Processes

Kartik Gopalan

References:

- Chapter 2 of the Tanenbaum's book
- Chapter 4 of OSTEP book
- man pages in any UNIX/Linux system

Process versus Program

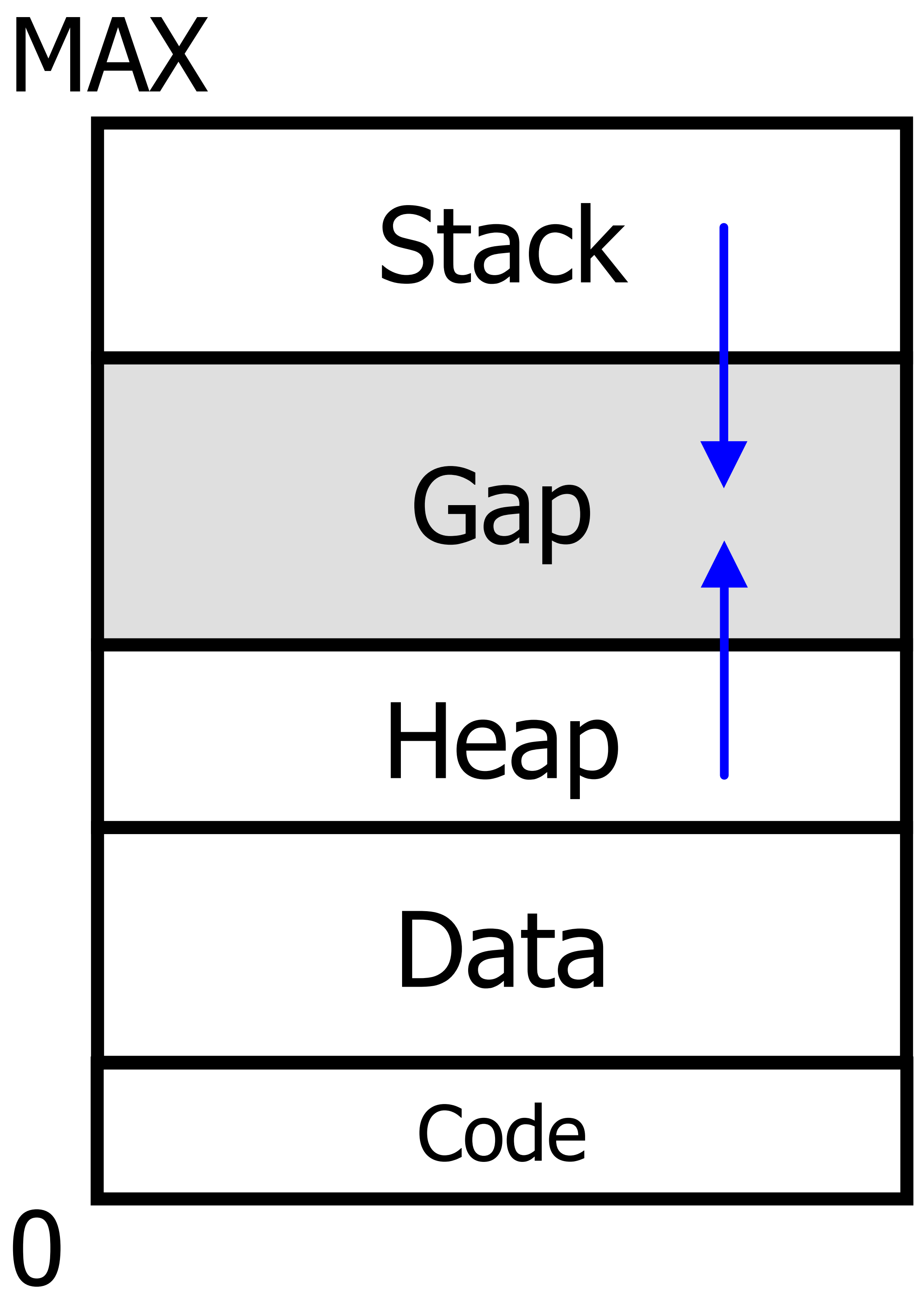
- Program is a passive executable file stored in the disk
  - Contains static code and static data

- Process is a program in execution.

