ALLEGHENY COLLEGE

Operating Systems:

Processes

Chapter 2

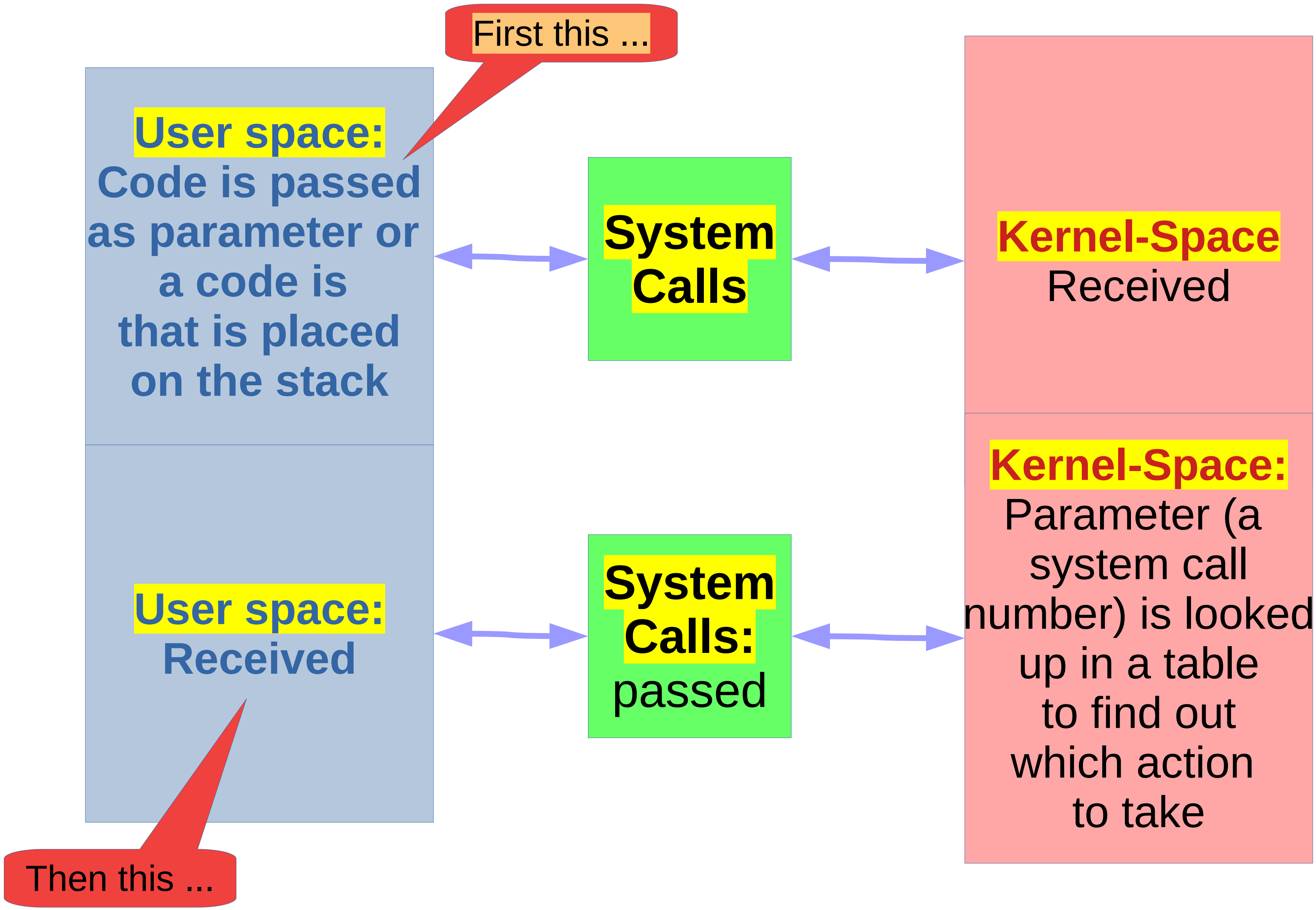
CS400

Week 2: 23rd Jan

Spring 2020

Oliver BONHAM-CARTER

Remember *System Calls (SCs)*?



A Notable System Call

`fork()`

Fork()

- Fork system call is used for creating a new process, called a *child process*,
- Process runs concurrently with another process (the parent process) that was initiated by fork() call.
- Both processes execute the next instruction following the fork() system call (in code).
- A child process uses the same program counter, the same CPU registers, and the same open files as its parent process.

`Fork()`: Return Values at Run-time

- How do we know whether `fork()` has run successfully?
  - If `fork()` returns a negative value, the creation of a child process was unsuccessful.
  - Process `fork()` returns a zero to the newly created child process.
  - Process `fork()` returns a positive value, the process ID of the child process, to the parent.
  - Because the process ID is a mere integer the `getpid()` can be used to determine the process ID assigned to this process.

Execute a Program with `Fork()`

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

   // The next line is run as a
   // child of the parent process
   fork();

   printf("Hello world!\n");
   return 0;
}
```

- The program runs normally up to the `fork()` and then executes two iterations of the process.
- This means making two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the `fork()` call.

Execute a Program with `Fork()`

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

    // The next line is run as a
    // child of the parent process
    fork();

    printf("Hello world!\n");
    return 0;
}
```

