

HPC & Parallel Programming

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
- Process on Distributed OSes



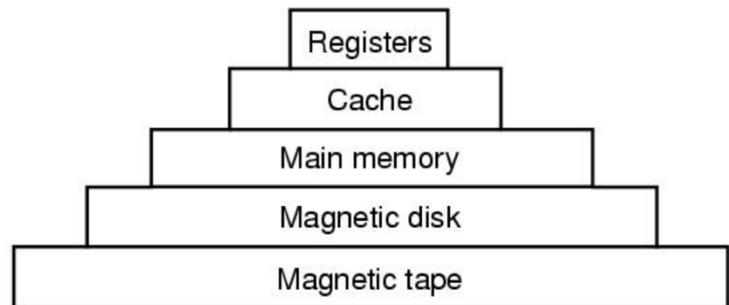
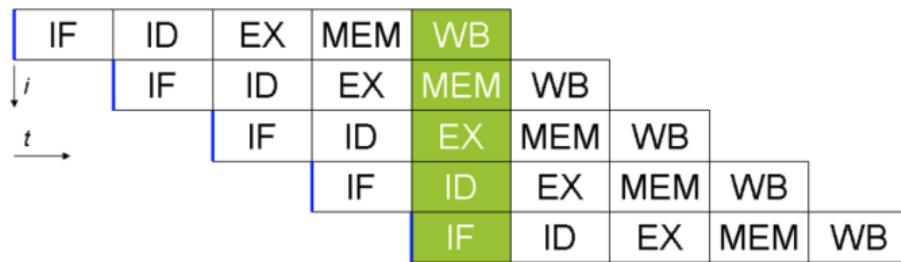
Process

- 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 \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

The image shows two screenshots side-by-side. On the left is a Mac OS X Applications window showing a list of installed applications like Chess, Contacts, and Google Chrome. A blue callout bubble points to Google Chrome with the text: "Program: An executable file in long-term storage". On the right is a Mac OS X Activity Monitor window showing a list of running processes. A red dashed box highlights multiple "Google Chrome Helper" entries, and a blue callout bubble points to one of them with the text: "Process: The running instantiation of a program, stored in RAM". Below the process list is a summary bar showing CPU usage and threads/processes.

Program:
An executable file in
long-term storage

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

One-to-many
relationship between
program and processes

Process Name	User	PID
identityservicesd	admin	270
iconservicesagent	admin	304
icdd	admin	279
Google Chrome Helper	admin	813
Google Chrome Helper	admin	795
Google Chrome Helper	admin	506
Google Chrome Helper	admin	1839
Google Chrome Helper	admin	1527
Google Chrome Helper	admin	807
Google Chrome Helper	admin	1549
Google Chrome Helper	admin	1925
Google Chrome Helper	admin	2107
Google Chrome Helper	admin	507
Google Chrome Helper	admin	1560
Google Chrome Helper	admin	2102
Google Chrome Helper	admin	2141
Google Chrome	admin	496
frontend	admin	263
fsck_udpd	admin	746
ClientXPCService	admin	494

CPU Usage:	3.65%	CPU LOAD	Threads: 1246
	6.11%		Processes: 350
	90.24%		



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



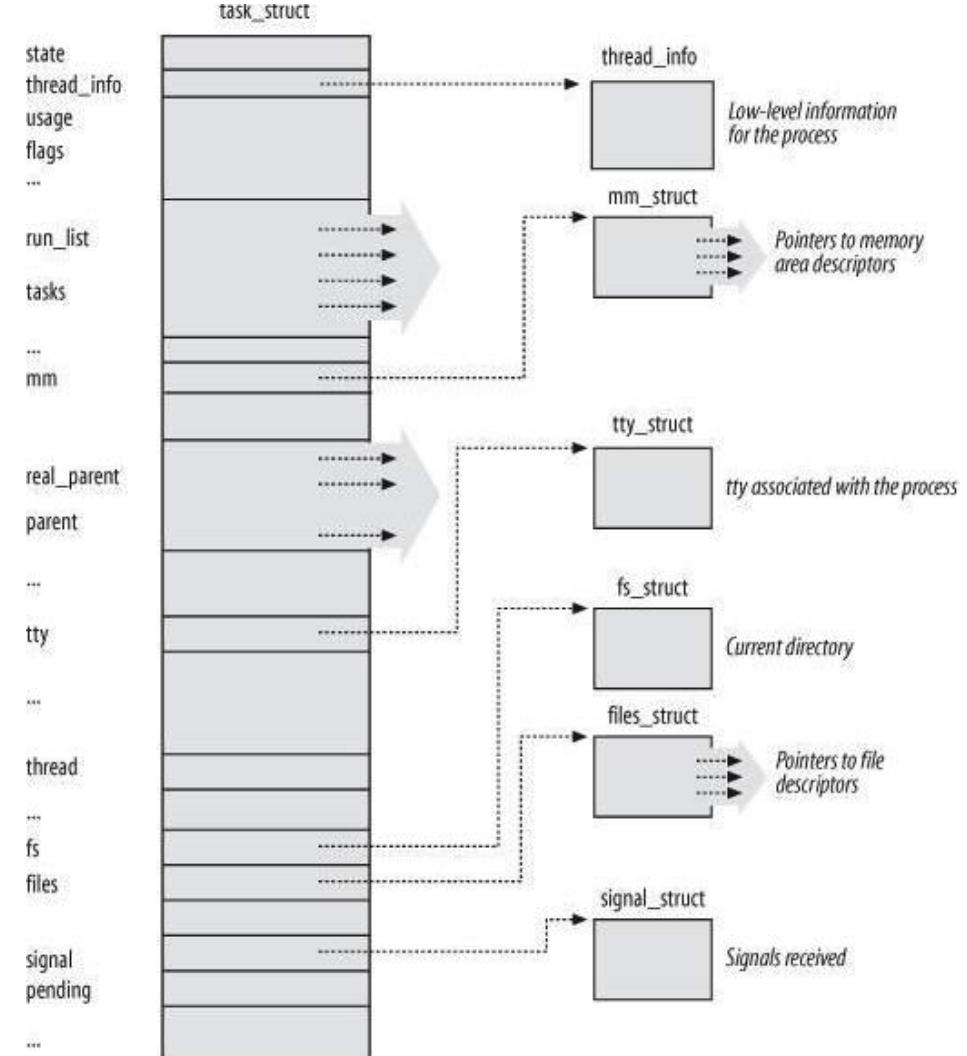
- Driving license
 - ID
 - Name
 - Address
 - Birth
 - Time
 - ...
- Process control block (PCB)
 - State
 - Identifiers
 - Scheduling info
 - File system
 - Virtual memory
 - Process specific context
 - ...



