

# CS 7172

# Parallel and Distributed Computing

## Process

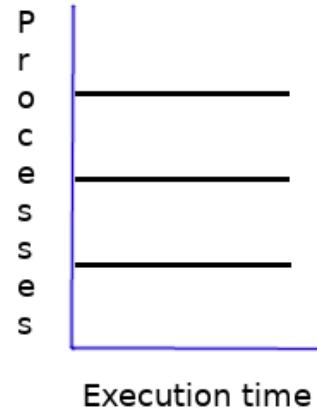
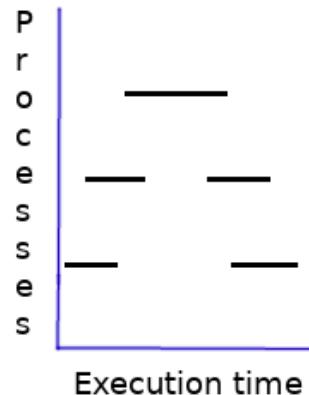
**Kun Suo**

Computer Science, Kennesaw State University

<https://kevinsuo.github.io/>

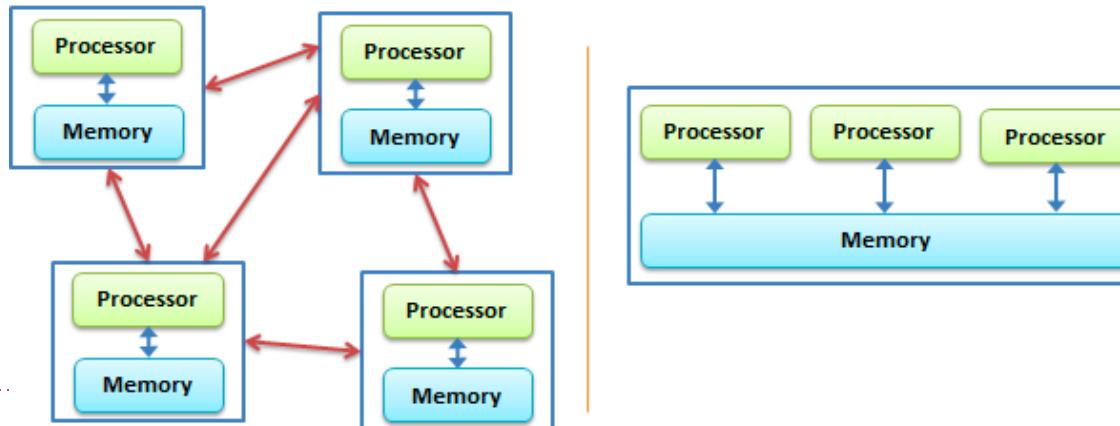
# Test

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



# Test

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



# Test

- Array A = Array B + Array C

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

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

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



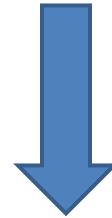
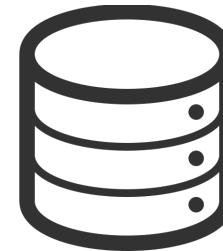
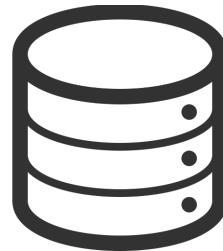
# Test

---

- You have a large data set, and you want to know the minimum, maximum, and average values of the data set
- How to design process this work using data parallelism or task parallelism?



# Test: data parallelism



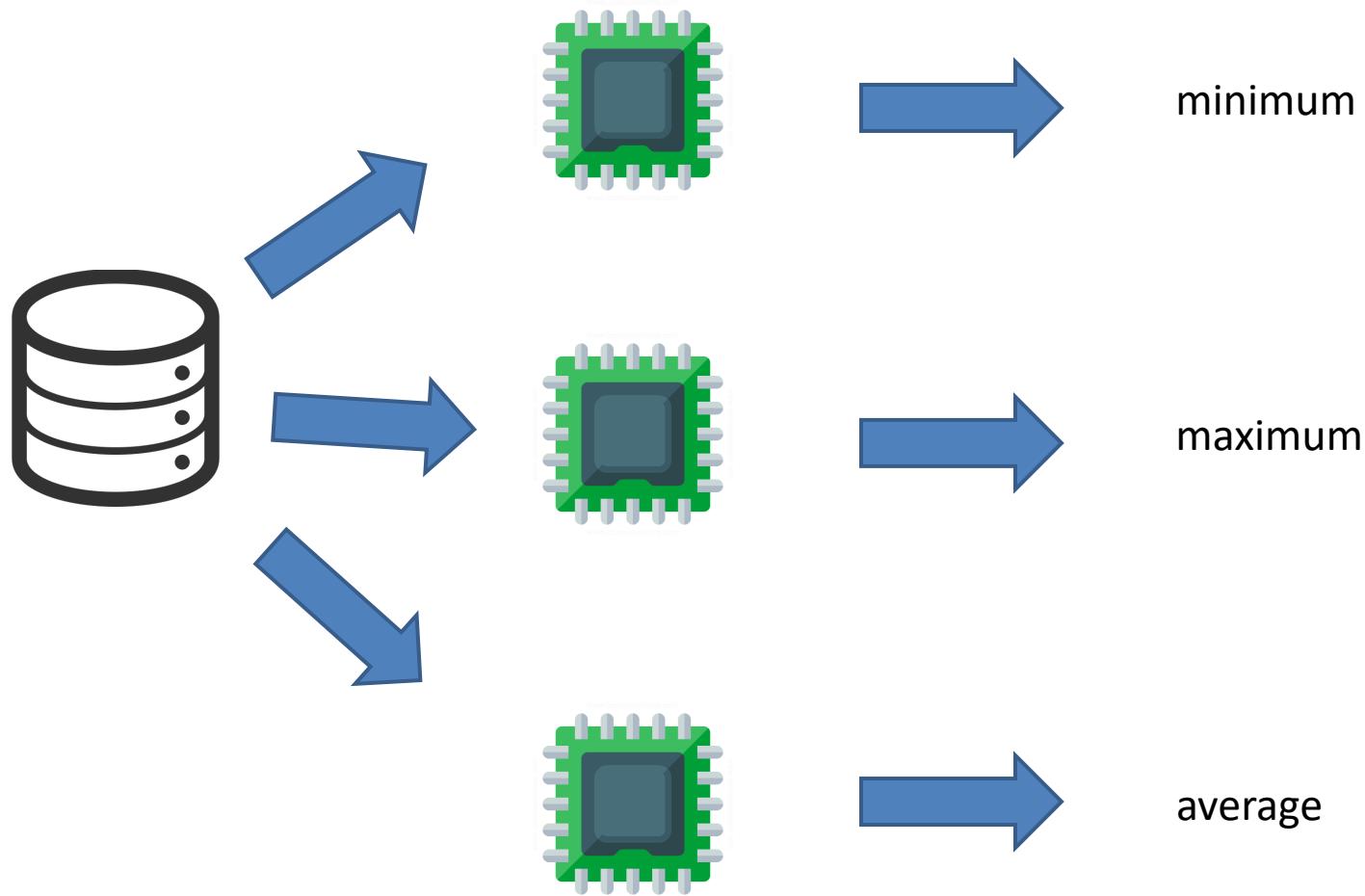
minimum,  
maximum,  
average

minimum,  
maximum,  
average

minimum,  
maximum,  
average

minimum,  
maximum,  
average

# Test: task parallelism



# Test: task parallelism

---

```
public T Min<T>(T[] items){  
    return items.Min();  
}  
  
public T Max<T>(T[] items) {  
    return items.Max();  
}  
  
Task<int> leftTask = Task.Run(() => mx.Max(array));  
Task<int> rightTask = Task.Run(() => mx.Min(array));  
Task.WaitAll(leftTask, rightTask);  
Console.WriteLine(leftTask.Result);  
Console.WriteLine(rightTask.Result);
```

```
public void (){  
    Console.WriteLine("AsyncOperation");  
}  
ThreadPool.QueueUserWorkItem(AsyncOperation);
```



# Test

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



- Theoretically speaking, the sorting problem is already difficult to optimize from the algorithm level.
- The optimal complexity is  $O(n \log n)$ . Mergesort = Quicksort = Heapsort = ...
- How to run this problem in parallel?