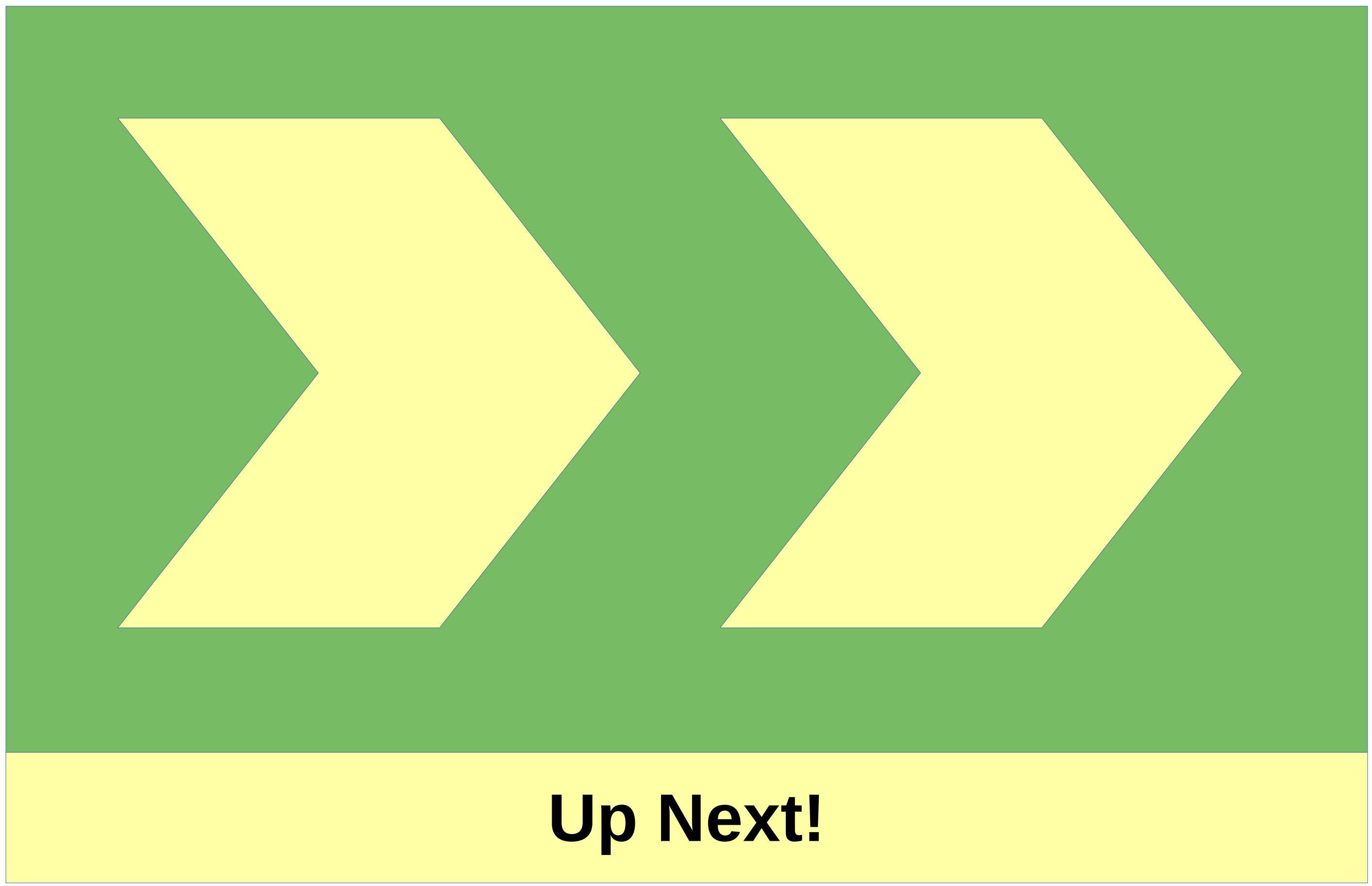
Parent Process

```c
...
    printf("Hello world!\n");
    return 0;
}
```

Child Process

```c
...
    printf("Hello world!\n");
    return 0;
}
```

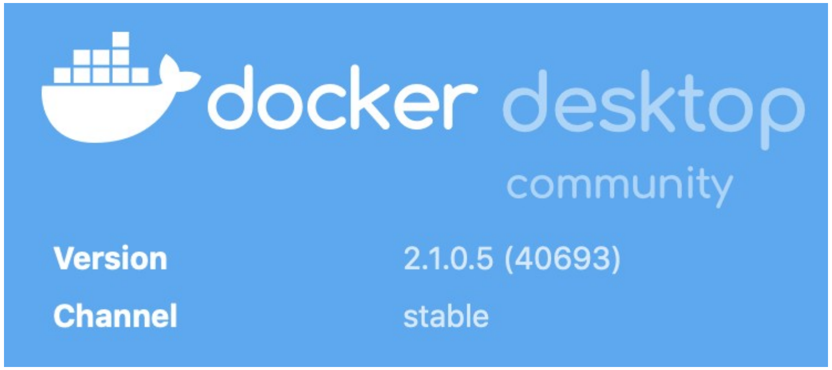
Let’s Code and Run Some Calls!



Commands to Run
From (Linux) Bash

- Build the container :
  - docker build -t gccdev .

- Run the container :
  - docker run -it gccdev



- Mount local drive and run container :
  - docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev

Note: the directory where you run this becomes your local directory in the container.

Programming with `Fork()`

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

    // The next line is run as a
    // child of the parent process
    fork();

    printf("Hello world!\n");
    return 0;
}
```

See file: `sandbox/fork1.c`

Consider the following

1. What is your output?

2. Can you explain how this output happened?

Programming with `Fork()`

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

#define MAX_COUNT 200
#define BUF_SIZE 100

void main(void)
{
    pid_t pid;
    int   i;
    char  buf[BUF_SIZE];


    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "  This line is from pid %d,
             value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

See file: `sandbox/fork2.c`

Consider the following

1. What is your output?

2. Can you explain how this output happened?

Programming with `Fork()`

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#define  MAX_COUNT  200
void  ChildProcess(void);      /* child process prototype  */
void  ParentProcess(void);    /* parent process prototype */
void  main(void)
{
    pid_t  pid;
    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
void  ChildProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}
void  ParentProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}
```

See file: `sandbox/fork3.c`

Consider the following

1. What is your output?
2. Can you explain how this output happened?

Let’s Talk About *fork3.c*

- Both processes of *fork3.c* print lines according to:
  - Whether the line is printed by the child or by the parent process, and
  - The value of variable i.
  - When the main program executes `fork()`, an identical copy of its address space, including the program and all data, is created.
  - System call `fork()` returns the child process ID to the parent and **returns 0 to the child process**.

```
main()
{
  pid = fork();
  if (pid == 0)
      childProcess();
  else
      parentProcess();
}
void childProcess(/* args */)
    {/* code */}

void parentProcess(/* args */)
   {/* code */}
```

Child /Parent Processes

```
main()
{
  pid = fork();
  if (pid == 0)
      childProcess();
  else
      parentProcess();
}
void childProcess(/* args */)
    {/* code */}

void parentProcess(/* args */)
   {/* code */}
```

pid = 0

```
main()
{
  pid = fork();
  if (pid == 0)
      childProcess();
  else
      parentProcess();
}
void childProcess(/* args */)
    {/* code */}

void parentProcess(/* args */)
   {/* code */}
```

pid = 123

After fork(),
Say, pid = 0
--> childProcess()