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)

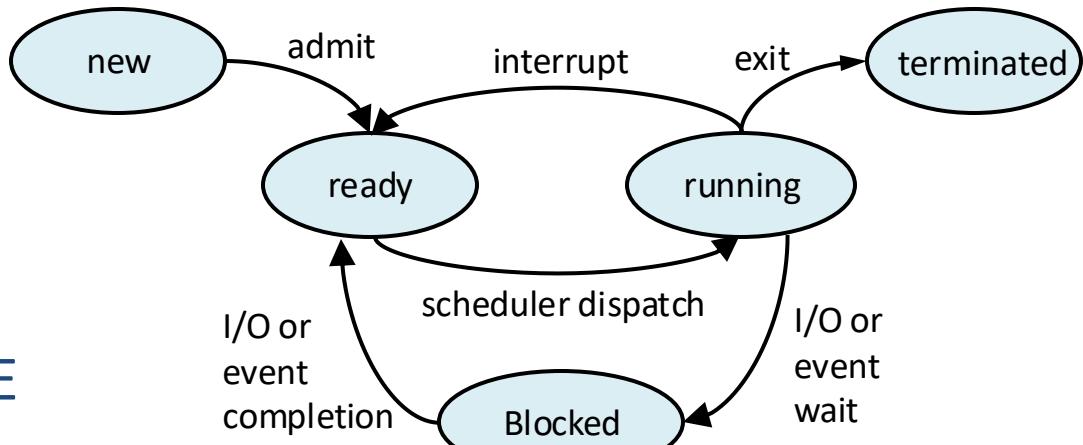
- State ←
- Identifiers
- Scheduling info
- File system
- Virtual memory
- Process specific context
- ...



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

```
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



\$ 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



```
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



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')

