

Administrative

- Slides and Video up on website
- Project clarification up on website
 - Demo or Video - both need to be shown in class
- What time slots?
- Own laptops? Environment installed?
 - Native or VM
 - Linux
 - Python 3.x from Anaconda
 - GCC, GDB, htop, etc.
- Homework will be handed out by tomorrow, due on Saturday noon

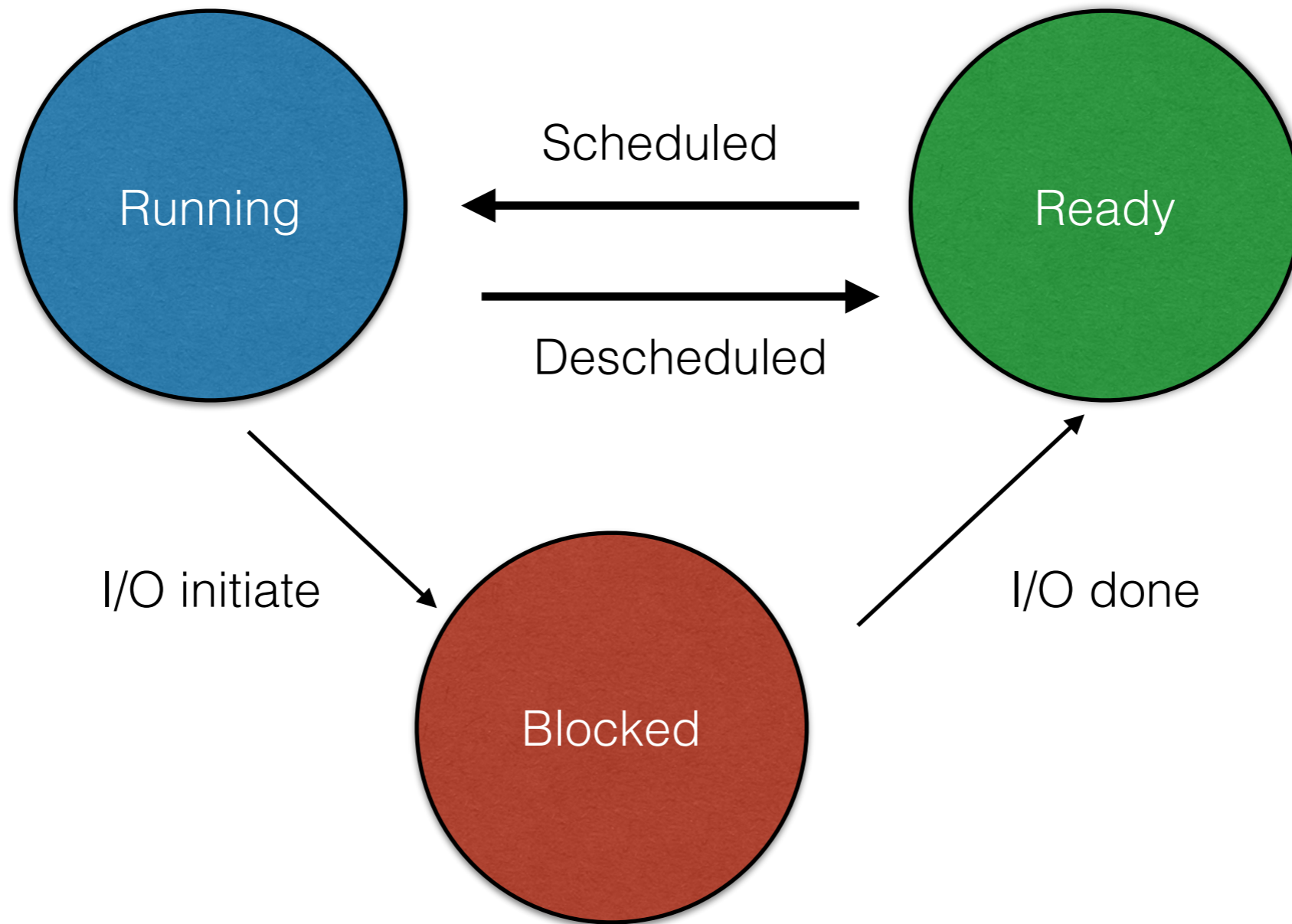
Operating Systems

Lecture 3: The Process API

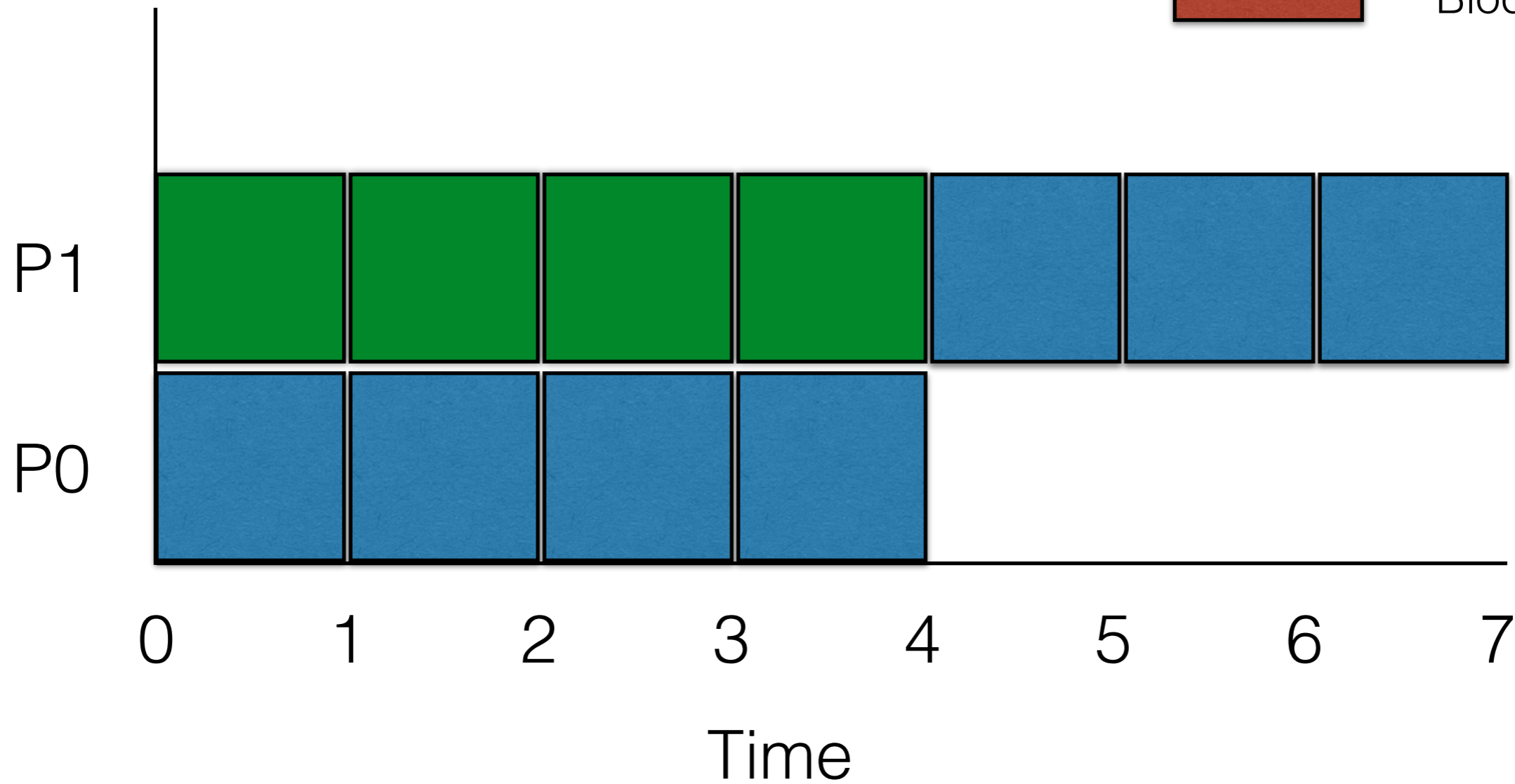
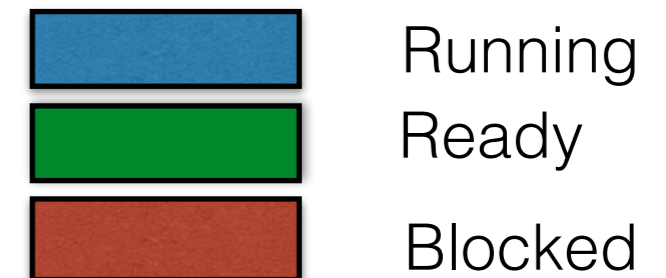
Nipun Batra
Aug 7, 2018