# Test: merge sort

---

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



# Test: quick sort

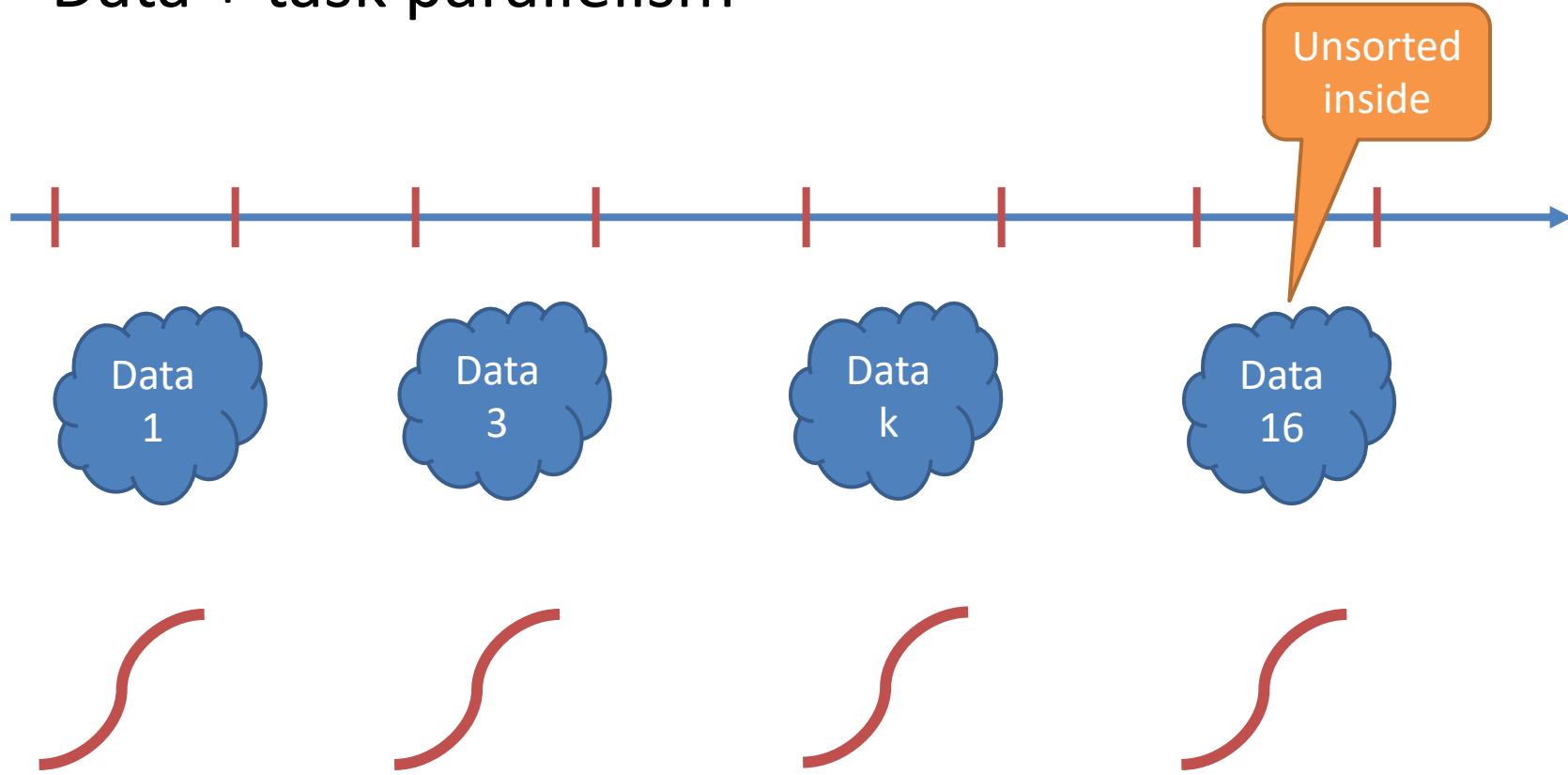
---

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



# Test: quick sort

- Data + task parallelism



# Outline

---

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



# Process

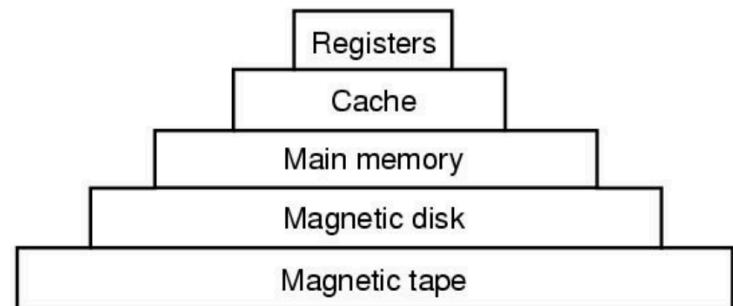
---

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



# Process

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



# Program vs. Process

---

- Program  $\neq$  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 diagram illustrates the relationship between a program and its processes. On the left, a screenshot of the Mac OS X Applications folder shows the application "Google Chrome" selected. A blue callout bubble labeled "Program: An executable file in long-term storage" points to it. On the right, a screenshot of the Activity Monitor shows multiple processes running under the name "Google Chrome Helper". A red dashed box highlights these processes. A blue callout bubble labeled "Process: The running instantiation of a program, stored in RAM" points to them. Below the processes, a summary table provides system statistics.

Program:  
An executable file in  
long-term storage

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

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



# 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.



# Process Descriptor

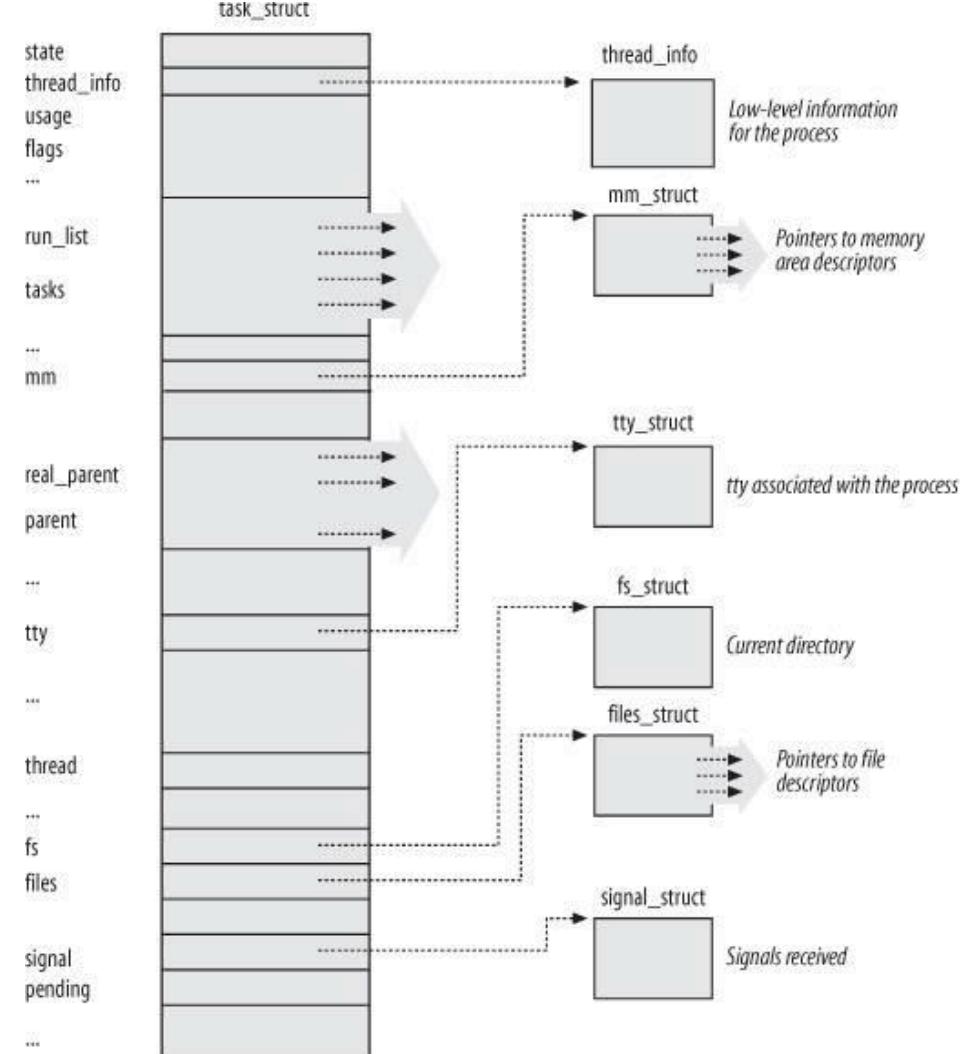


- Driving license
  - ID
  - Name
  - Address
  - Birth
  - Time
  - ...
- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info
  - File system
  - Virtual memory
  - Process specific context
  - ...

# 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

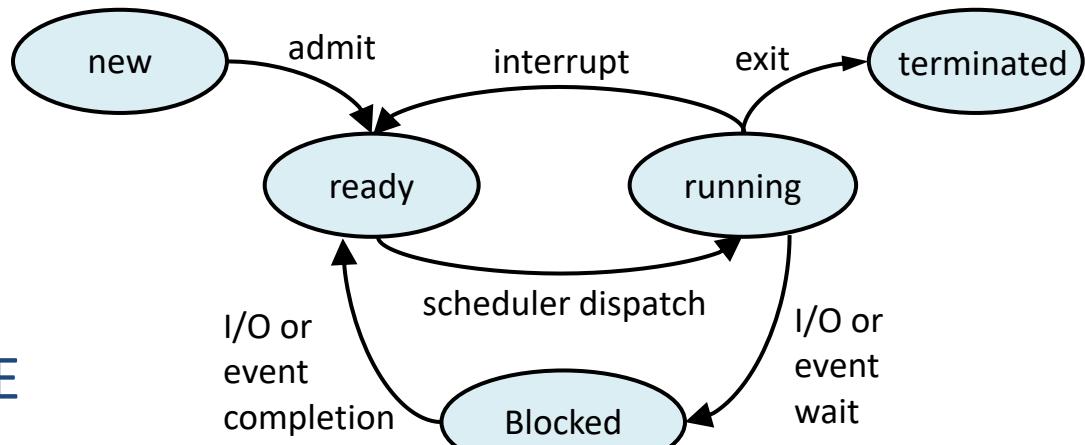
- ...