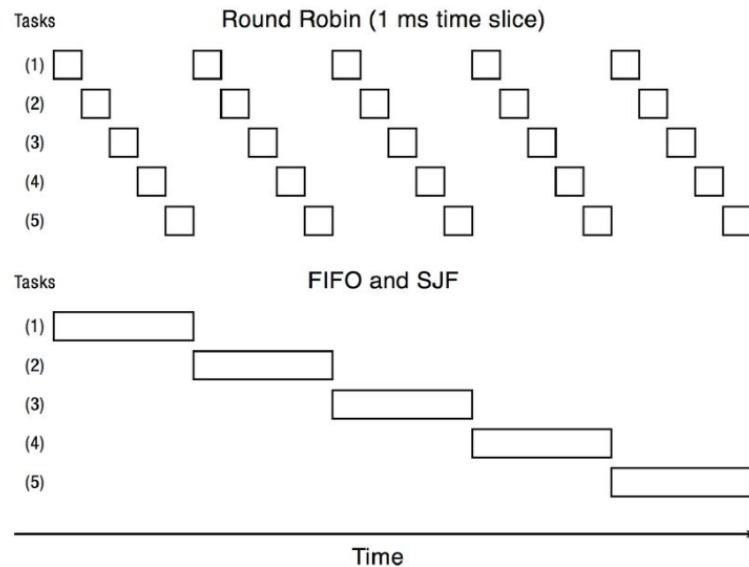
- Scheduling information
 - rt_priority



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

\$ 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



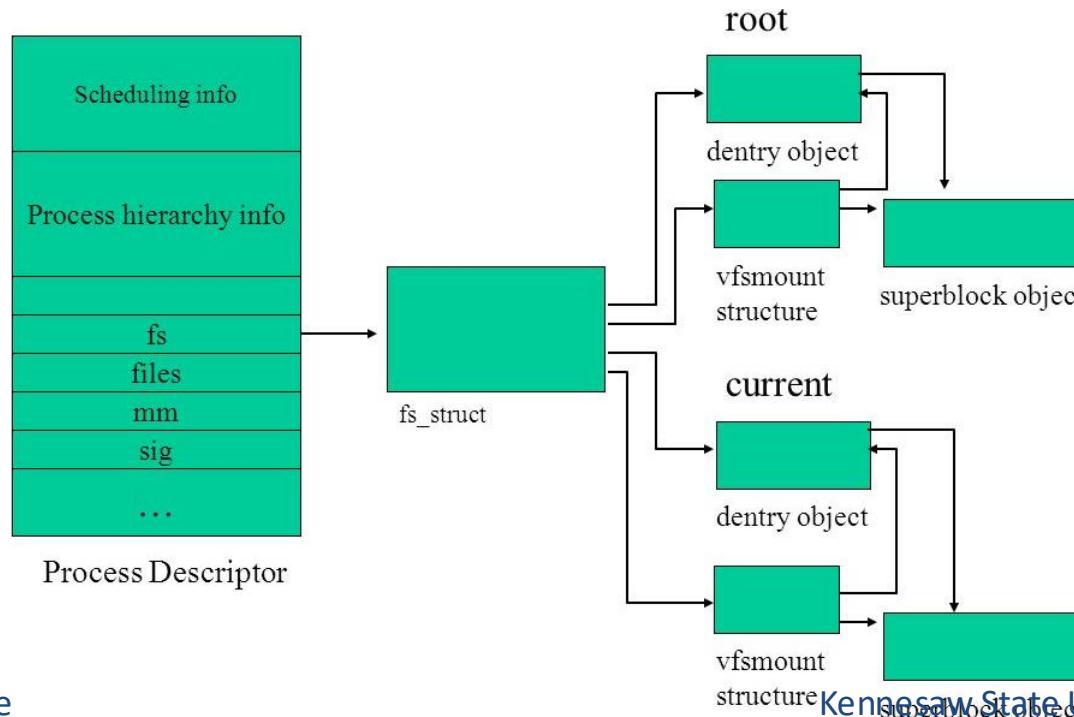
Process in Linux

- 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

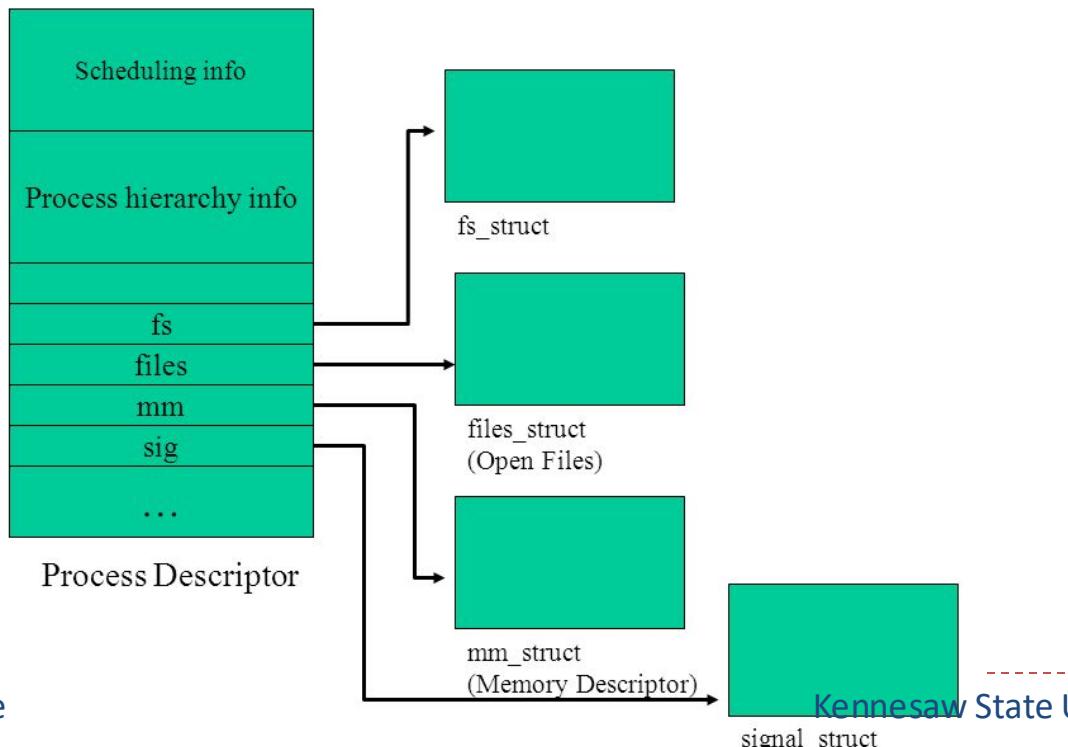


Linux Process Control Block (cont')

- 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

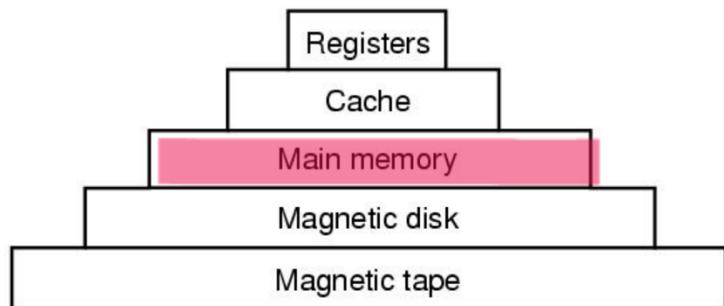
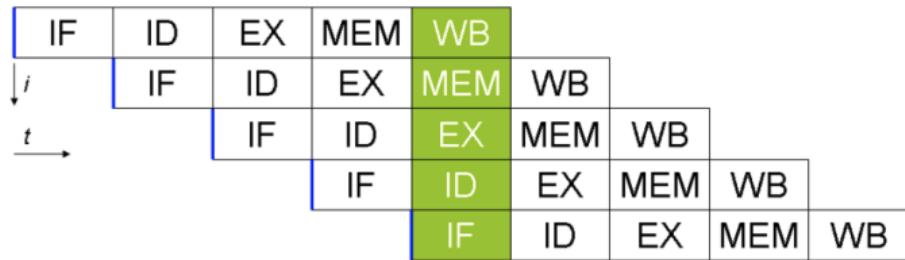
- Process control block (PCB)

- 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



\$pmap: memory mapping

```
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
76c0000   68K r-x-- libnsl-2.24.so
76b17000   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     8K ----- libnss_compat-2.24.so
76c1f000     4K r---- libnss_compat-2.24.so
```

Memory address

size

Read/write/execution permission

Lib/execution file

All memory accessed by bash

memory used by process bash

Kennesaw State University

\$pmap: memory mapping

```
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
76c0000  68K r-x-- libnsl-2.24.so
76b17000  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
```

Memory address

size

Read/write/execution permission

Lib/execution file

Code loaded by bash

Constant/static variable loaded by bash

Data loaded by bash

Bash in memory

\$pmap: memory mapping

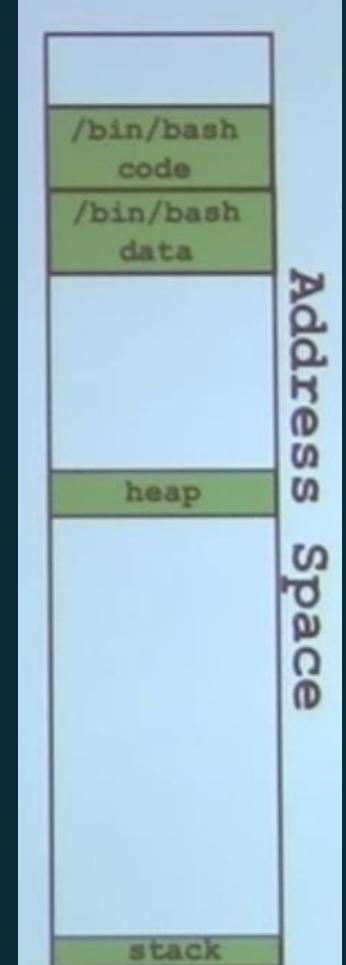
Memory address

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
76c0000  68K r-x-- libnsl-2.24.so
76b17000  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
80K ----- libnss_compat-2.24.so
76c1f000   4K r---- libnss_compat-2.24.so
```



The diagram illustrates the memory layout of the /bin/bash process. It shows four main regions: /bin/bash code (read-only), /bin/bash data (read-write), heap (read-write), and stack (read-write). Red arrows point from the corresponding columns in the pmap output to these regions.

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

Download: wget Raw-file-URL

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