Process States

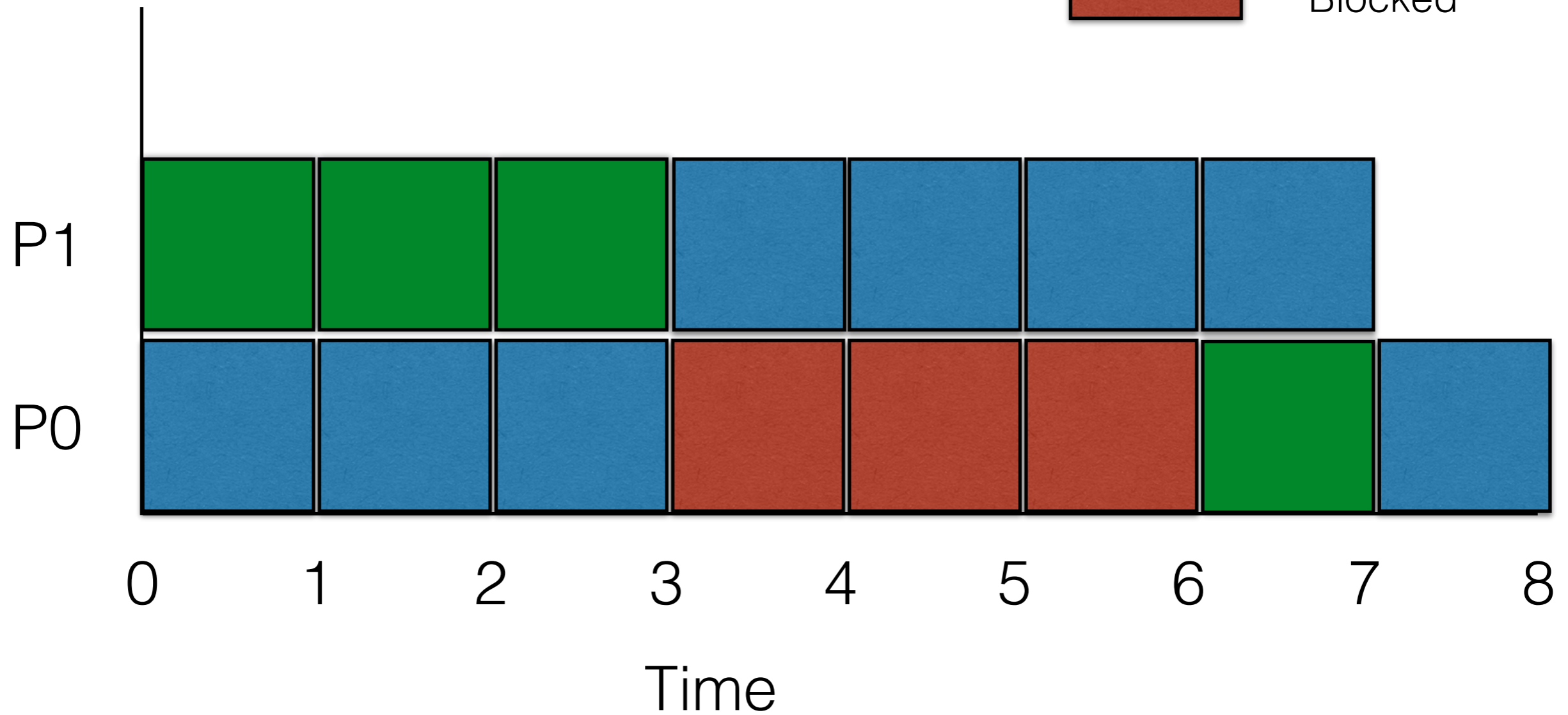
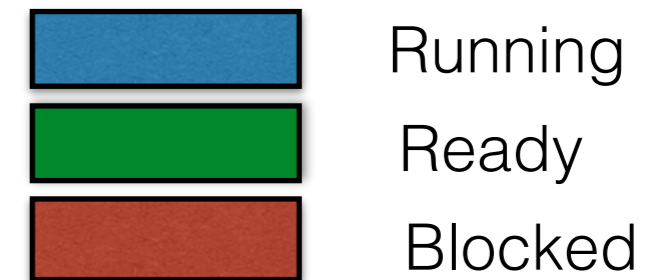
Process States



Process States



Process States



Revisiting process-run and Randomisation

Project
idea

Revisiting process-run and Randomisation

Project idea

www.pythontutor.com

Work New folder New folder Google Analytics Outlook Web App imac - Google Sear...

Python Tutor (code on GitHub) supports seven languages (despite its name!):

1. Python 2.7 and 3.6 with limited module imports and no file I/O. The following modules are supported: collections, copy, datetime, functools, hashlib, heapq, itertools, json, math, operator, typing, io/stringIO. [Backend source code.](#)
2. Java using Oracle's Java 8. The original [Java visualizer](#) was created by David B. Smith. It supports [all the standard Java libraries](#), [Stack](#), [Queue](#), and [GC](#). (To use it, you need to write `import java.util.*` at the top of your code.)
3. JavaScript running in Node.js v6.0.0 with limited support for ES6. [Backend source code.](#)
4. TypeScript 1.4.1 running in Node.js v6.0.0. [Backend source code.](#)
5. Ruby 2 using MRI 2.2.2. [Backend source code.](#)

CHATBOT: Someone from San Diego, California, US just joined this chat.