# 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

# 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')

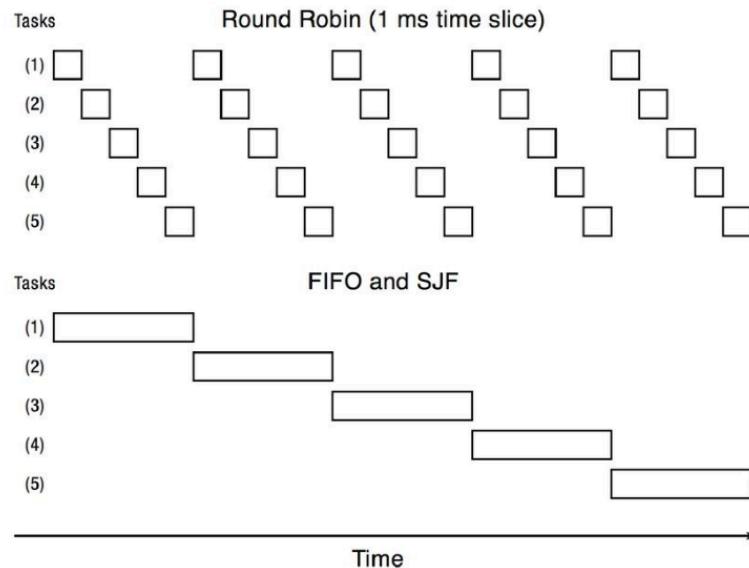
- Scheduling information
  - rt\_priority



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

```
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
- ...



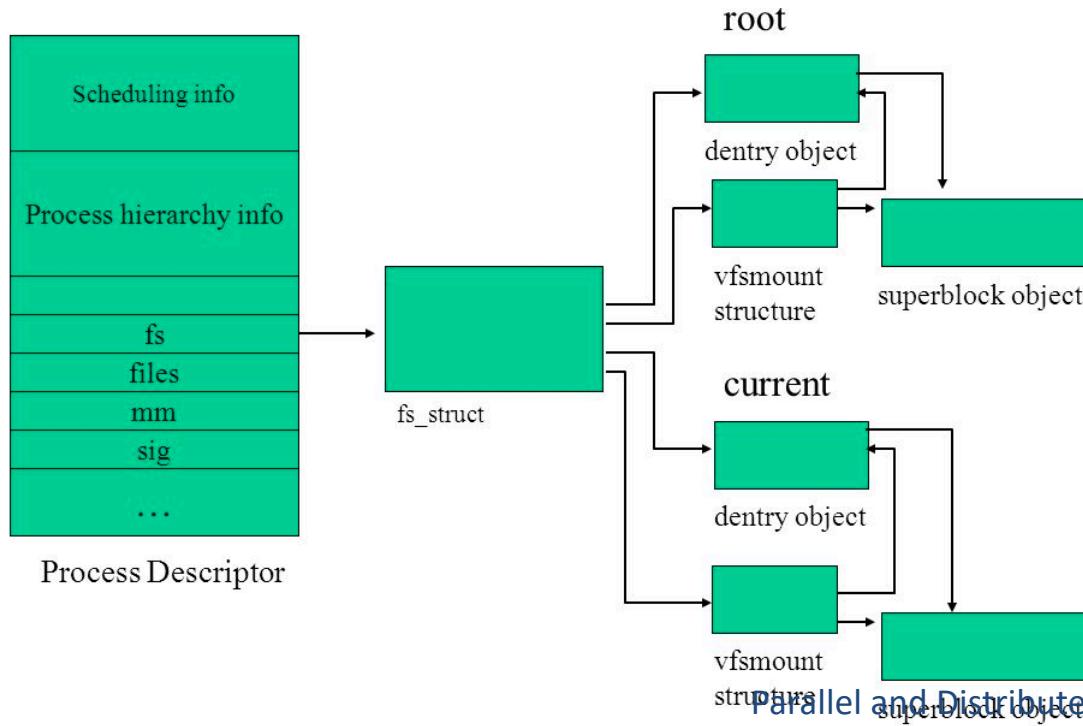
# Linux Process Control Block (cont')

- Files

- **fs\_struct**

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

- ▶ file system information: root directory, current directory



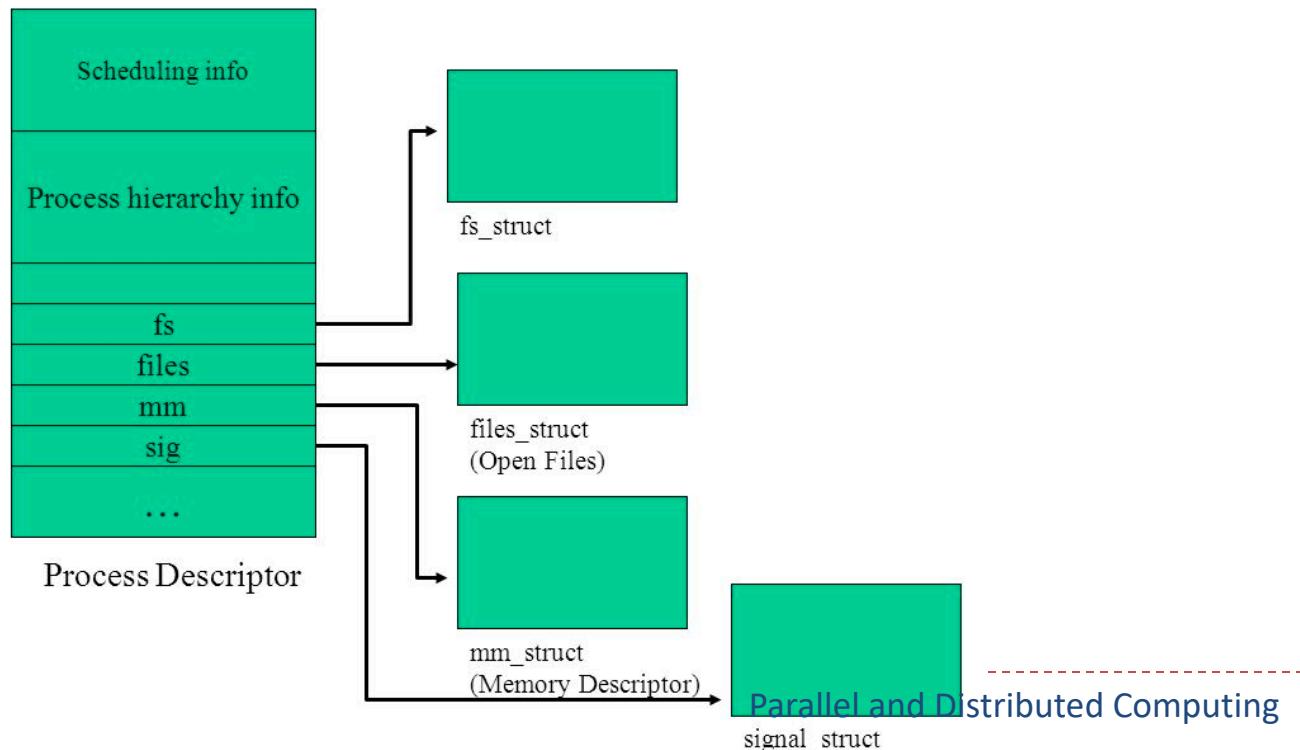
# Linux Process Control Block (cont')

- 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

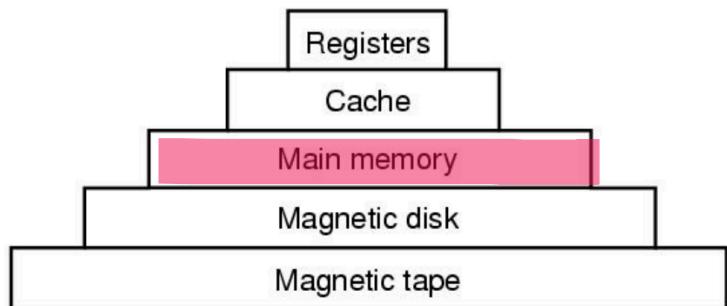
---

- 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

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

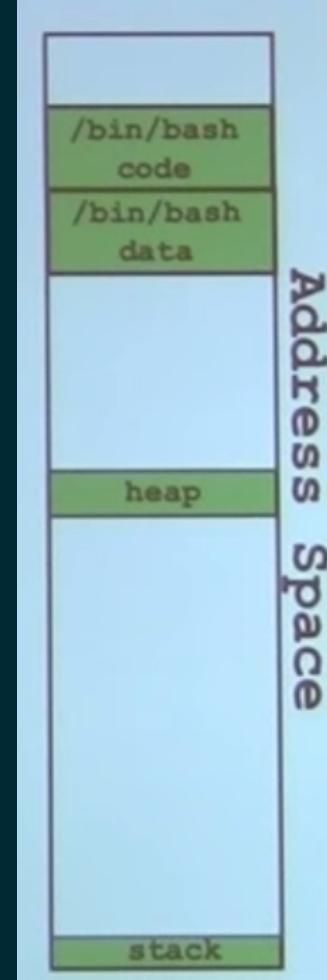
Memory address

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
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
```



The diagram illustrates the memory layout of the Bash process. It shows a vertical stack of memory regions. From top to bottom, the regions are: /bin/bash code (green), /bin/bash data (green), heap (green), and stack (green). To the right of the stack, the text "Address Space" is written vertically.