- There can be multiple processs running the same program
  - Example: many users can run “ls” at the same time

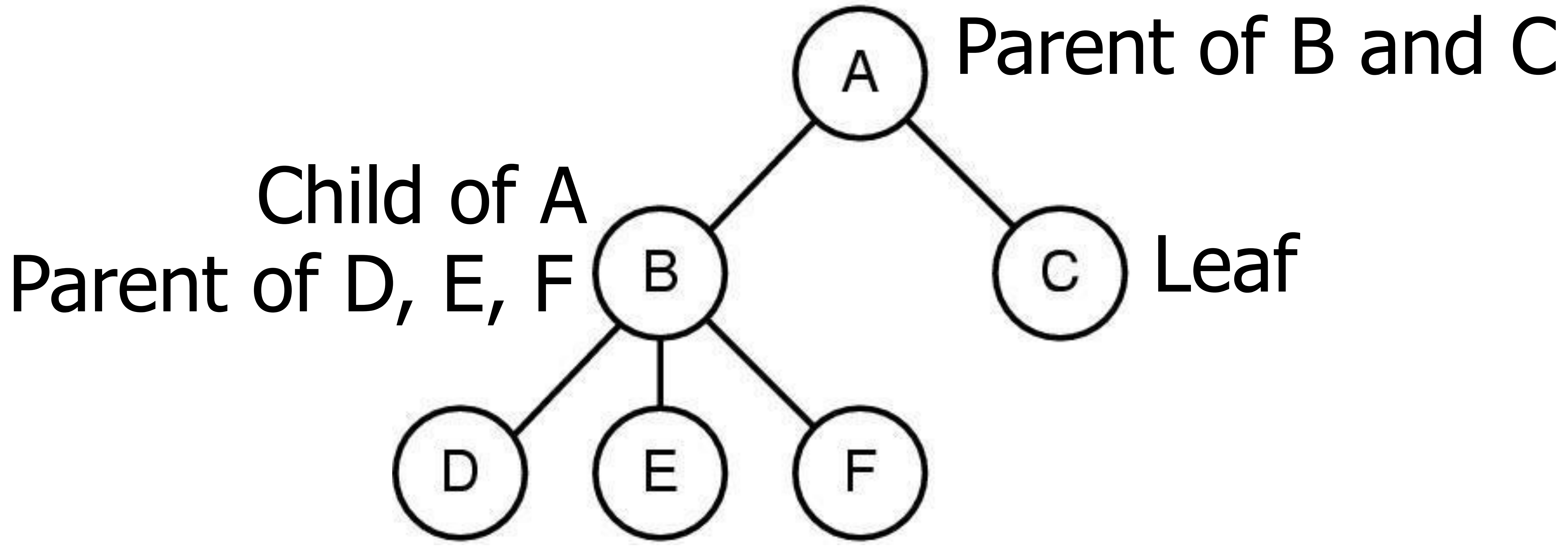
What's in a process?

- Memory
  - Code
  - Static and dynamic data
  - Procedure call stack
- CPU state
  - Program counter
  - Stack pointer
  - General purpose registers, etc
- I/O state
  - Open files, devices, network



Typical memory layout of a process

Process Hierarchy Tree



- A created two child processes, B and C
- B created three child processes, D, E, and F

