThread}(453)
                         ├ {llvmpipe-0}(447)
                         ├ {llvmpipe-1}(448)
                         ├ {llvmpipe-2}(449)
                         └ {llvmpipe-3}(450)
          ├ lightdm(456)─ lxsession(473)─ lxpanel(556)─ sh(695)
                         ├ {gdbus}(597)
                         ├ {gmain}(596)
                         ├ {menu-cache-io}(732)
                         └─ lxpolk(553)─ {gdbus}(569)
                           └ {gmain}(568)
                         └─ openbox(552)
                         └─ pcmanfm(558)─ {gdbus}(592)
                           └ {gmain}(591)
                         └─ ssh-agent(522)
                           ├ {gdbus}(528)
                           └ {gmain}(527)
                         └─ {gdbus}(460)
                           └ {gmain}(459)
          └─ {gdbus}(419)
             └ {gmain}(416)
          └─ login(407)─ bash(586)
          └─ menu-cached(622)─ {gdbus}(628)
                         └ {gmain}(627)
          └─ packagekitd(1342)─ {gdbus}(1344)
```

pstree is a Linux command that shows the running processes as a tree



# Fork() example

A screenshot of a terminal window titled "1. vim /home/pi/Downloads". The code in the editor is:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Hello world!\n");
    return 0;
}
```

Two arrows point from the "fork();" line to two speech bubbles: a blue one labeled "Parent" and a red one labeled "Child".

pi@raspberrypi ~\$ ./a.o  
Hello world!  
Hello world!

A screenshot of a terminal window titled "1. vim /home/pi/Downloads". The code in the editor is:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello world!\n");
    return 0;
}
```

To the right of the code is a tree diagram with a root node. Blue arrows point from the root to three child nodes, which then have their own blue arrows pointing to four and five leaf nodes respectively, illustrating the hierarchical nature of multiple fork operations.

pi@raspberrypi ~\$ ./a.o  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!



# Issues with fork()

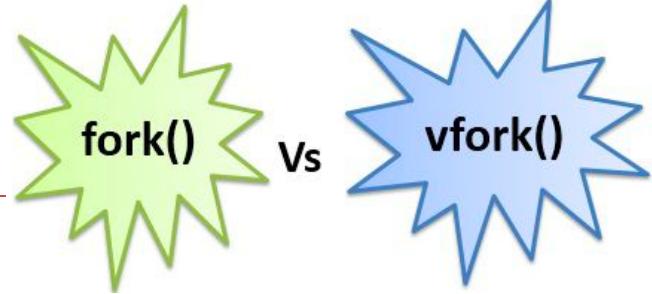
---

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.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Hello world!\n");
    return 0;
}
```

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

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    execl("/bin/echo", "echo", "Hello", NULL);
    printf("Hello world!\n");
    return 0;
}
```

pi@raspberrypi ~/Downloads> ./b.o  
Hello  
pi@raspberrypi ~/Downloads>

# 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

```
#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
```

```
#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

---

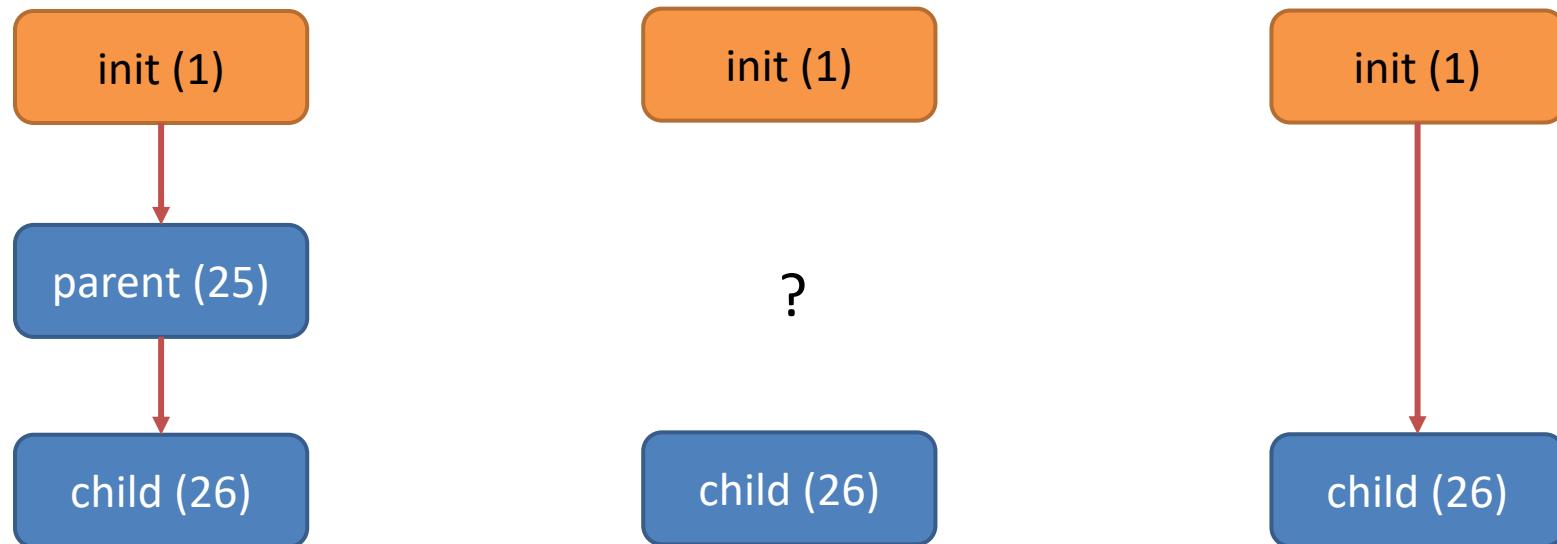
- What is process?
  - Process vs Program
  - Linux Process Control Block
- Process related System calls
  - Fork
  - Exec
  - Wait
- Orphan and Zombie process