# Outline

---

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



# 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)
                └─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/v5.4/source/init/main.c#L580>

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



# Fork() system call

---

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



# Fork() system call

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



```
1. vim /home/pi/Downloads (ssh)
vim /home/pi/Dow... %1

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

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

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

    return 0;
}
```



# Fork() system call

---

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



# Fork() system call

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

```
#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
```

ting

# Fork() system call

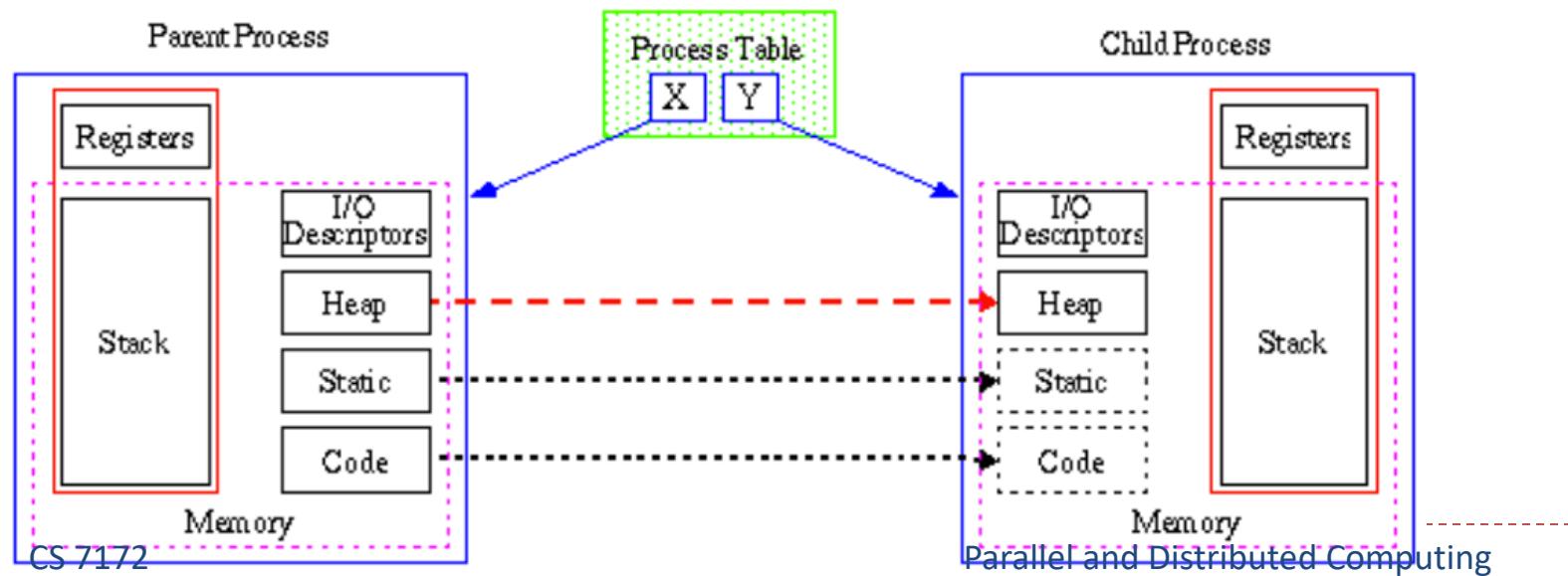
---

- 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.



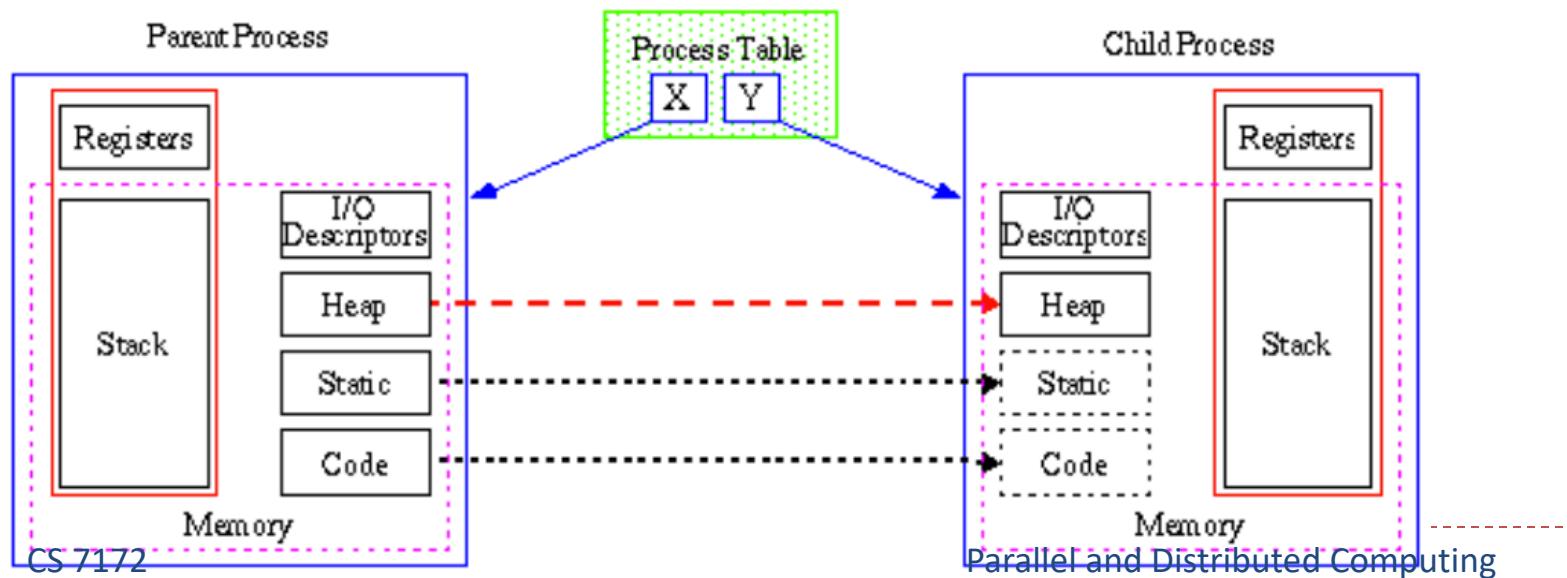
# Parent process and child process

- 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 "vim /home/pi/Downloads 1". 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 original "fork();" statement to two separate "fork();" statements in the code. A blue arrow points to the first child's "fork();" and a red arrow points to the second child's "fork();". Below the code, a blue speech bubble labeled "Parent" and a red speech bubble labeled "Child" are positioned.

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

A screenshot of a terminal window titled "vim /home/pi/Downloads 1". 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;
}
```

The code contains three nested "fork();" statements. A blue arrow points to the first child's "fork();", a red arrow points to the second child's "fork();", and another blue arrow points to the third child's "fork();". To the right of the code, a diagram shows a tree structure where the root node branches into three children, each of which further branches into two grandchildren, illustrating the recursive nature of the fork operation.

```
pi@raspberrypi ~/Downloads> ./a.o
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

۷۸۱

fish /Users/ksuo/OneDrive - Kennesaw State University

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;
}
```

"te

۲

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
```

۷۸۱

fish /Users/ksuo/OneDrive - Kennesaw State University ● #1

vim /home/ubuntu (ssh)

⌘2

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

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

"test.c" 13L, 218C

13.3

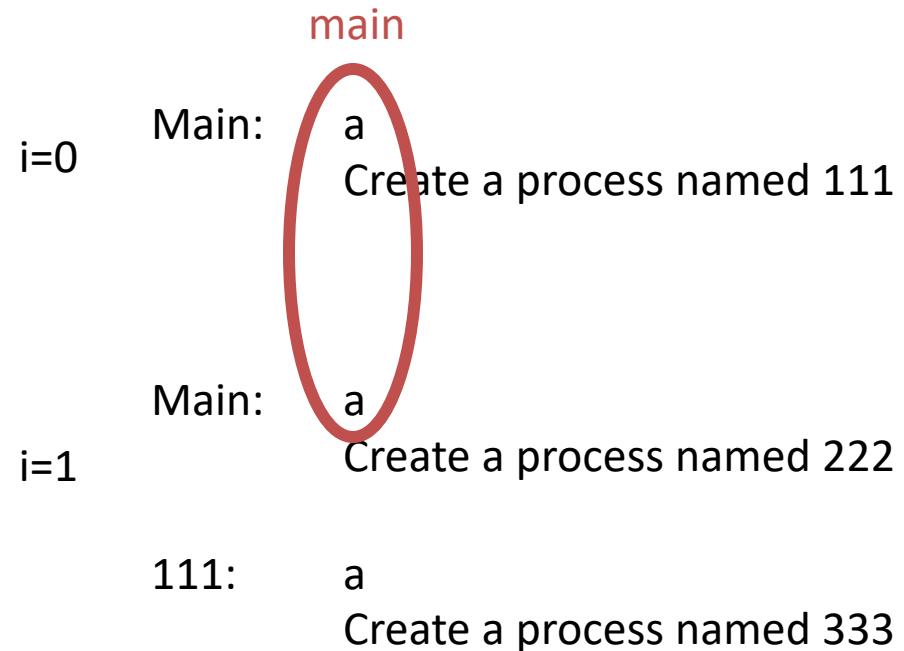
All

# Fork() example

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

```
#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;
}
```