```
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)
                           └─ lxpolk(593)─ {gdbus}(610)
                                         └ {gmain}(609)

fish /home/pi
```

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



```
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

```
#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
```

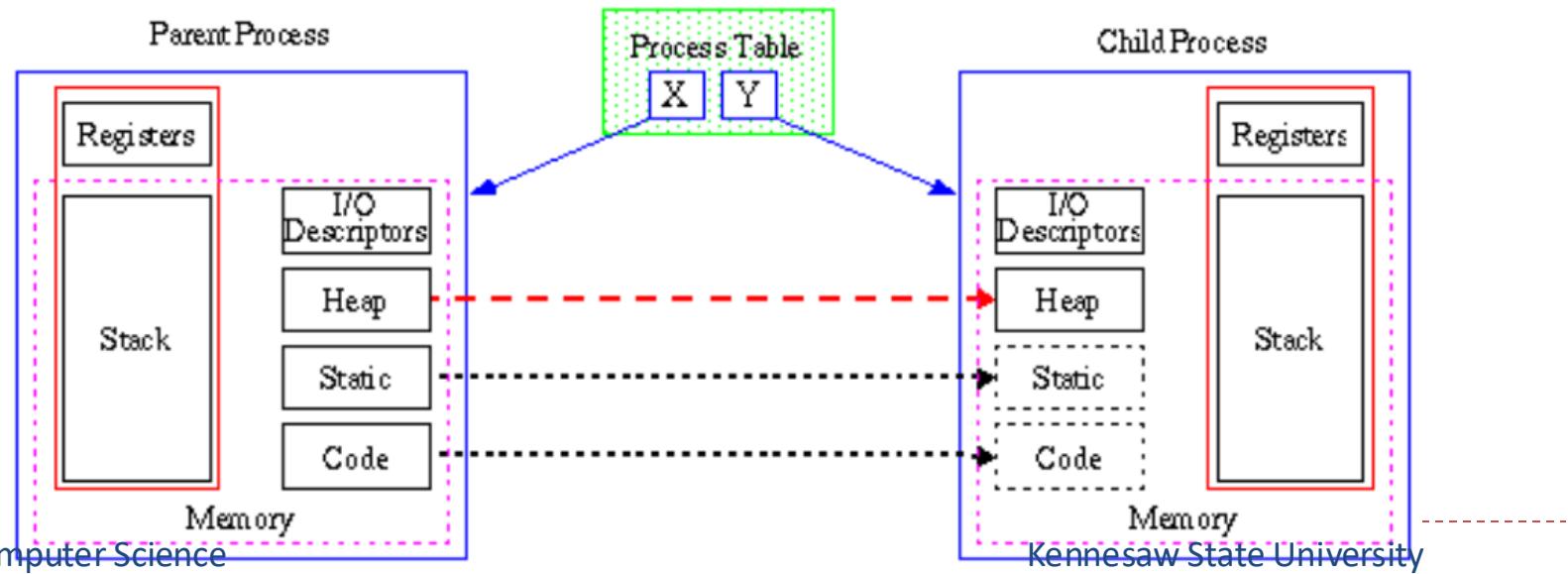
Fork() system call

- 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.



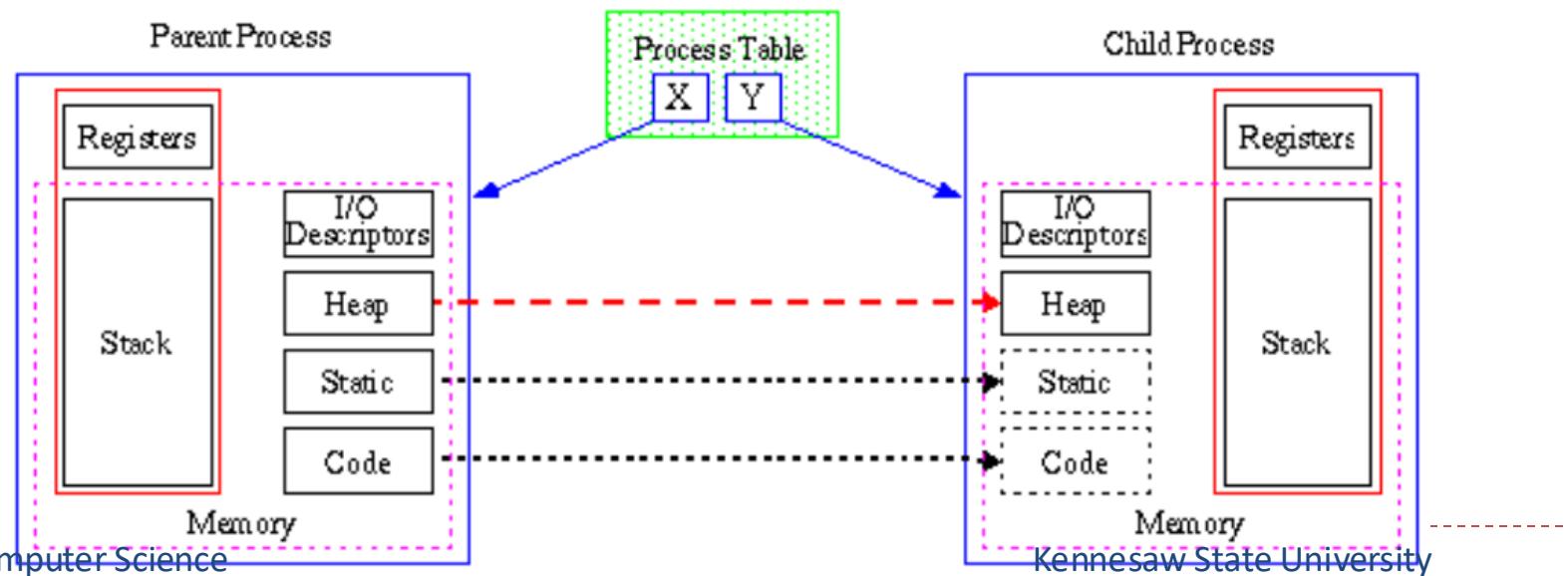
Parent process and child process

- 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.