After fork(),
Say, pid = 123
--> parentProcess()

ALLEGHENY COLLEGE

Child /Parent Processes

- The CPU scheduler will assign a run time permission (a *time quantum*) to each process.
- Each process (parent or child) gets to run for a specified time before the other process gets a turn.
- Whichever process that is running, is able to print to the screen while the other waits.
- MAX_COUNT = 200 (see fork3.c code) implies that each process is able to run for enough time (two or more *time quanta*) before the other process gets its turn

```
...
This line is from parent, value = 199
   This line is from child, value = 182
This line is from parent, value = 200
   This line is from child, value = 183
*** Parent is done ***
```

```
...
   This line is from child, value = 190
   This line is from child, value = 191
   This line is from child, value = 192
   This line is from child, value = 193
 *** Child process is done ***
```

Output from running: sandbox/fork3.c

Programming with `Fork()`

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

int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

See file: `sandbox/fork4.c`

Consider the following

1. What is your output?

2. Can you explain how this output happened?

Programming with `Fork()`

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

int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

See file: `sandbox/fork4.c`

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3

       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2       L2   // There will be 2 child processes
 /  \     /  \  //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

- A total of eight processes created: seven new child processes and one original process).
- Total_Number_of_Processes = 2*n, where n is number of fork system calls

Programming with `Fork()`

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

int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:

```
            P0
        /   |   \
      P1    P4   P2
    /  \          \
  P3    P6         P5
 /
P7

The main process: P0
Processes from 1st fork: P1
Processes from 2nd fork: P2, P3
Processes from 3rd fork: P4, P5, P6, P7
```

What are Processes?

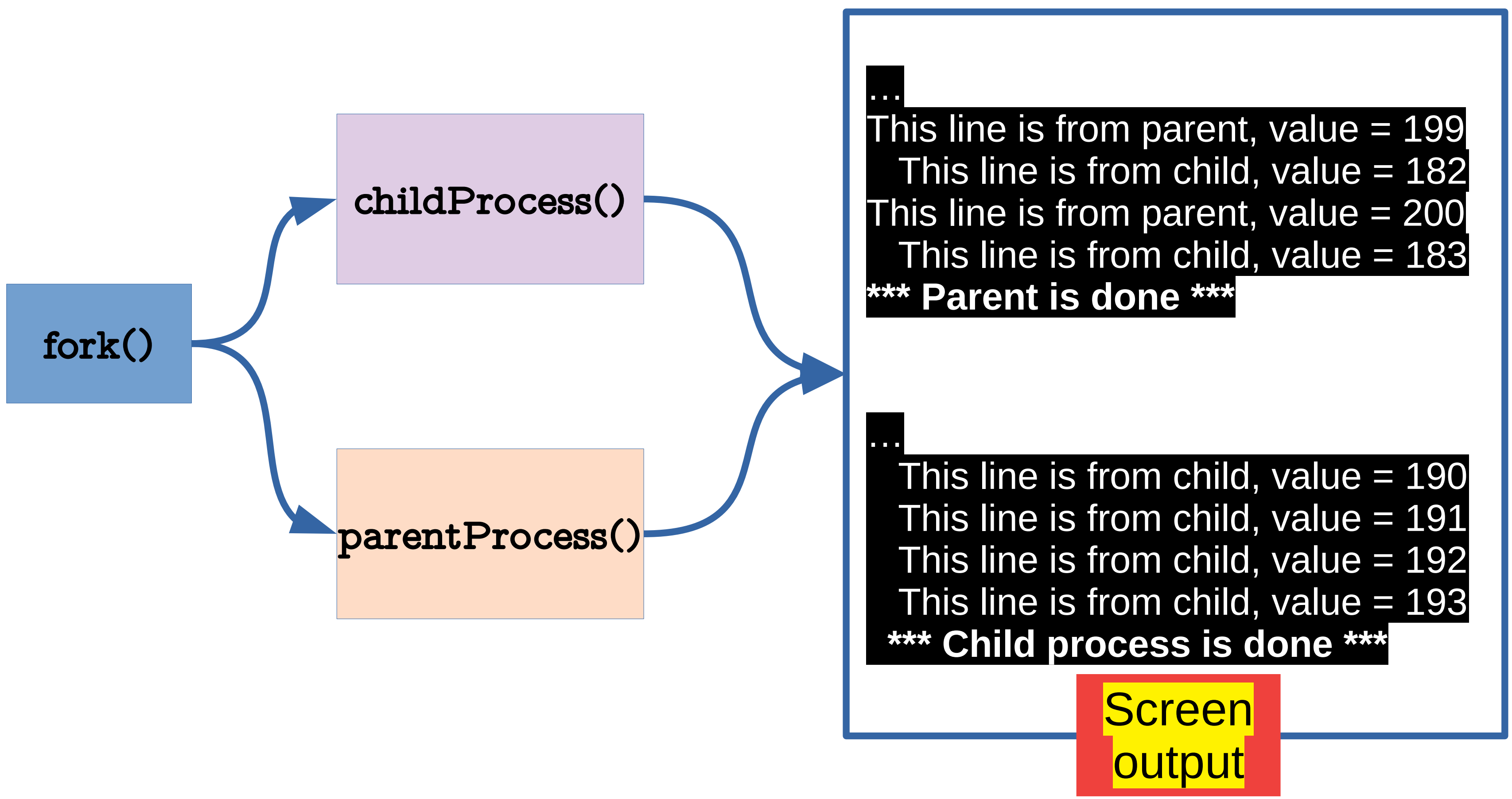
```
howtogeek@ubuntu: ~
top - 03:48:40 up 19 min,  1 user,  load average: 0.16, 0.09, 0.16
Tasks: 143 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.6%us,  0.7%sy,  0.0%ni, 96.7%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:   1025656k total,   678580k used,   347076k free,    79936k buffer
Swap:        0k total,        0k used,        0k free,   310528k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1216 root      20   0 32624 3460 2860 S  0.7  0.3   0:05.31 vmtoolsd
 2025 howtogee  20   0 81456  23m  17m S  0.7  2.3   0:01.41 unity-2d-p
   17 root      20   0     0    0    0 S  0.3  0.0   0:00.34 kworker/0:
   36 root      20   0     0    0    0 S  0.3  0.0   0:00.10 scsi_eh_1
 1081 root      20   0  199m  60m 7340 S  0.3  6.0   0:13.42 Xorg
 1973 howtogee  20   0  6568 2832  916 S  0.3  0.3   0:06.24 dbus-daemo
 2153 howtogee  20   0  147m  16m 9820 S  0.3  1.7   0:03.63 unity-pane
 2313 howtogee  20   0  136m  13m  10m S  0.3  1.4   0:00.84 gnome-term
 2697 howtogee  20   0  2820 1148  864 R  0.3  0.1   0:00.05 top
    1 root      20   0  3456 1976 1280 S  0.0  0.2   0:02.31 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.07 ksoftirqd/
```