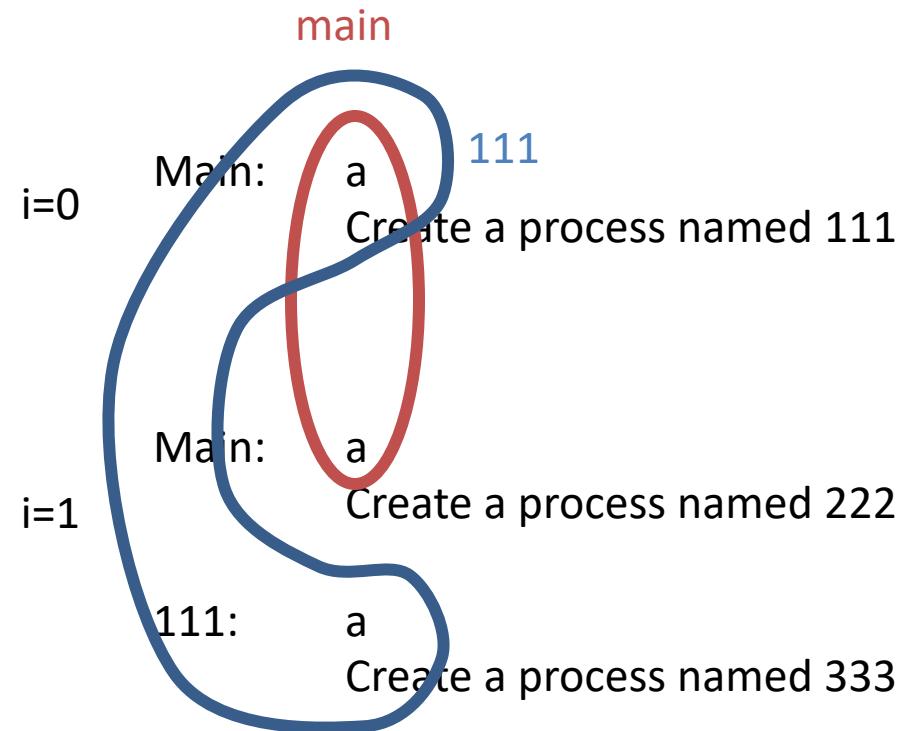


# Fork() example

- 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;
}
```

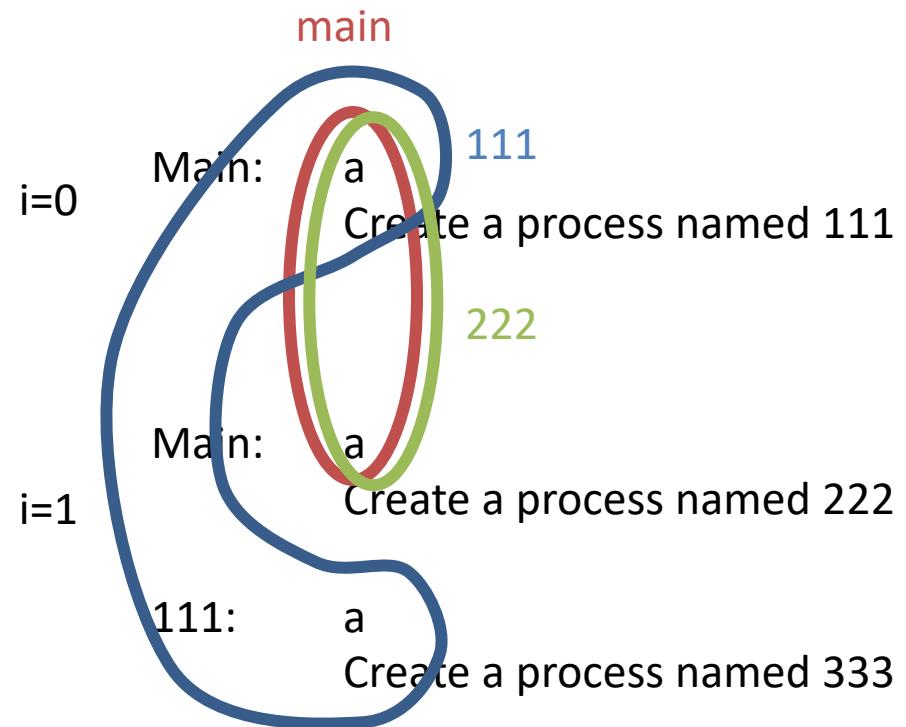


# Fork() example

- 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;
}
```

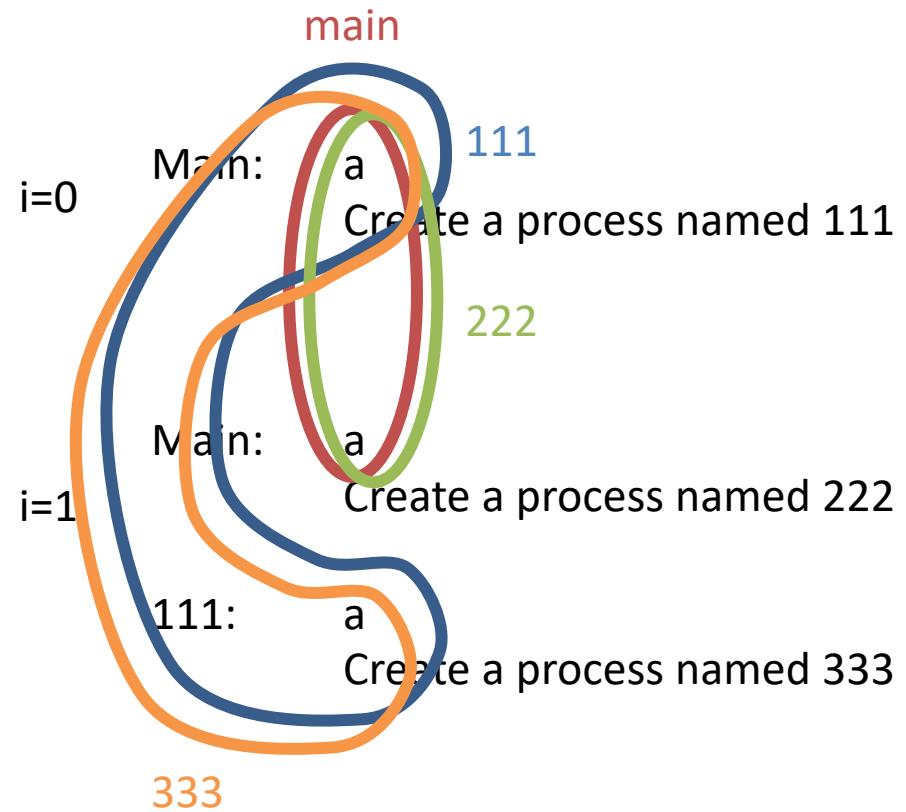


# Fork() example

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

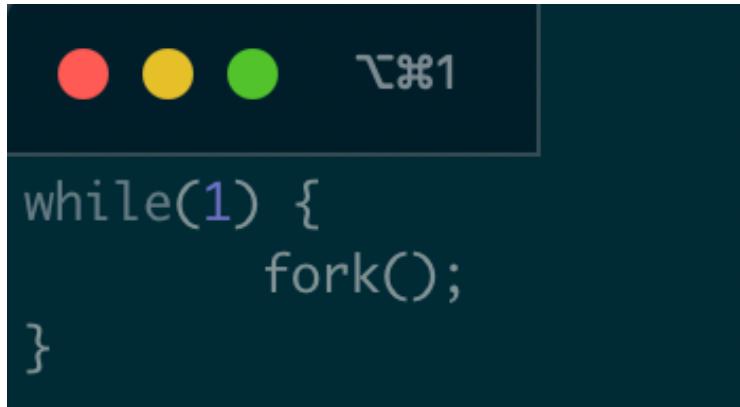
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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?



A screenshot of a terminal window with a dark background. At the top left are three colored circles (red, yellow, green) and a small icon. To their right is the text '✉⌘1'. Below this, the code for a fork() bomb is displayed:

```
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
```