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 speech bubbles are overlaid on the code: a blue one labeled "Parent" pointing to the first fork(), and a red one labeled "Child" pointing to the second fork(). A vertical blue arrow points down from the first fork() to the printf() line, and a vertical red arrow points down from the second fork() to the same line.

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 identical to the first screenshot:

```
#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 diagram illustrating the execution flow. It shows a root node with three blue arrows pointing down to three child nodes. Each child node has two red arrows pointing down to four grandchild nodes each, forming a tree structure.

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



Fork() example

- How many a it will output?

```
#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

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

Kennesaw State University 13,3

All

Fork() example

- How many a it will output?

```
#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?

```
#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

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

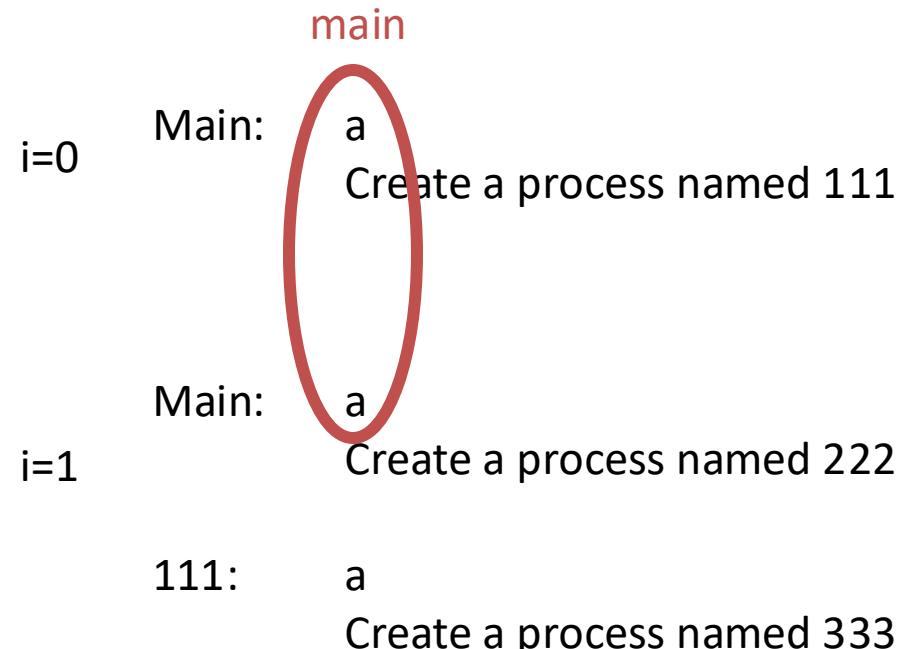
Kennesaw State University 13.3 All

Fork() example

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

```
#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;
}
```

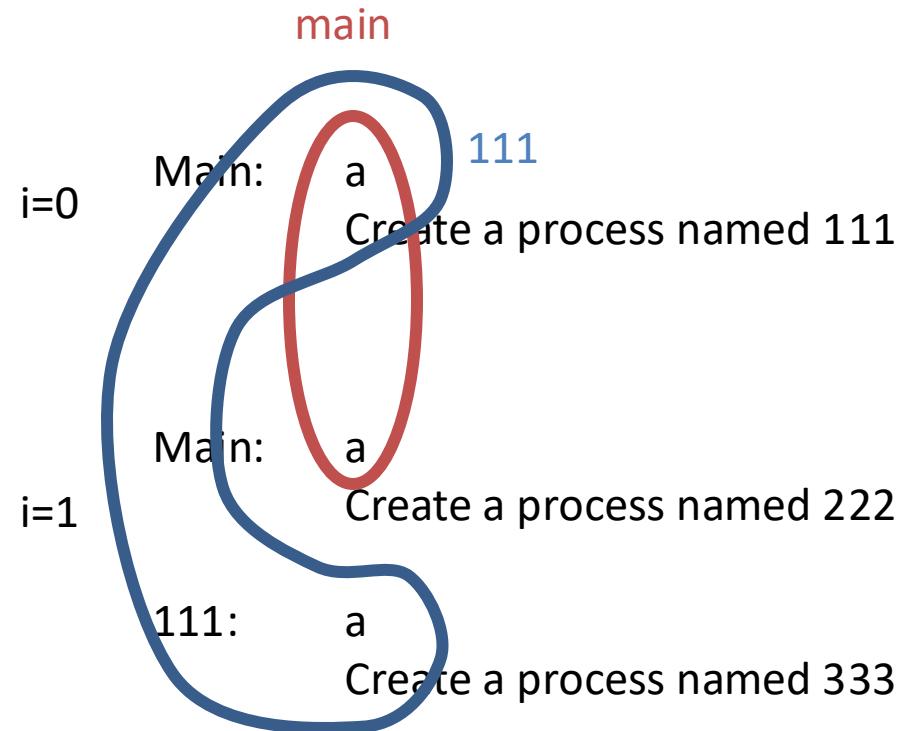


Fork() example

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

```
#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;
}
```