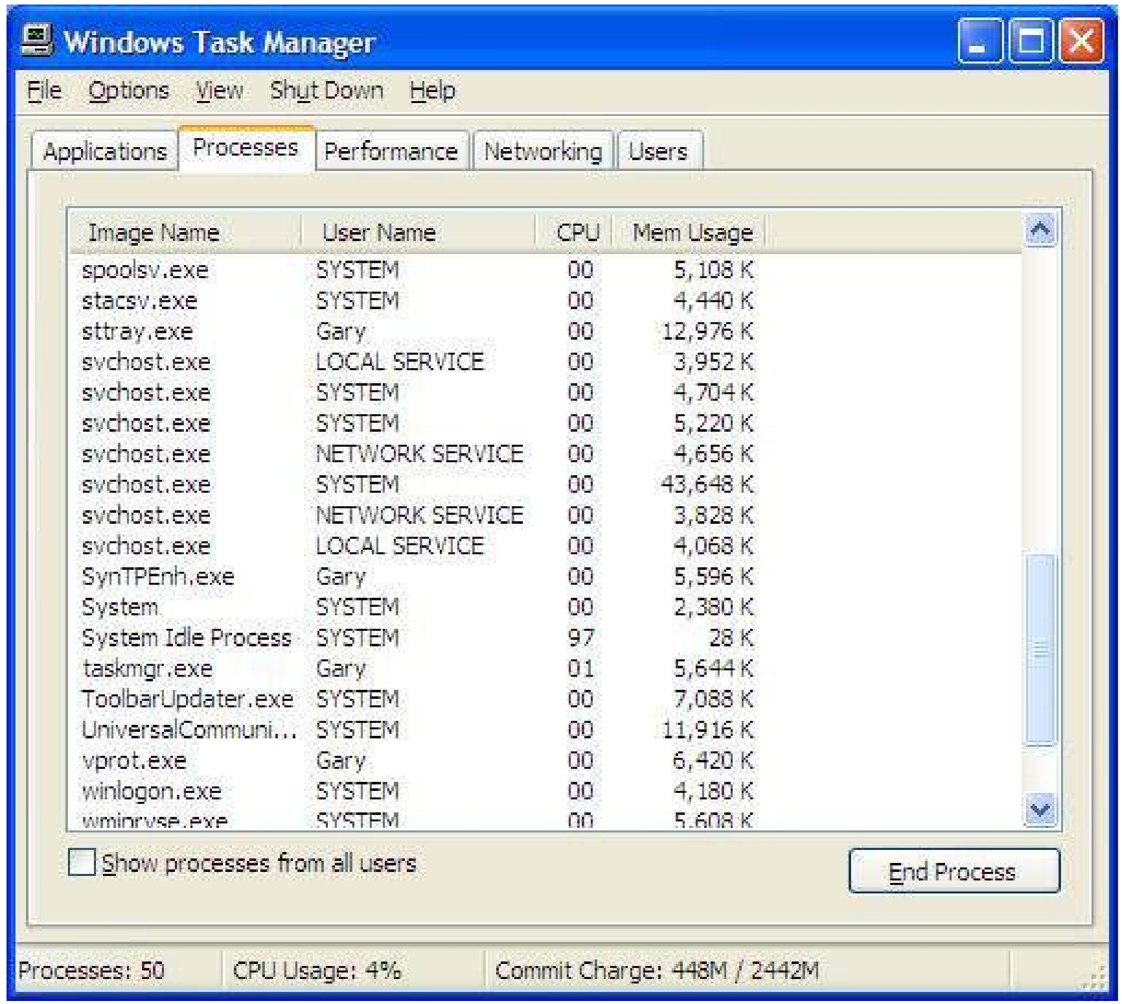
- An instance of a program, replete with registers, variables, and a program counter
- Type `TOP` or **`ps -a`** at your shell to see the processes running.
- Now try, **`ps -aef –forest`** or **`ps f`** to see the hierarchical process tree

What are Processes?

The above system calls launched system processes!

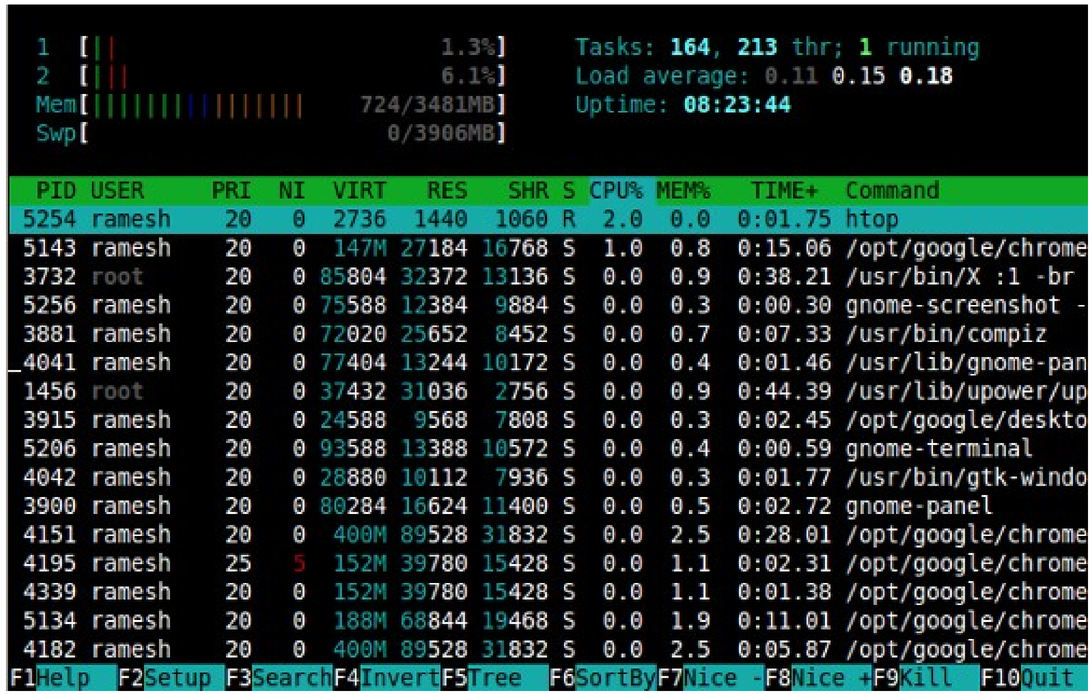


What are Processes?