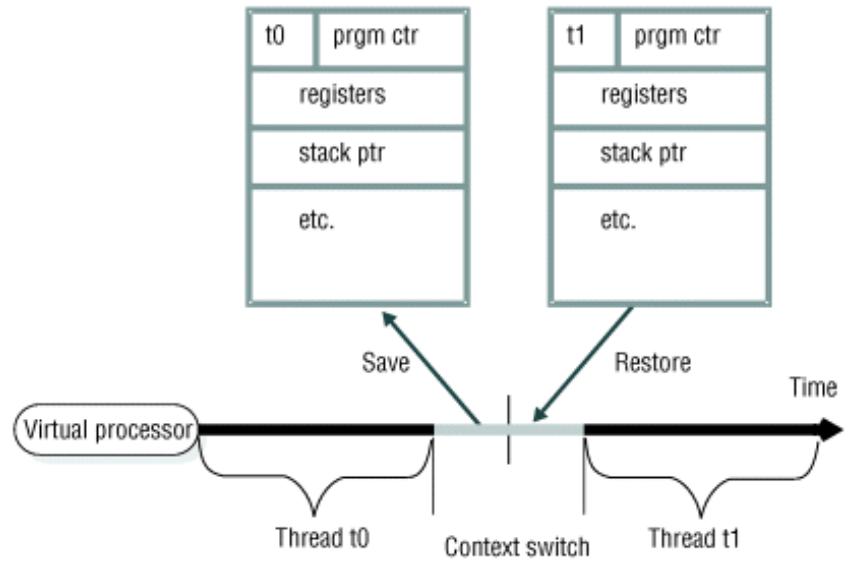
# Few other useful syscalls

---

- `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

# Context Switch

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



# Context Switch

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <time.h>
5 #include <sched.h>
6 #include <sys/types.h>
7 #include <unistd.h>      //pipe()
8
9 int main()
10 {
11     int x, i, fd[2], p[2];
12     char send    = 's';
13     char receive;
14     pipe(fd);
15     pipe(p);
16     struct timeval tv;
17     struct sched_param param;
18     param.sched_priority = 0;
19
20     while ((x = fork()) == -1);
21     if (x==0) {
22         sched_setscheduler(getpid(), SCHED_FIFO, &param);
23         gettimeofday(&tv, NULL);
24         printf("Before Context Switch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
25         for (i = 0; i < 10000; i++) {
26             read(fd[0], &receive, 1);
27             write(p[1], &send, 1);
28         }
29         exit(0);
30     }
31     else {
32         sched_setscheduler(getpid(), SCHED_FIFO, &param);
33         for (i = 0; i < 10000; i++) {
34             write(fd[1], &send, 1);
35             read(p[0], &receive, 1);
36         }
37         gettimeofday(&tv, NULL);
38         printf("After Context SWitch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
39     }
40     return 0;
41 }
```

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

When A writes pipe, B will be blocked

fd[1]  
Write()

Pipe

fd[0]  
Read()

When B reads pipe, A will be blocked

# Context Switch

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <time.h>
5 #include <sched.h>
6 #include <sys/types.h>
7 #include <unistd.h>      //pipe()
8
9 int main()
10 {
11     int x, i, fd[2], p[2];
12     char send    = 's';
13     char receive;
14     pipe(fd);
15     pipe(p);
16     struct timeval tv;
17     struct sched_param param;
18     param.sched_priority = 0;
19
20     while ((x = fork()) == -1);
21     if (x==0) {
22         sched_setscheduler(getpid(), SCHED_FIFO, &param);
23         gettimeofday(&tv, NULL);
24         printf("Before Context Switch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
25         for (i = 0; i < 10000; i++) {
26             read(fd[0], &receive, 1);
27             write(p[1], &send, 1);
28         }
29         exit(0);
30     }
31     else {
32         sched_setscheduler(getpid(), SCHED_FIFO, &param);
33         for (i = 0; i < 10000; i++) {
34             write(fd[1], &send, 1);
35             read(p[0], &receive, 1);
36         }
37         gettimeofday(&tv, NULL);
38         printf("After Context SWitch Time%u s, %u us\n", tv.tv_sec, tv.tv_usec);
39     }
40
41     return 0;
42 }
```

Run 10k times of  
read and write

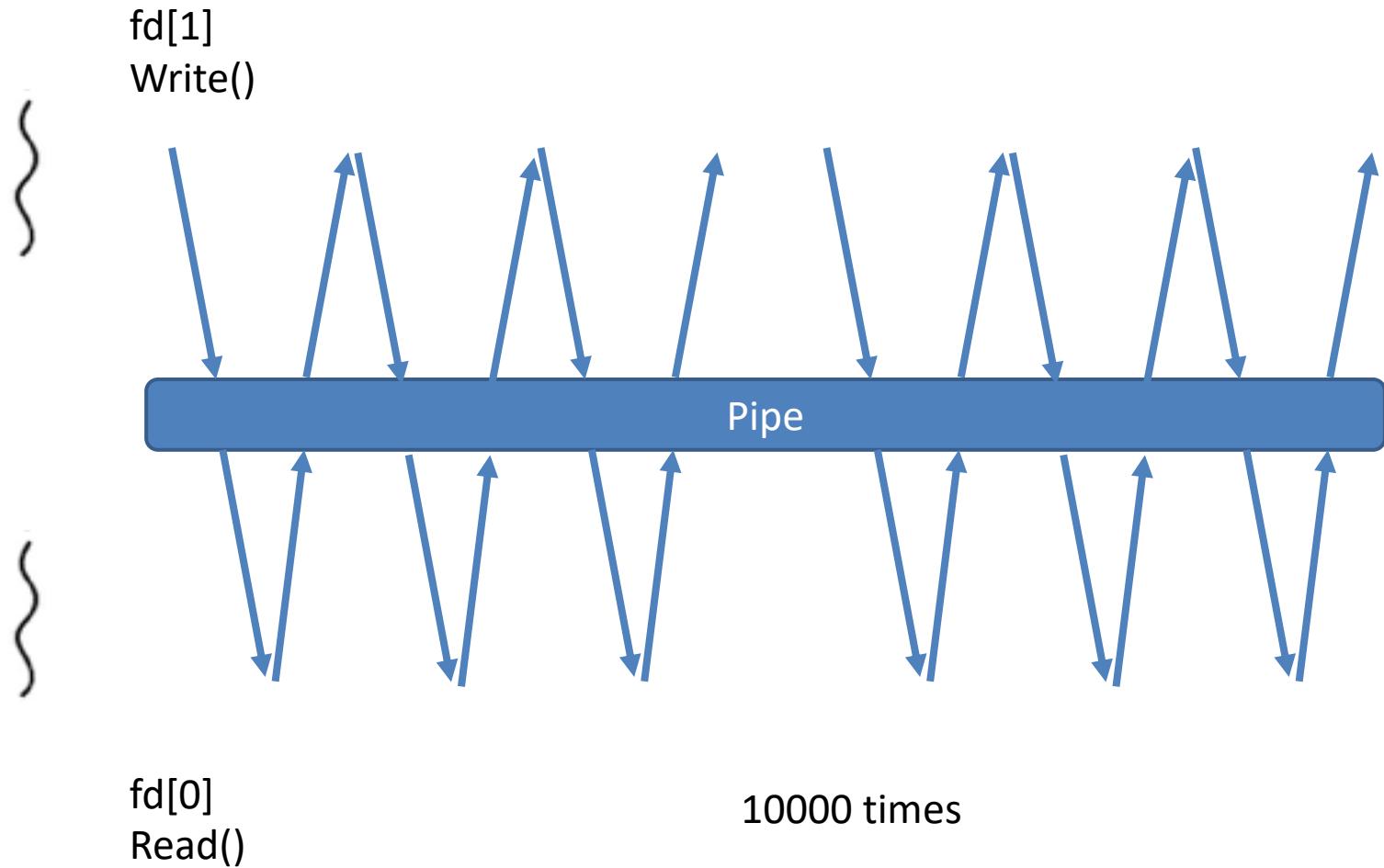
Run 10k times of  
read and write

fd[1]  
Write()

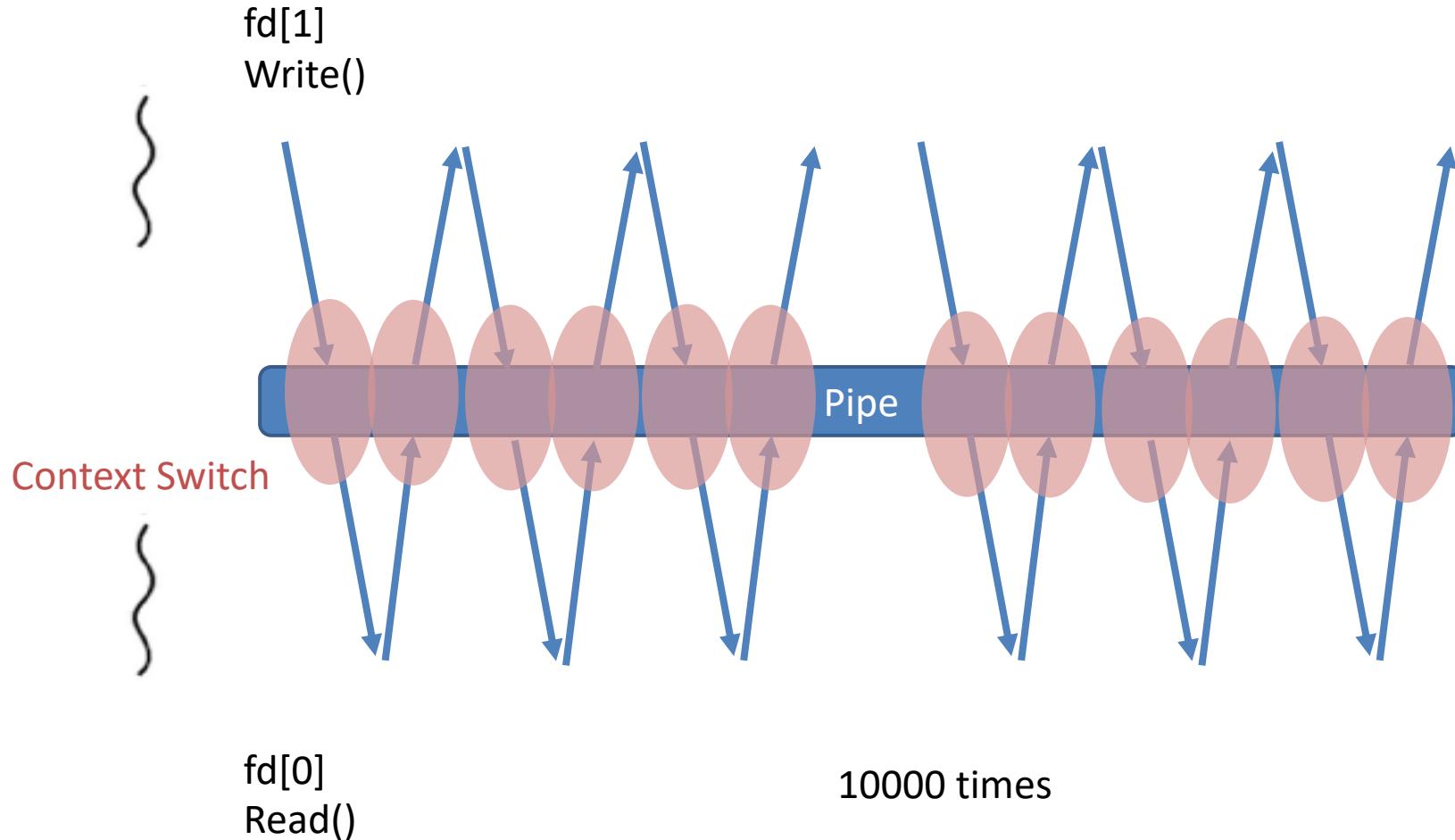
Pipe

fd[0]  
Read()