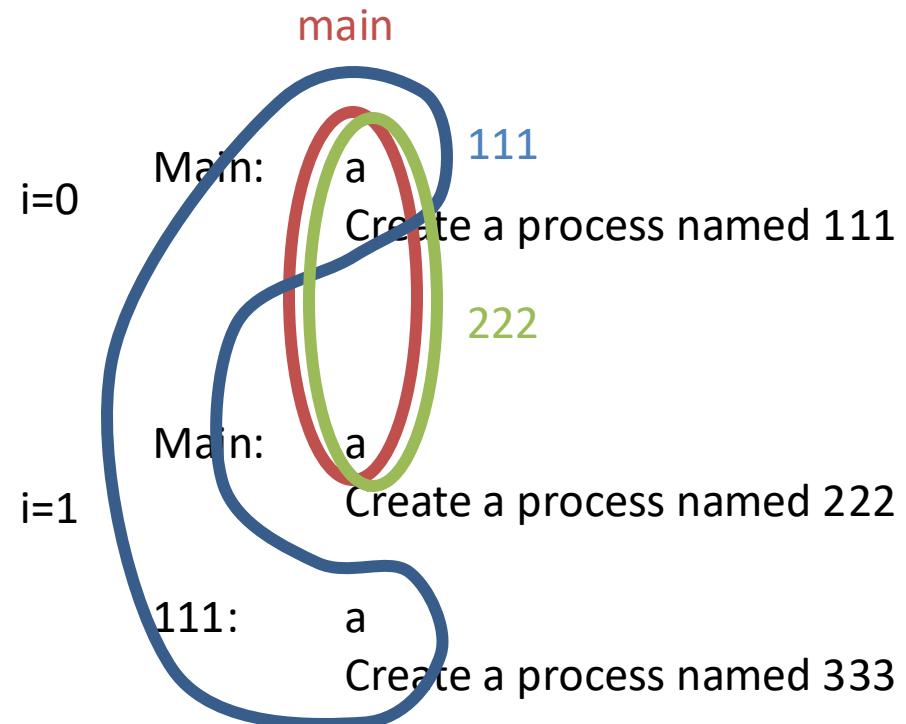


Fork() example

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

```
#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;
}
```