# Orphan process

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

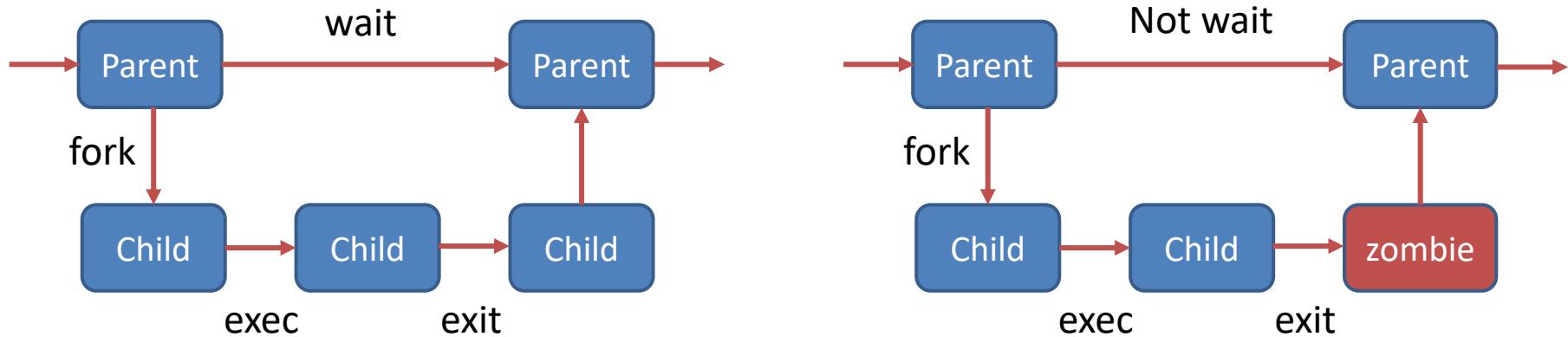
- When a parent process dies, child process becomes an orphan process.
- The init process (pid = 1) becomes the parent of the orphan processes.
- Parent stops first



# 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

```
#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;
}
```

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

```
#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> ./zombie.o
parent process pid:3857
child process pid:3858
```

The child process  
is zombie process

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

After 10 seconds

Kennesaw State

# 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

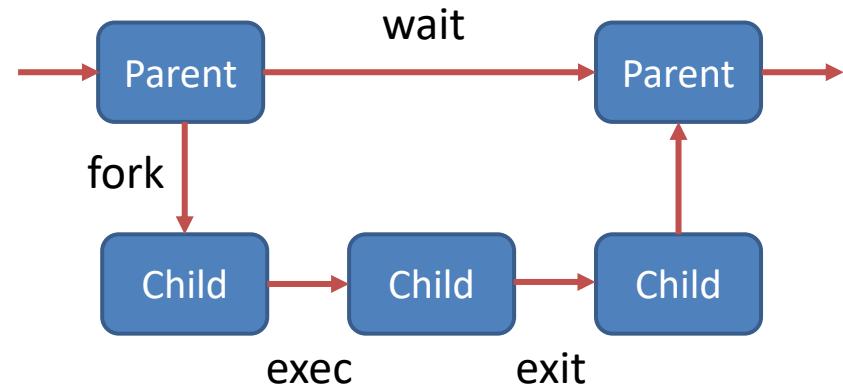
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

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.