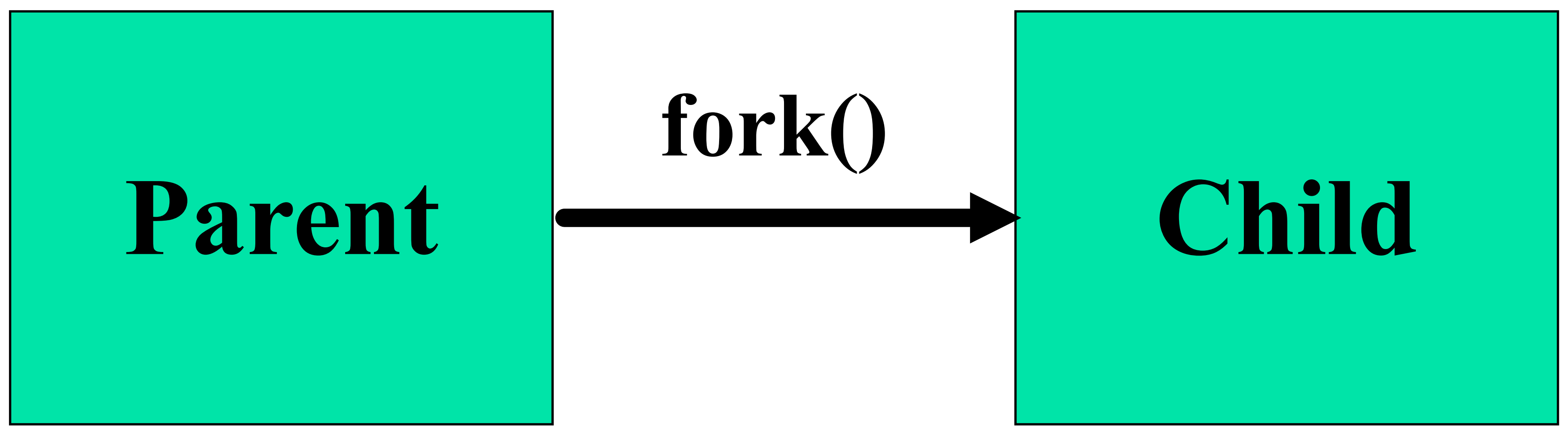
System calls to control a process

- fork() - Create a process
- exec() - Run a new program
  - More accurately: Replace the current process with a new program image
- wait() or waitpid() - wait for a child process to terminate
- exit() - Terminate the calling process

Example : fork() and waitpid()

https://oscourse.github.io/examples/fork_ex.c

```
pid = fork();

if (pid < 0) {
        perror("fork failed:”);
        exit(1);
}

if (pid == 0) { // Child executes this block
        printf(“This is the child\n");
        exit(0);
}

if (pid > 0) { //Parent executes this block

  printf(“This is parent. The child is %d\n", pid);

ret = waitpid(pid, &status, 0);
if (ret < 0) {
   perror(“waitpid failed:”)
   exit(2);
}

printf(“Child exited with status %d\n”, status);
exit(0);
```



- fork() is called once, but it returns twice!!
  - in the parent and the child
- Child is an exact “copy” of parent.
- Return value of fork in child = 0
- Return value of fork in parent = [process ID of child]
- fork’s return value lets the parent and child take different code paths.

exec() - Example code

https://oscourse.github.io/examples/exec_ex.c


```
if ((pid = fork()) < 0) {
 fprintf(stderr, "fork failed\n");
 exit(1);
}

if (pid == 0) {
 if( execlp("echo", "echo", "Hello from
    the child", (char *) NULL) == -1)
     fprintf(stderr, "execl failed\n");

 exit(2);
}

printf("parent carries on\n");
```

- exec() replaces the caller's memory with a new program image.
- exec() is called once but doesn't return!!
- All I/O descriptors that were open before exec() stay open after exec().
  - I/O descriptors = file, socket, pipe etc.
- This property is very useful for implementing filters.

Different Types of exec()

- `int execl(char * pathname, char * arg0, … , (char *)0);`
  - Full pathname + long listing of arguments

- `int execv(char * pathname, char * argv[]);`
  - Full pathname + arguments in an array

- `int execle(char * pathname, char * arg0, … , (char *)0, char envp[]);`
  - Full pathname + long listing of arguments + environment variables

- `int execve(char * pathname, char * argv[], char envp[]);`
  - Full pathname + arguments in an array + environment variables

- `int execlp(char * filename, char * arg0, … , (char *)0);`
  - Short pathname + long listing of arguments

- `int execvp(char * filename, char * argv[]);`
  - Short pathname + arguments in an array

- More info: check “man 3 exec”

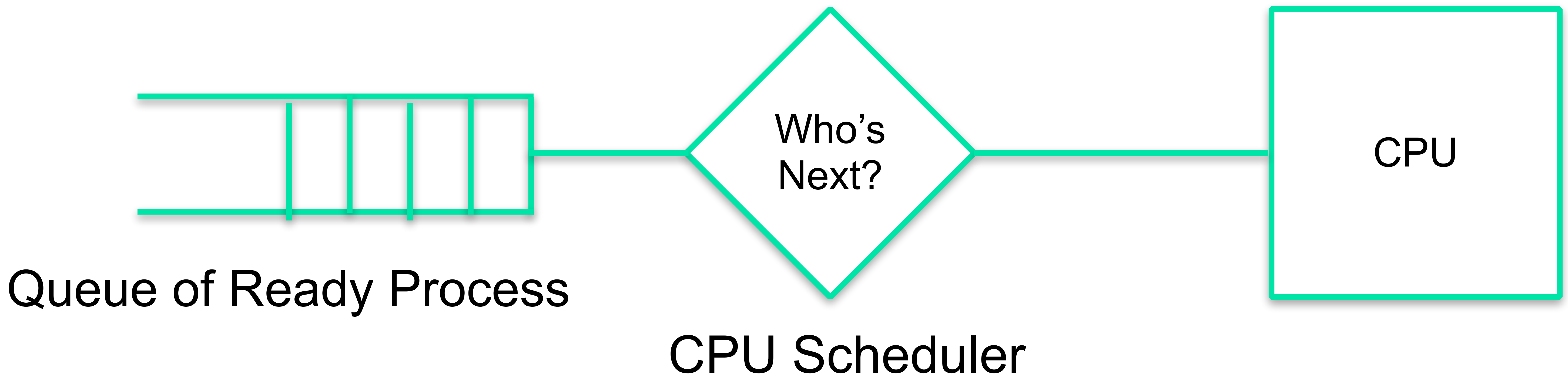
wait() and exit()