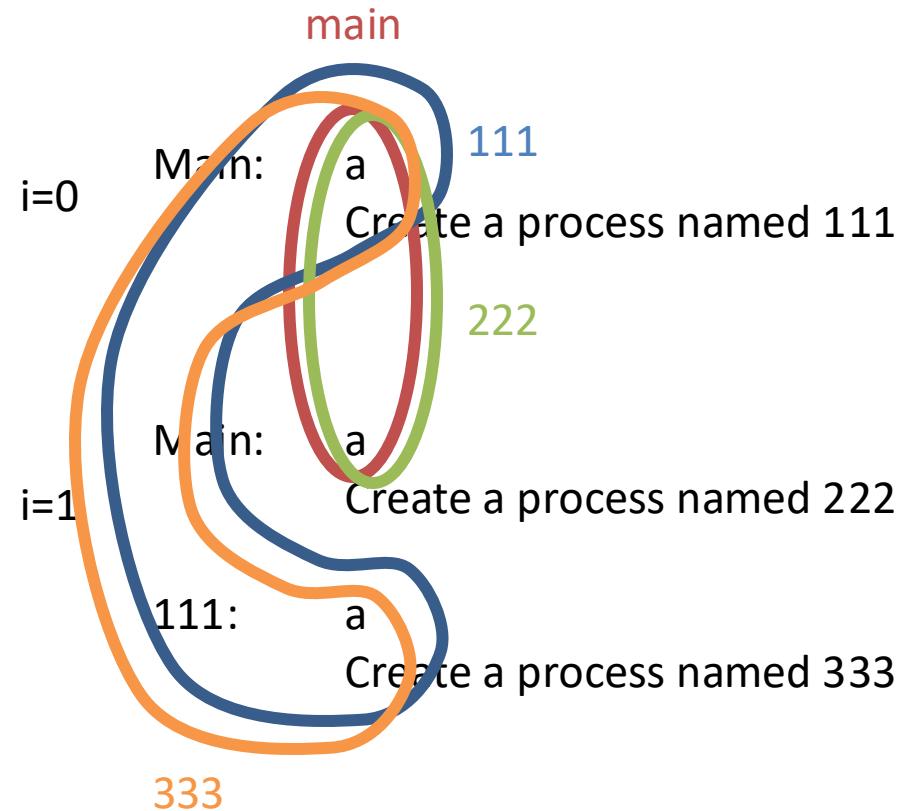


Fork() example

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

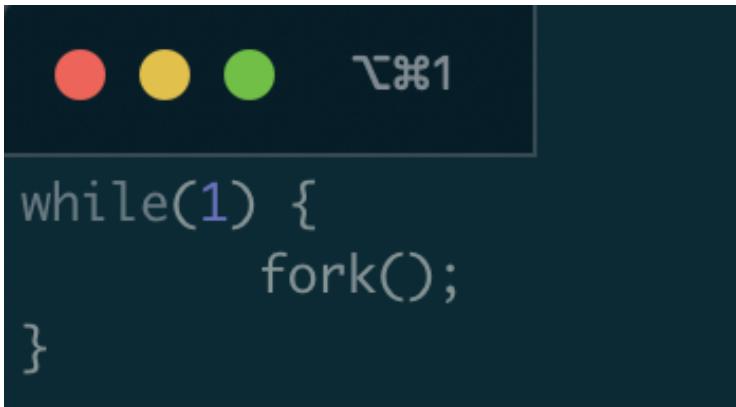
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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?



```
● ● ● ↵#1  
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/CS7172/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



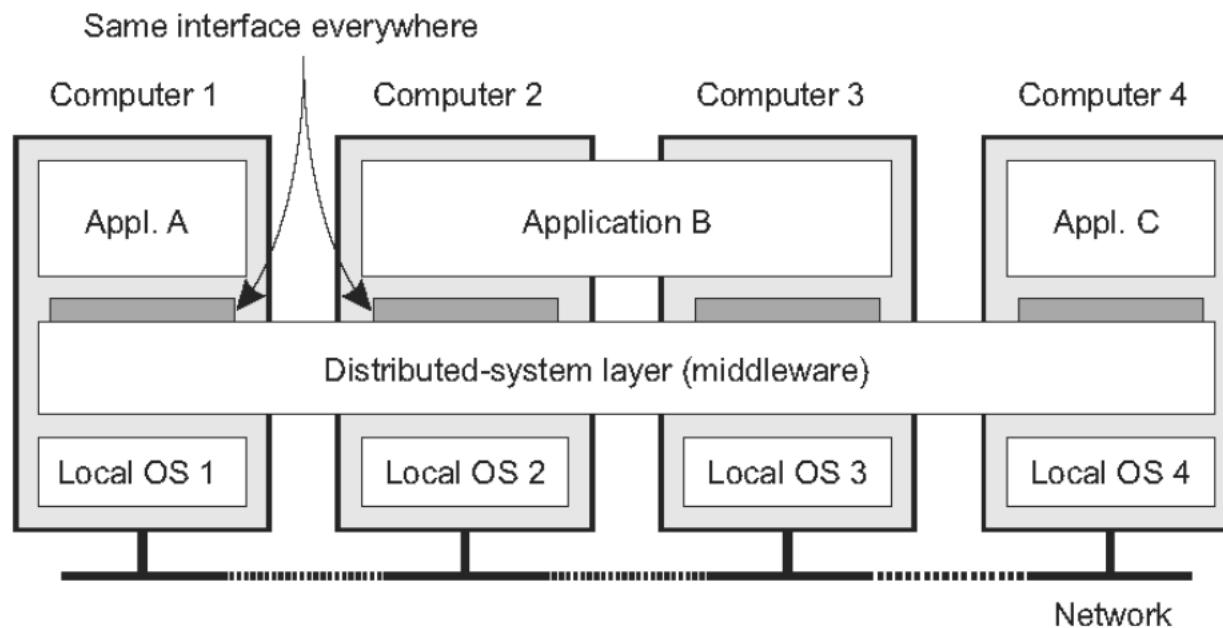
Outline

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



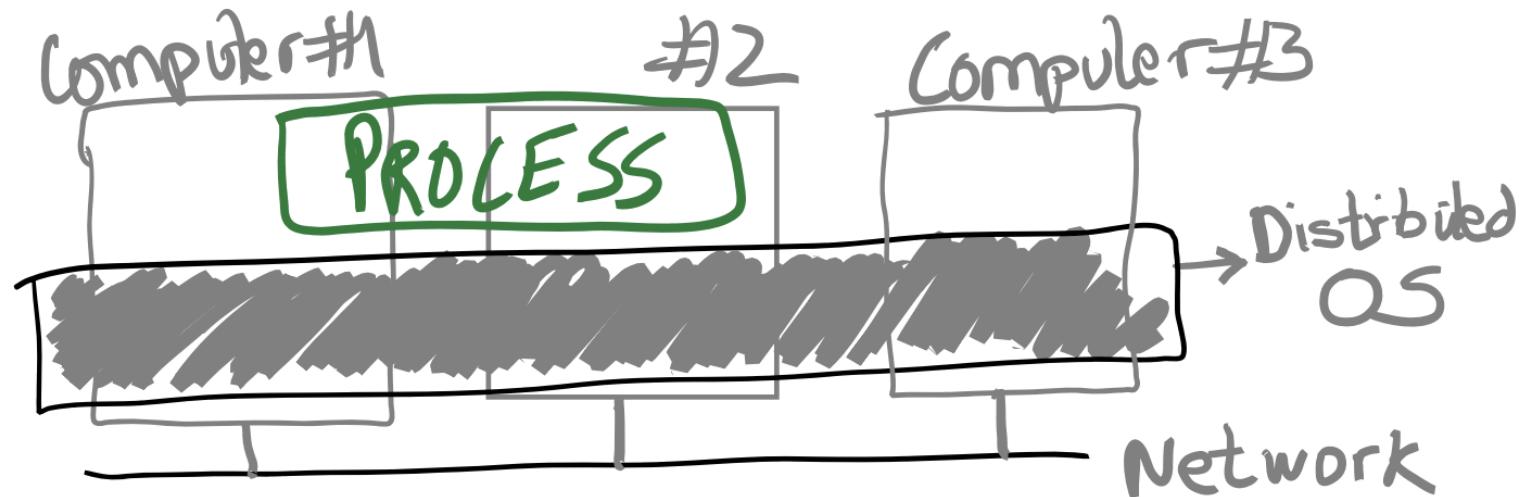
The OS of Distributed Systems

- Commonly used components and functions for distributed applications



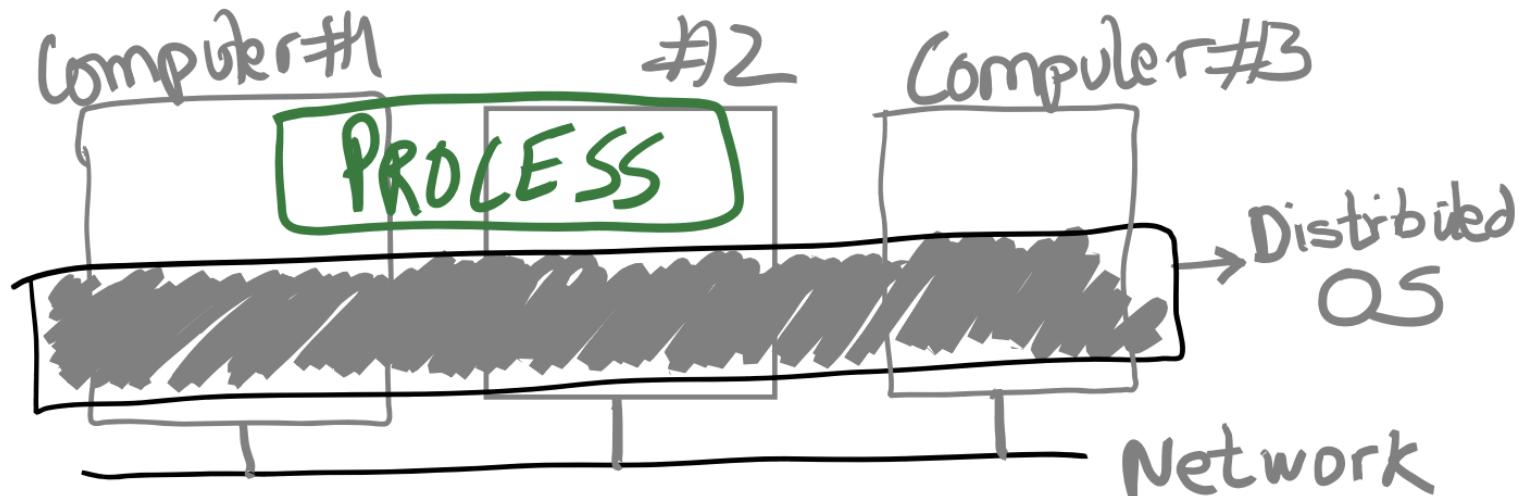
The OS of Distributed Systems

- 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