- Processes in Windows using the Task Manager

What are Processes?

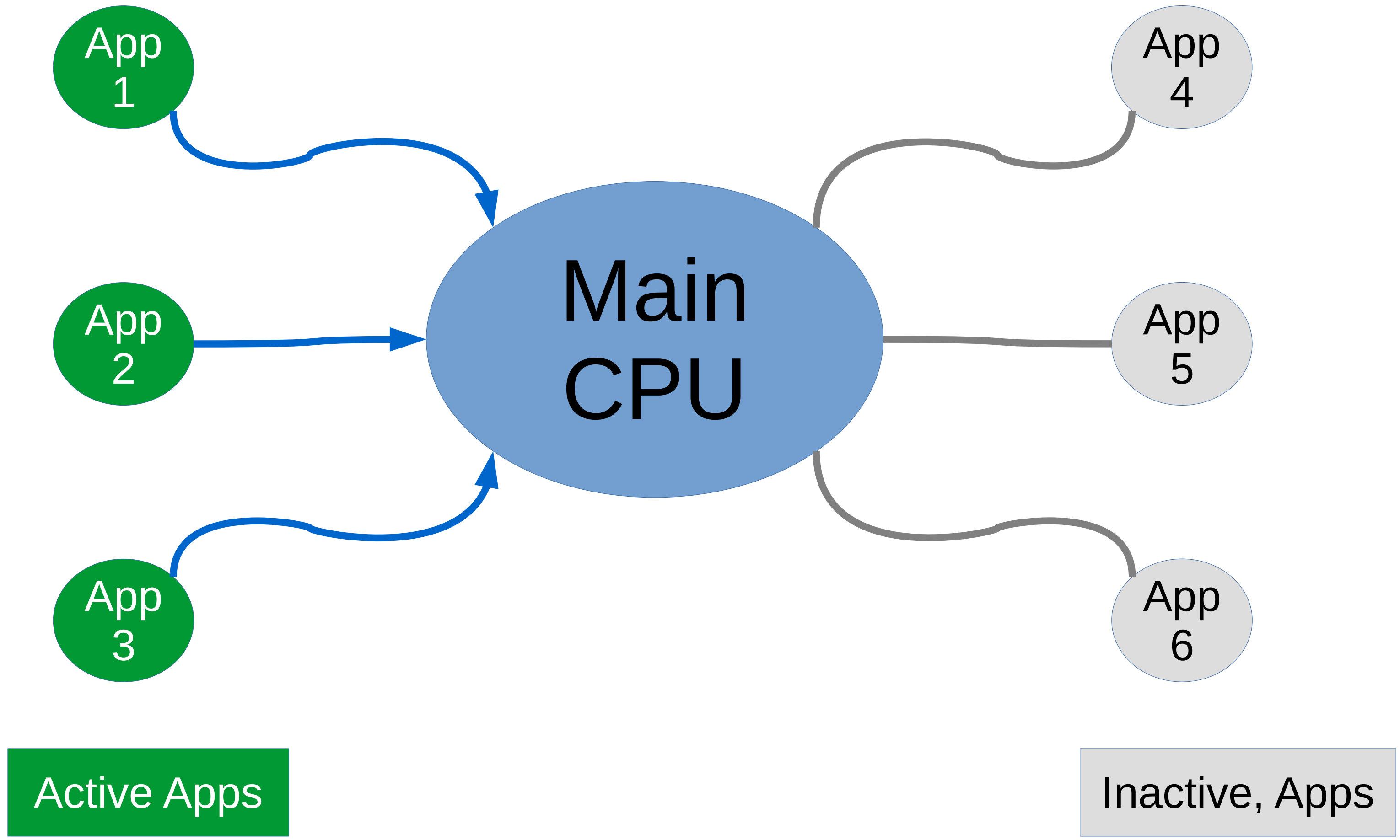


- If installed, *htop*, is another way to see the system processes in Linux.
- Why is this idea of *unique processes* necessary?
  - A computer manages many computations concurrently-need an abstraction to describe how it does it

So, What are Processes?

- Unique tasks performed by the OS
- Tools on your computer that run concurrently:
  - Browser: different tabs open
  - Media players
  - Editors : often with several different docs open
  - Terminal: multiple sessions open
  - And etc.
- Each of these tasks is a separate application running on the CPU, using memory.

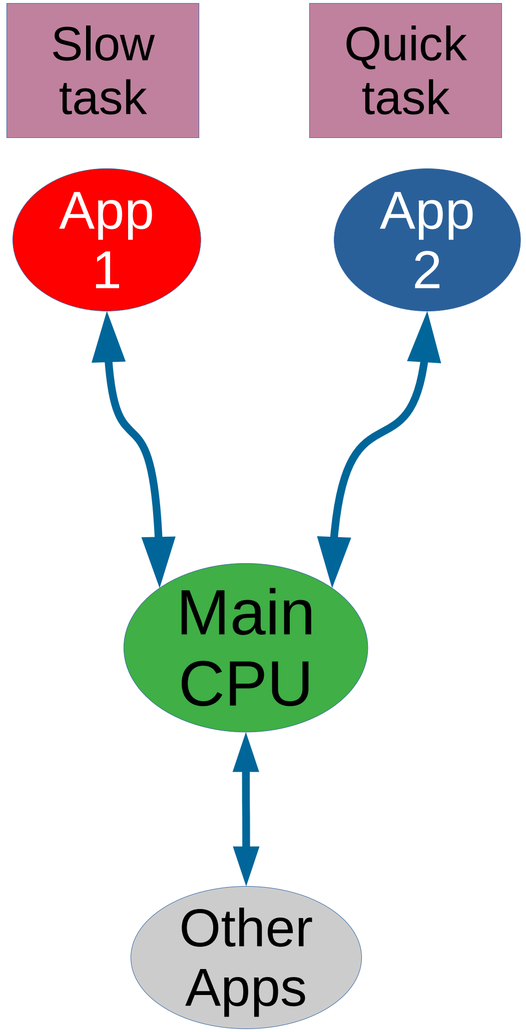
How the CPU Gets Things Done



- Each application, active or inactive, is a process that is running on the machine.