- wait() and waitpid()
  - Called by parent to wait for child to terminate

- Terminating a process
  - Either return from main()
  - Or call exit(status) anywhere in the code
    - Status is retrieved by the parent using wait().
    - 0 for normal status, non-zero for error

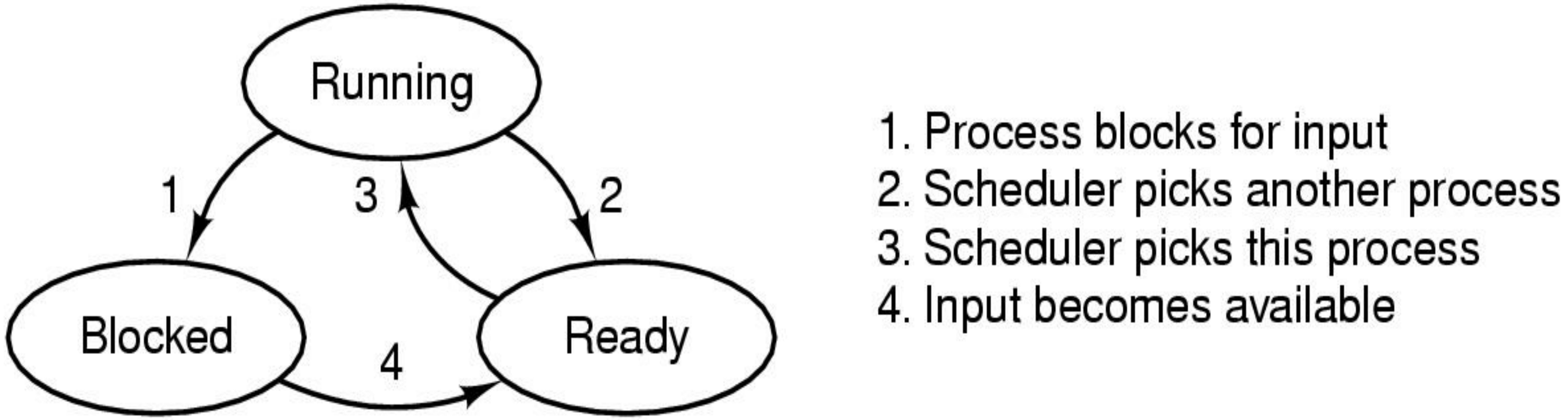
Scheduling processes on the CPU

- Time-sharing (temporal multiplexing)
  - Many processes share one or more CPUs



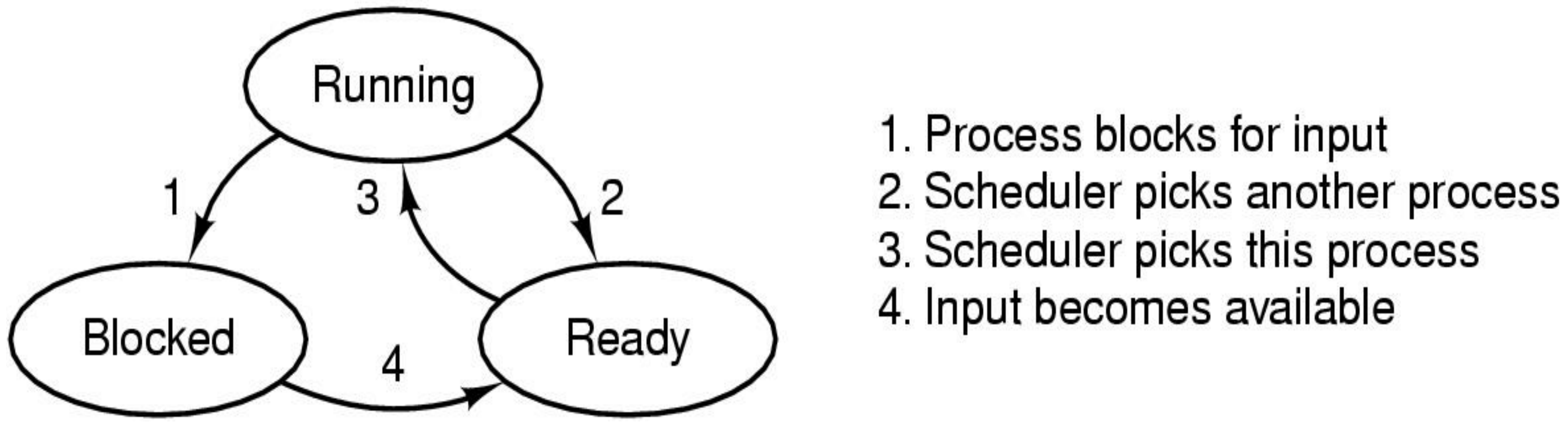
- Scheduling algorithms depend on performance objectives
  - Round-robin, FIFO, Shortest Job First, Fair scheduling etc
- Linux implements CFS
  - So-called "completely" fair scheduling