- In fact, providing abstract, virtual resources is one of the main OS services
- Providing the process **abstraction** and resource **virtualization**
- 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.



Transparency Issues In Distributed OS

PROCESS

Process state:

- Code segment
- Memory pages
- Files
- Sockets
- Security permissions

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

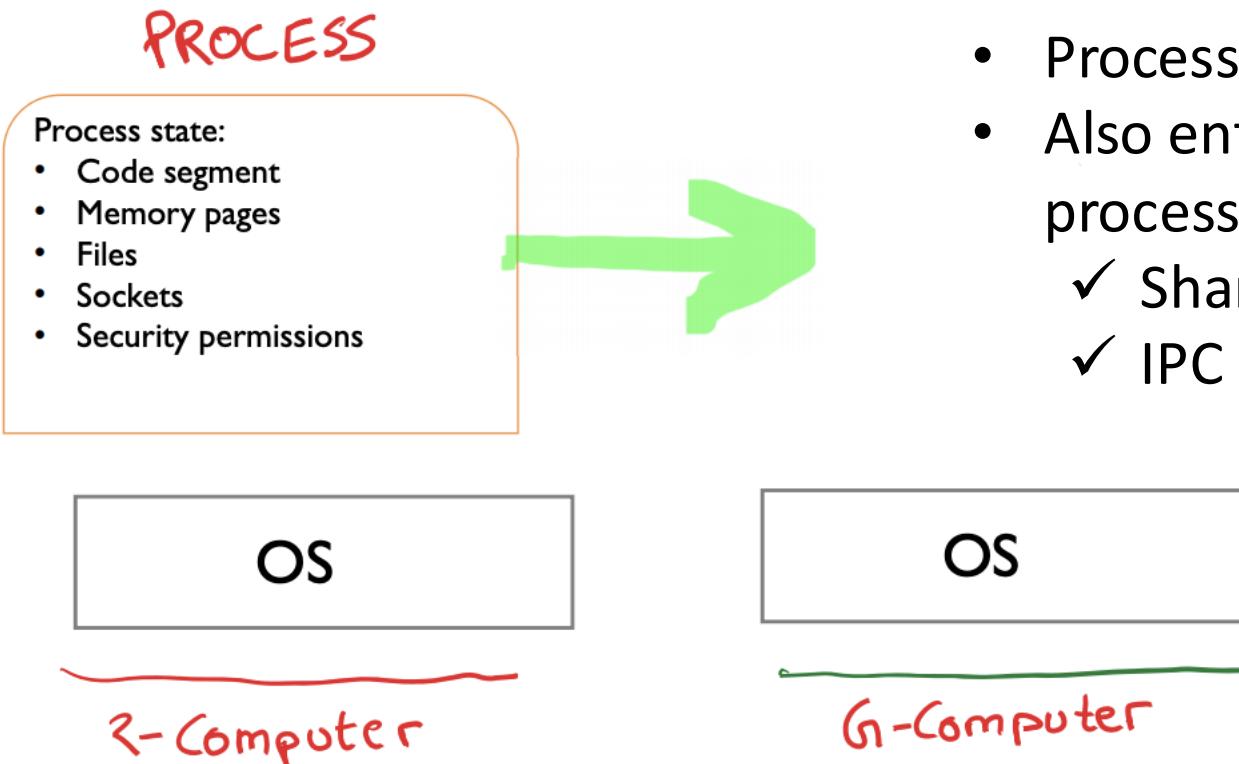
Distributed OS

2-Computer

6-Computer



Process Migration

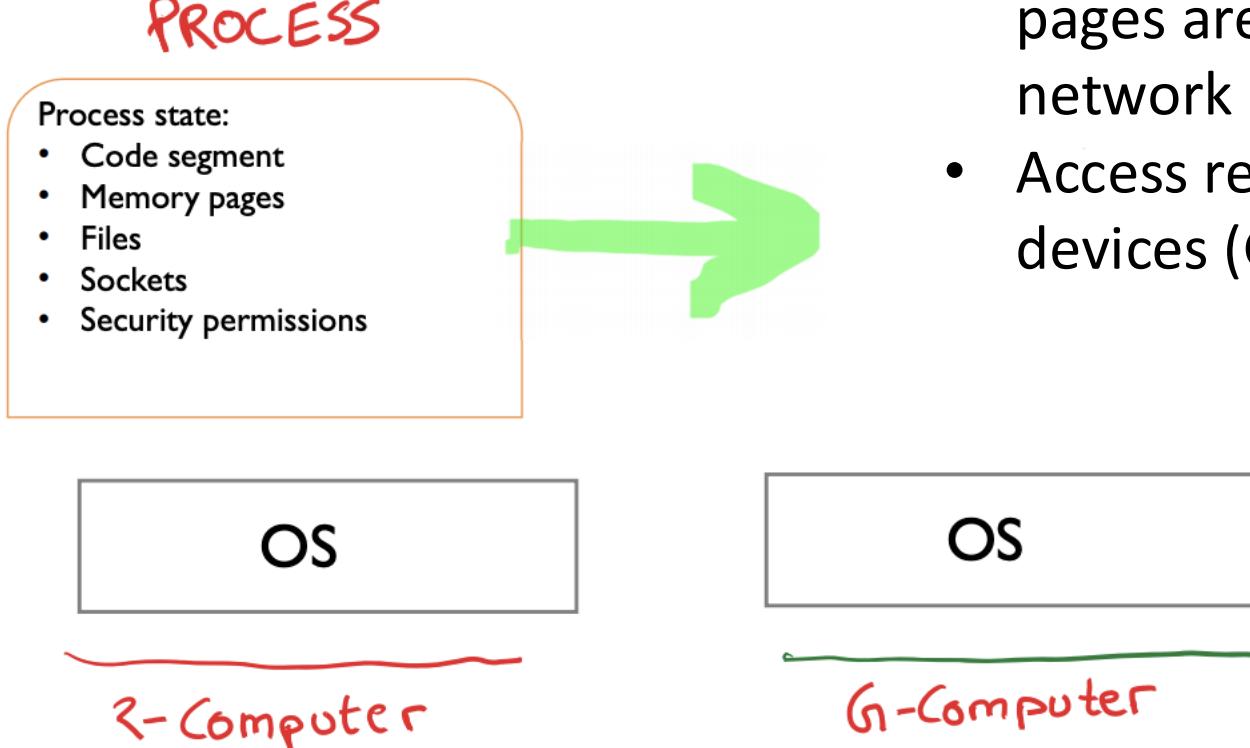


- Move all process state from one computer to another
- Process state can be vast
- Also entangled with other process states
 - ✓ Shared files?
 - ✓ IPC (pipes etc)



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