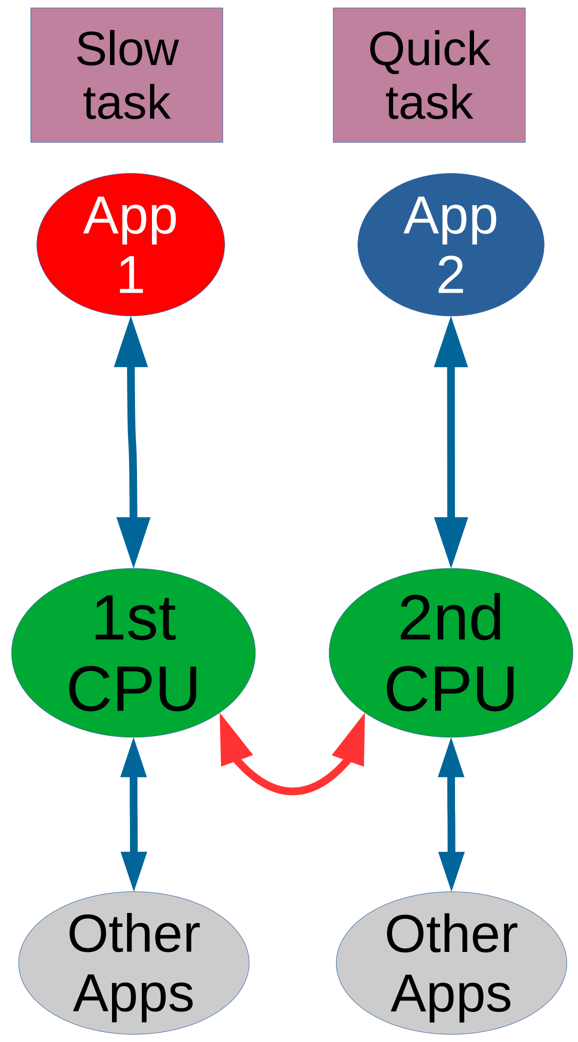
A CPU Running Two Processes

- All apps running on a **single-processor** (one CPU) are *time-sharing* resources and are ultimately running in *serial*.
- For CPUs, a second is plenty of time to be productive.
- Built-in time management
  - **Simple Example**: App1 (a slow task) and App2 (a quick task) awaits CPU focus
  - Since App1 takes at least a second to load, CPU does not wait. App2 is initiated at this time so as not to waste the CPU cycles.
  - When App2 begins its processing, focus is quickly returned to App1 for status.
  - App2 is again a CPU focus and completes.
  - App1 regains the focus and complets.



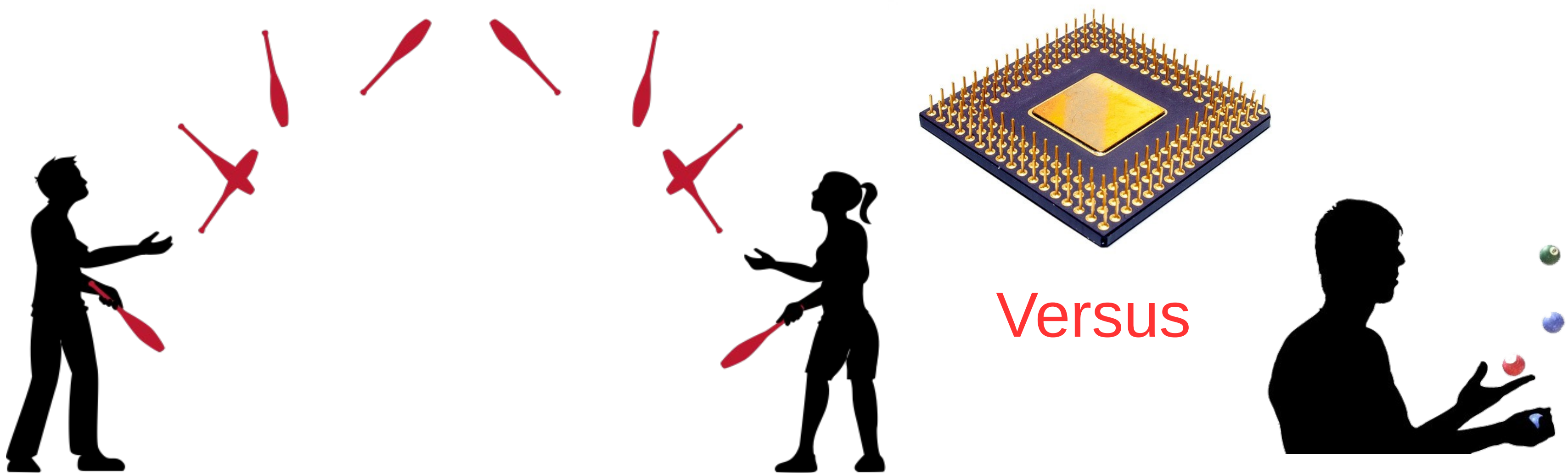
Two CPUs Running Two Processes

- All apps running on **multiprocessor** (two or more CPUs) machines run in *parallel*.
- Built-in time management
  - App1 gets the focus of 1st CPU, loads, runs and completes.
  - App2 gets the focus of 2nd CPU, loads, runs and completes.
- The CPUs are able to communicate between themselves to decide which one should pick-up which tasks.
- Computing speed is build into this model
- *What would happen to the speed with more CPUs available for tasks?*