Process Lifecycle



- Ready
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running
  - Process is executing on the CPU
- Blocked
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue.

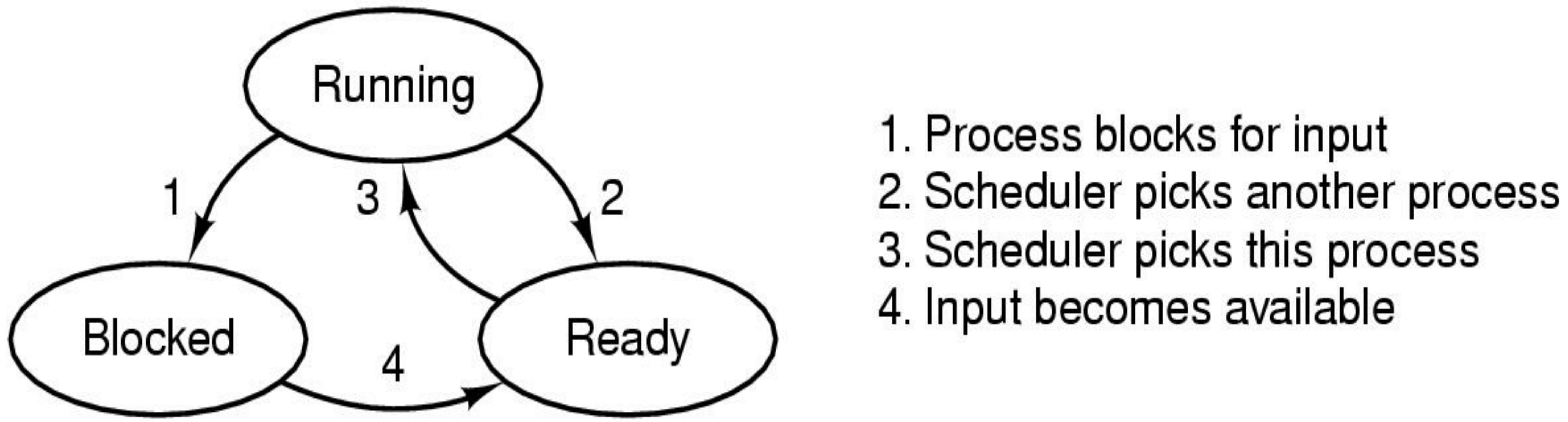
Example 1: Multiple processes sharing CPU



Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Figure 4.3: **Tracing Process State: CPU Only**

Example 2: Multiple processes sharing CPU



Time	**Process$_0$**	**Process$_1$**	**Notes**
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process$_0$ initiates I/O
4	Blocked	Running	Process$_0$ is blocked,
5	Blocked	Running	so Process$_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process$_1$ now done
9	Running	–	
10	Running	–	Process$_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Examining Processes in Unix/Linux

- ps command
  - Standard process attributes

- /proc directory
  - More interesting information if you are the root.

- top command
  - Examining CPU and memory usage statistics.

Orphans and Zombies

- <u>Orphan</u>
  - When a parent dies, child becomes an orphan process.
  - The init process (pid = 1) takes over as parent of the orphaned children.
  - Here's an example: https://oscourse.github.io/examples/orphan.c
  - Do a 'ps –l' after to check parent's PID of the orphan process.
  - After you are done remember to kill the orphan process 'kill –9 <pid>'

- <u>Zombie</u>
  - The child becomes a zombie when it terminates and it's parent doesn't call wait().
  - Status "Z" seen with ps.
  - Zombies cleared when parent eventually calls wait() or waitpid().
  - Zombies don't take up any system resources. Just an integer status is kept in the OS.