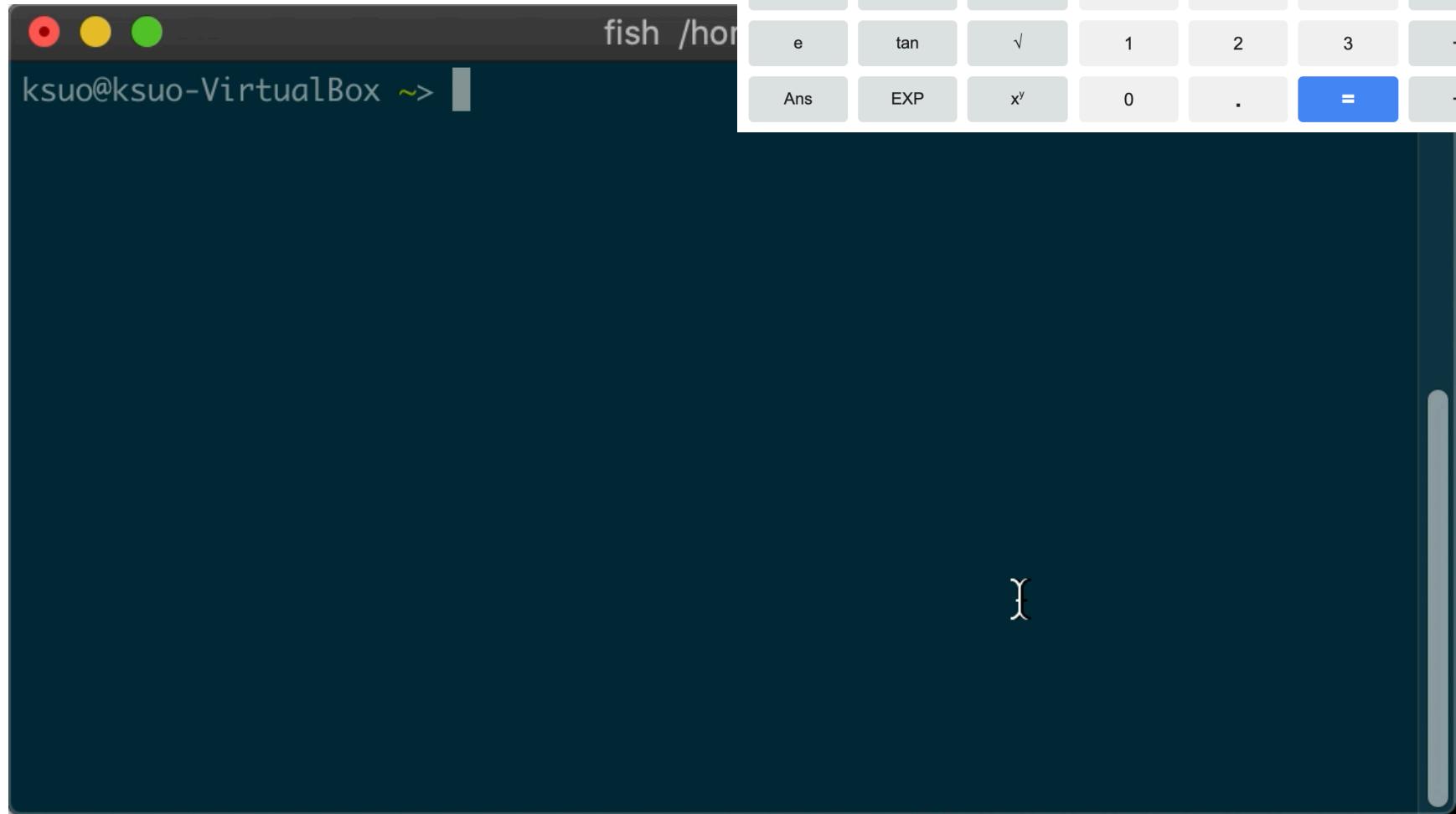
# Context Switch



# Context Switch

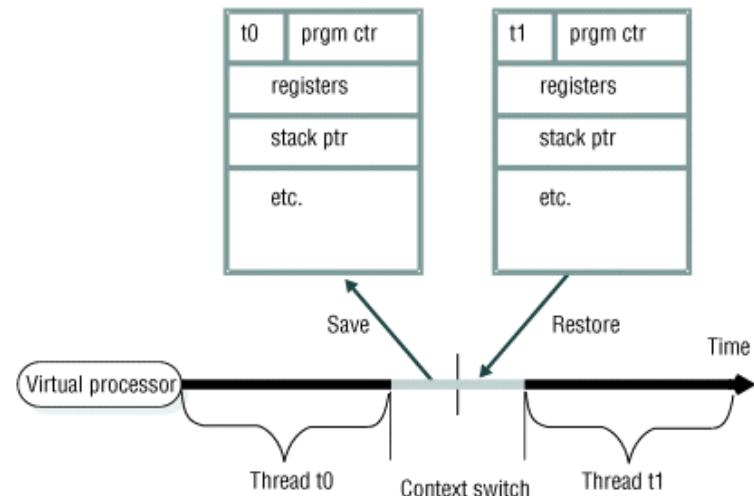
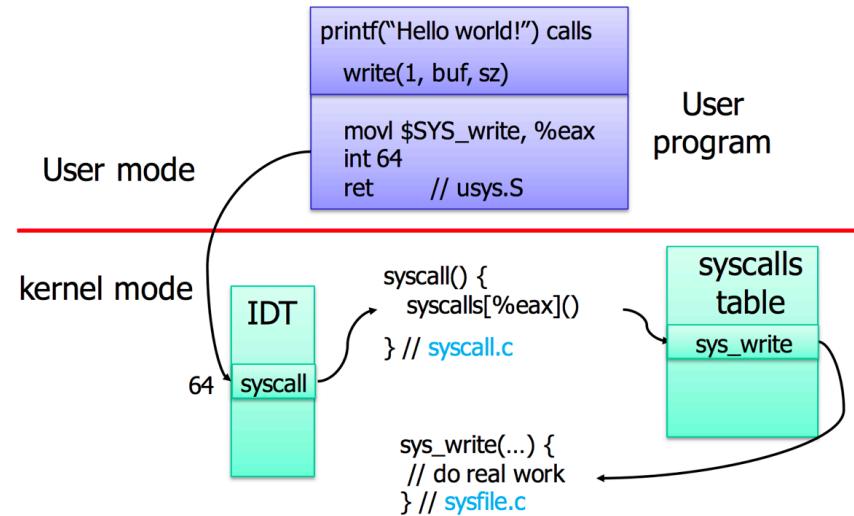


# Context Switch



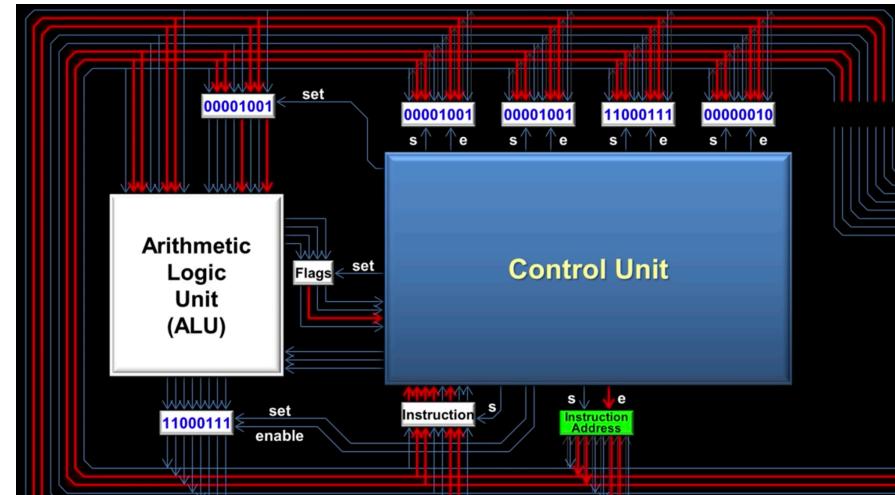
# Context Switch

- The general system call time consumption is around 200ns
- Context switch is 5us >> 200ns