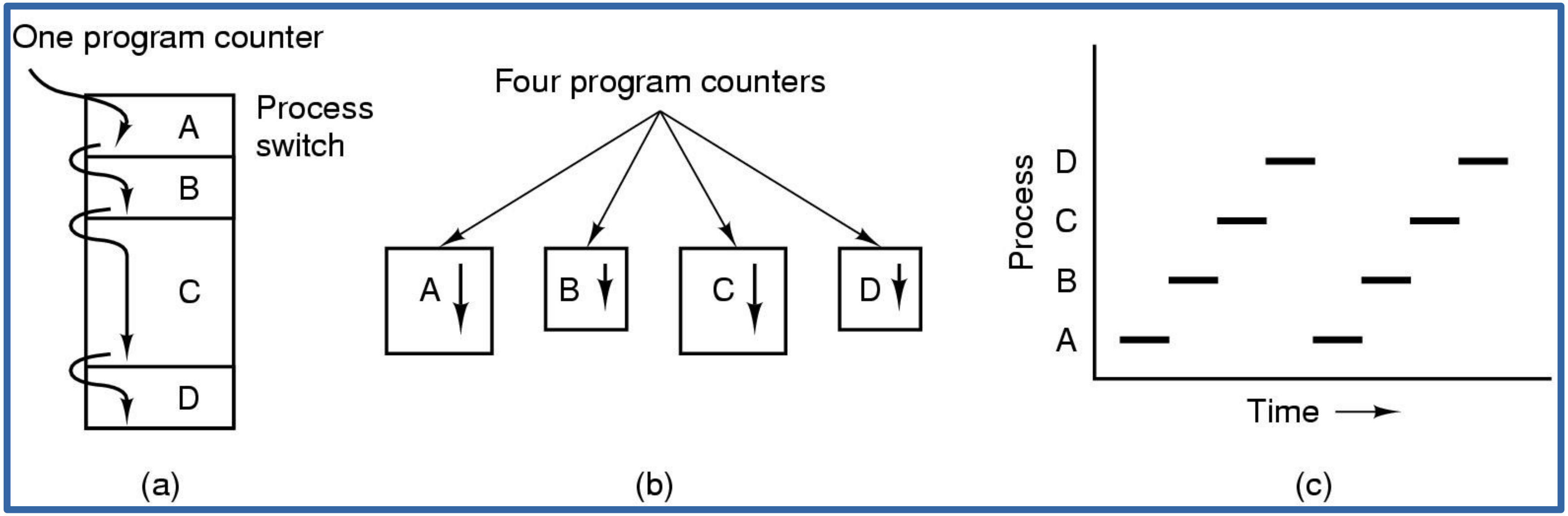


Terms

- **Parallelism** – The ability to run two or more tasks at exactly the same time.
- **Pseudoparallelism** – The illusion of running tasks simultaneously. A *time-sharing* approach.
- **True Parallelism** – Running *n* processes on *n* CPUs at the exact same time (using multiprocessor systems).
- **Multiprogramming** - The CPU switches from task to task. No task is actually run at the exact same time.
  - In our textbook's discussion, we generally assume a single CPU.



Multi-Programming Models



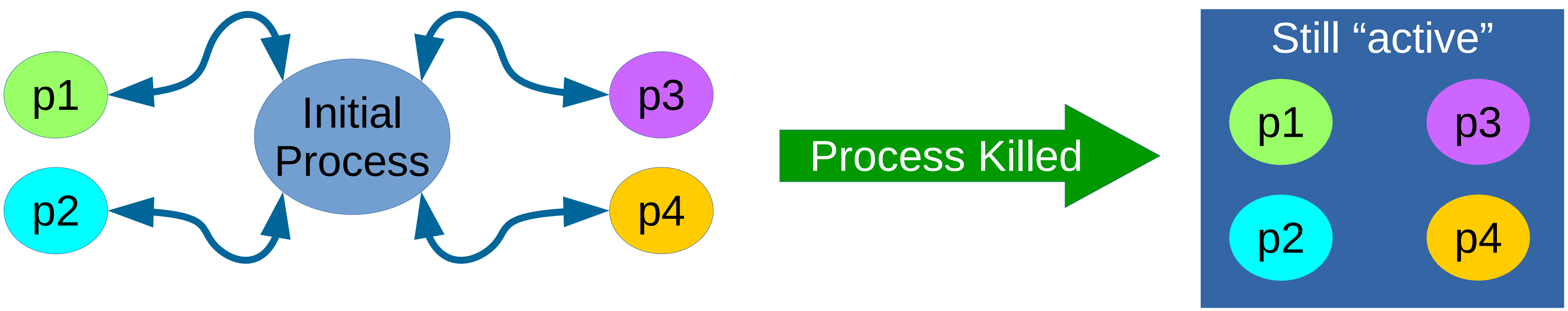
- A) *Multiprogramming* of four programs
- B) Conceptual model of four independent, sequential processes
- C) Only one program is active at a time.

Create a Process

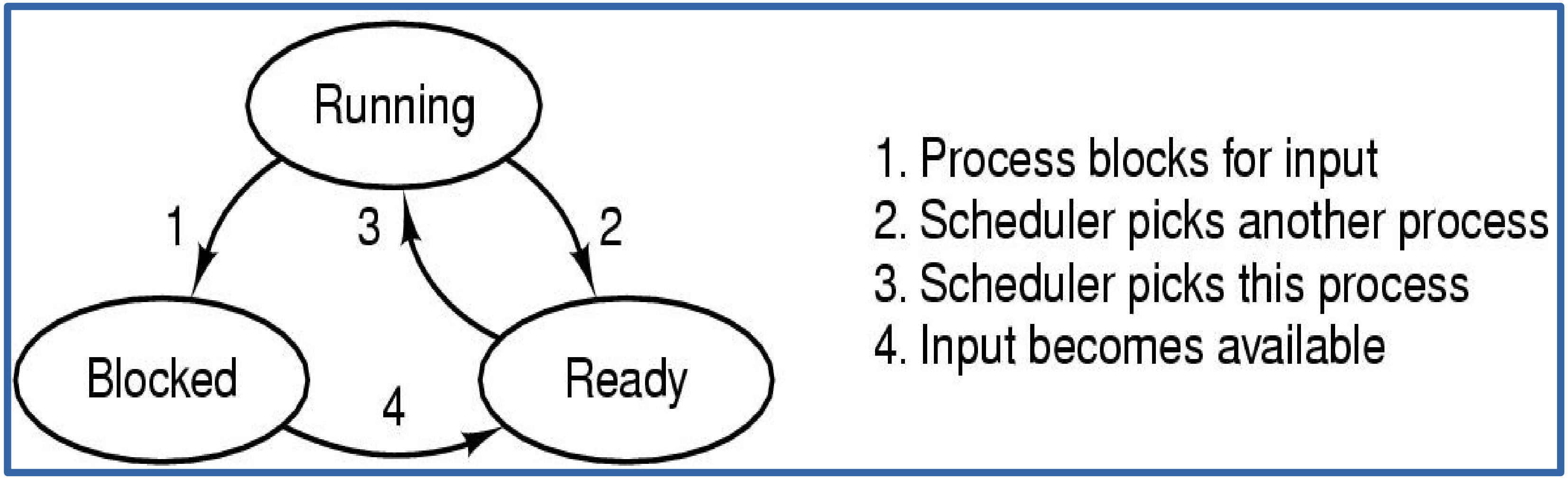
- Events to initiate the creation of a process
  - System initialization
  - Execution of a process creation system call by a running process.
  - A user request to create a new process
  - Initiation of a batch job