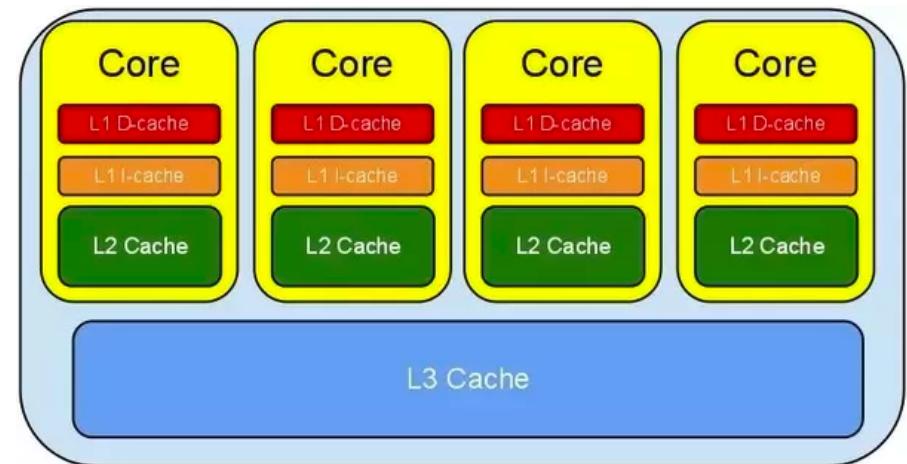
# The Cost of Context Switch

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



# The Cost of Context Switch

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



# Outline

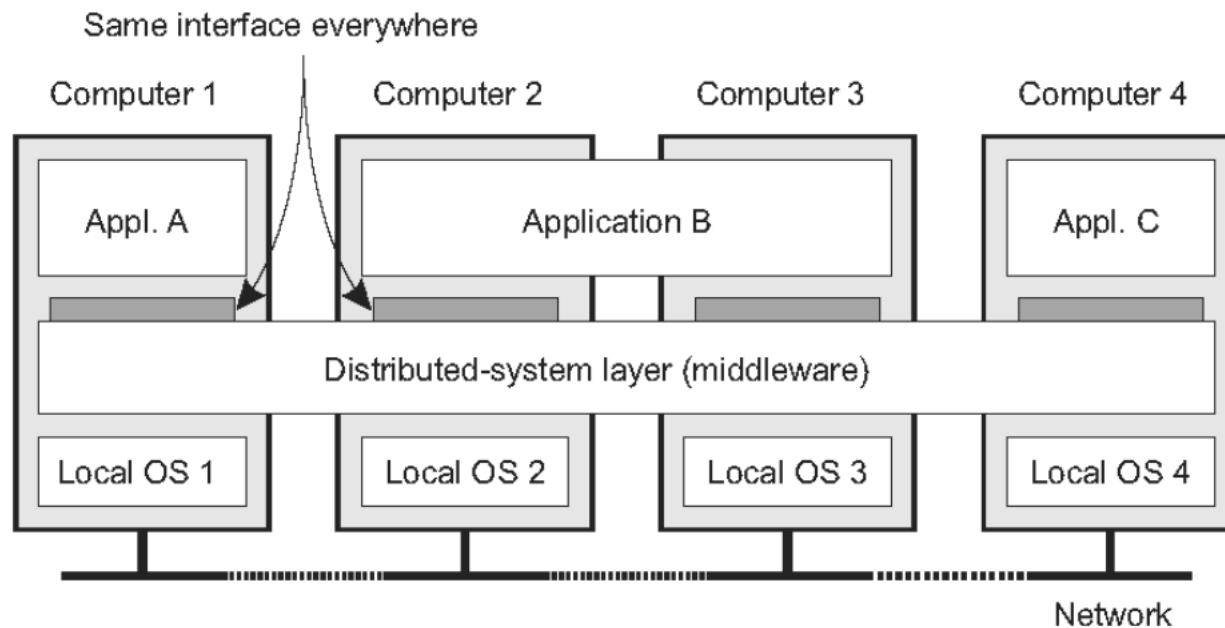
---

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



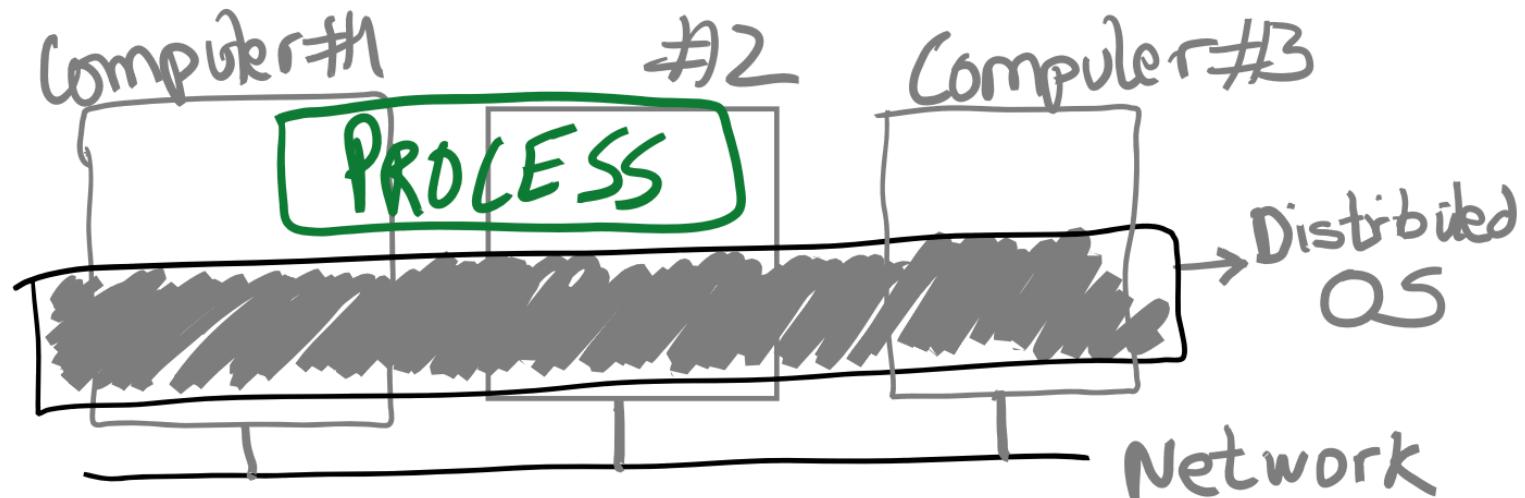
# The OS of Distributed Systems

- 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



# Distributed OS Challenges

---

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



# Processes In Distributed OS

## PROCESS

Process state:

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

Distributed OS

2-Computer

6-Computer



# 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?

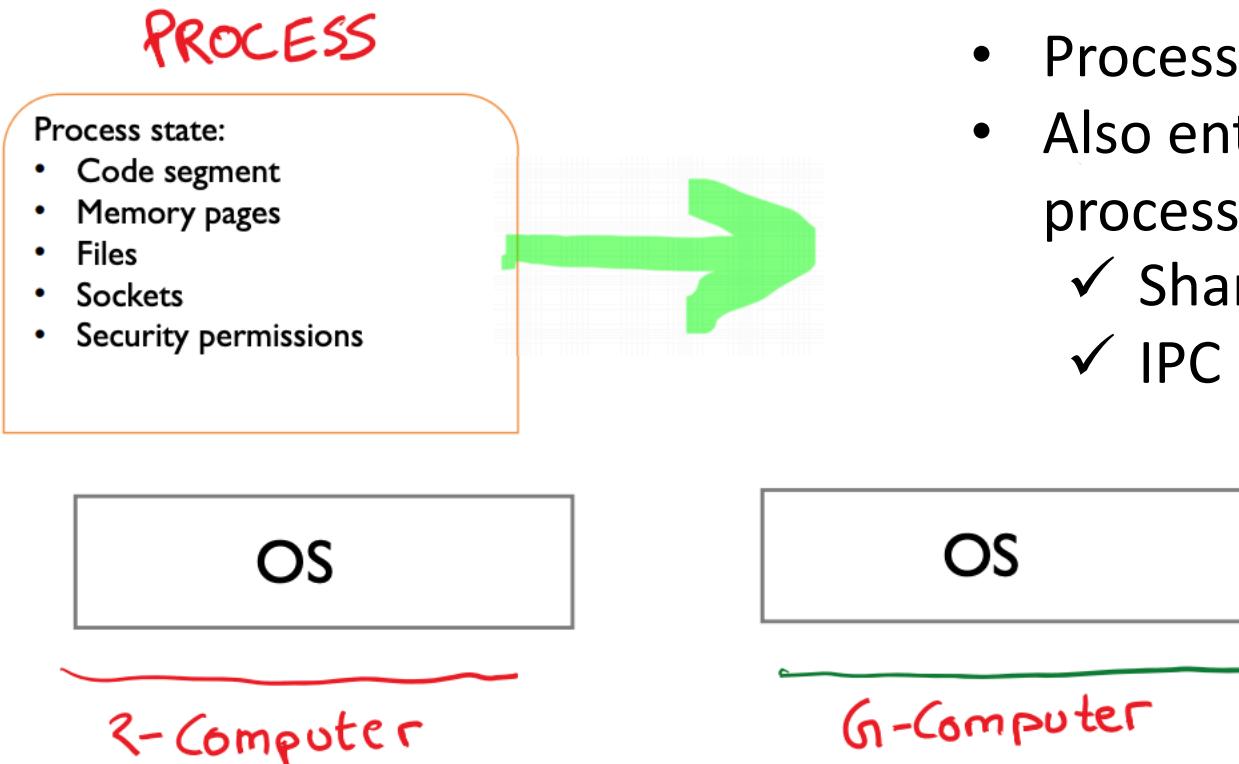
## Distributed OS

R-Computer

G-Computer



# Process Migration



# 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