Terminating a Process

- Events which cause process termination:
  - Normal exit (voluntary).
  - Error exit (voluntary).
  - Fatal error (involuntary).
  - Killed by another process (involuntary).
- Upon termination in UNIX and Windows, the associated processes, as created by an initial process, are not terminated.
- **Why is it desirable to not crash all processes when one crashes??**

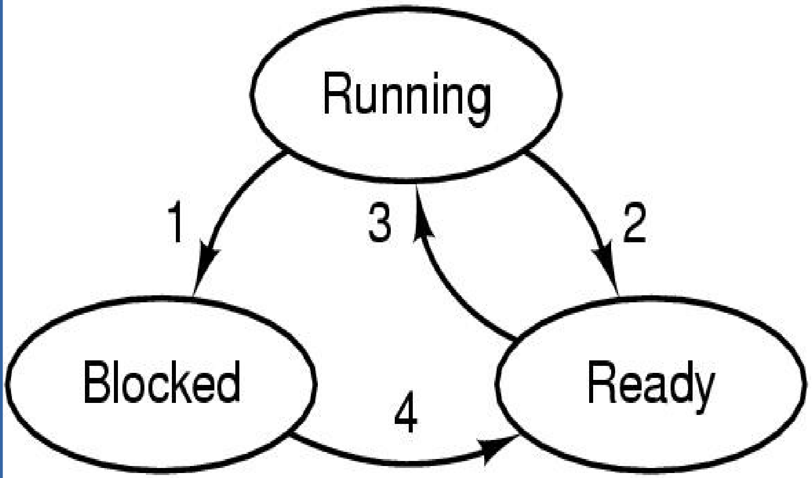


The States of Processes



- A process can be *running*, *blocked*, or in the *ready state*. Here we note the transition of these states.
- **Running** – Actually using the CPU at a particular moment
- **Ready** – Able to run, but waiting (queued to start)
- **Blocked** – Unable to run, waiting for external event

The States of Processes



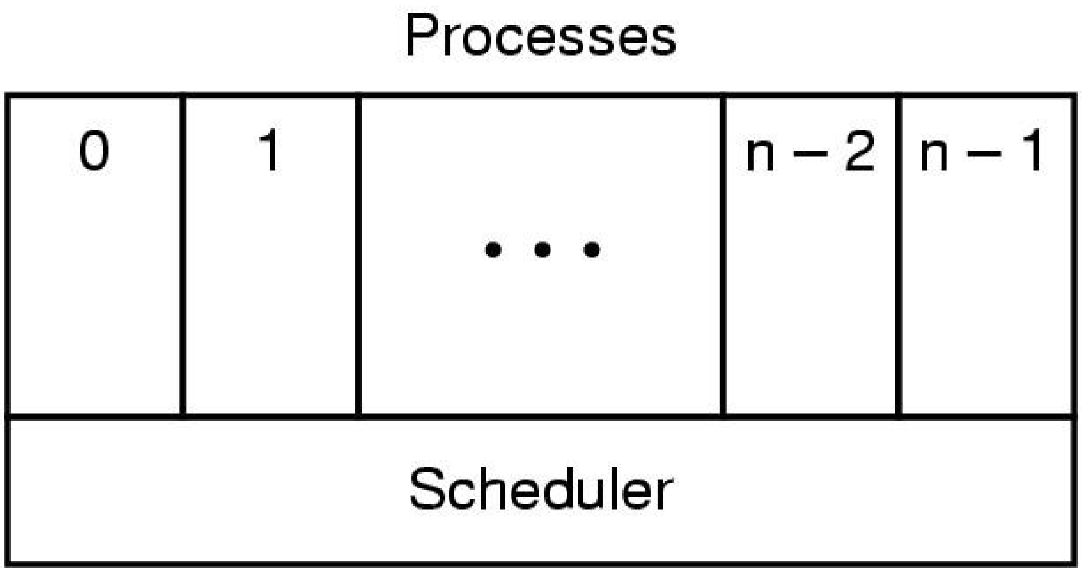
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- When the CPU switches back and forth among the processes, the rate for which the process performs its computation is not consistent and unlikely to be reproduced with similar circumstances.
- Why could this rate be inconsistent?

What is the Difference Between a **Program** and a **Process**?

- A **program** has a list of events which must be completed for termination. Each event may be completed with some time-lag spent in-between them. Programs may be interrupted and run in one priority.
- A **process** has differing priority levels. Higher priority levels may determine that the process must begin and finish when initiated.

Implementation of Processes (I)



- Priority: the lowest layer of the process-structured OS handles the *interrupts* and *scheduling* of processes.
- Above the handling layer are the sequential processes.
- Each column concerns an aspect of the process.

Managing Processes

Managing Processes, and which is able to run at a time, is similar to running all the planes that take-off and land at a busy airport.

The Process Table:
Keeping track of processes

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

- An OS table used to implement processes.
- Some of the fields of a typical process table entry.

The OS Managing an Interrupt for a Process

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

- The steps of what the OS does when processing a system interrupt.
- Steps {1, 2, 3, 4} – Receive the interrupt, save state of processes running prepare to run new process from a stack.
- Steps {5, 6, 7} – Register the new process, let the scheduler decide the order to run processes, go back to the assembly code (library) to setup the new process.
- Step {8} – Run the new process.

Consider This!

- We have been talking about processes which are handled by a CPU according to priority.
- Discuss why a priority of one process may be higher than another.
- How does this influence the way that the CPU handles these processes? Why?