Type your message here

4. TypeScript 1.4.1 running in Node.js v6.0.0. [Backend source code.](#)

5. Ruby 2 using MRI 2.2.2. [Backend source code.](#)

6. C using gcc 4.8, C11, and Valgrind Memcheck. [Backend source code.](#)

7. C++ using gcc 4.8, C++11, and Valgrind Memcheck. [Backend source code.](#)

Privacy Policy: By using Online Python Tutor, your visualized code, email address are logged on our server and may be analyzed for research purposes. However, the Online Python Tutor web site does not store any personal information or session state from users, nor does it issue cookies.

Copyright © Philip Guo. All rights reserved.

Type your message here

Here is a Python example:

Python 2.7

```
1 def listSum(numbers):
2     if not numbers:
3         return 0
4     else:
5         (f, rest) = numbers
6         return f + listSum(rest)
7
8 myList = (1, (2, (3, None)))
9 total = listSum(myList)
```

[Edit this code](#)

→ line that has just executed
→ next line to execute

< Back Step 20 of 22 Forward >

[Python Tutor](#) by [Philip Guo](#). Support with a [small donation](#).

Frames Objects

Global frame

- listSum → function listSum(numbers)
- myList → tuple (1, (2, (3, None)))

listSum

- numbers → tuple (1, (2, (3, None)))
- f → 1
- rest → tuple (2, (3, None))

listSum

- numbers → tuple (2, (3, None))
- f → 2
- rest → tuple (3, None)

listSum

- numbers → tuple (3, None)
- f → 3
- rest → None
- Return value → 3

Process API

- **Create process:**
 - Double click
 - Run on command line
- Destroy processes:
 - Task manager
 - Command line
- **Wait:**
 - Don't run process till other process completes
- Status:
 - How long run, what state it is in
 - Does top, ps give us this info?
- Misc.:
 - Suspend

Process API

- **Create process:**
 - `fork()`
 - `exec()`
- **Wait:**
 - `wait()`

The **fork()** System Call

> man fork

```

FORK(2)                                BSD System Calls Manual                                FORK(2)

NAME
    fork -- create a new process

SYNOPSIS
    #include <unistd.h>

    pid_t
    fork(void);

DESCRIPTION
    fork() causes creation of a new process. The new process (child process) is an exact copy of the calling
    process (parent process) except for the following:

    o The child process has a unique process ID.

    o The child process has a different parent process ID (i.e., the process ID of the parent process).

    o The child process has its own copy of the parent's descriptors. These descriptors reference the
      same underlying objects, so that, for instance, file pointers in file objects are shared between the
      child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subse-
      quent read or write by the parent. This descriptor copying is also used by the shell to establish
      standard input and output for newly created processes as well as to set up pipes.

    o The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES
    Upon successful completion, fork() returns a value of 0 to the child process and returns the process ID of the
    child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child
    process is created, and the global variable errno is set to indicate the error.
```

fork() demo

fork() demo

1. `fork_demo_1.c` : Get the PID of current process

fork() demo

1. `fork_demo_1.c` : Get the PID of current process
2. `fork_demo_2.c` : Get the PID of parent process

fork() demo

1. `fork_demo_1.c` : Get the PID of current process
2. `fork_demo_2.c` : Get the PID of parent process
3. Use `ps` (`man ps` to find more) to find what's the parent process?

fork() demo

1. fork_demo_1.c : Get the PID of current process
2. fork_demo_2.c : Get the PID of parent process
3. Use ps (man ps to find more) to find what's the parent process?
4. See the above in Activity Monitor

fork() demo

1. `fork_demo_1.c` : Get the PID of current process
2. `fork_demo_2.c` : Get the PID of parent process
3. Use `ps` (`man ps` to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. `fork_demo_3.c`: Add sleep to view more details in Activity Monitor

fork() demo

1. `fork_demo_1.c` : Get the PID of current process
2. `fork_demo_2.c` : Get the PID of parent process
3. Use `ps` (`man ps` to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. `fork_demo_3.c`: Add sleep to view more details in Activity Monitor
6. `fork_demo_4.c`: Use `fork()` to create child process

fork() demo

1. fork_demo_1.c : Get the PID of current process
2. fork_demo_2.c : Get the PID of parent process
3. Use ps (man ps to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. fork_demo_3.c: Add sleep to view more details in Activity Monitor
6. fork_demo_4.c: Use fork() to create child process
7. fork_demo_5.c: Add sleep to above and find these processes on Activity Monitor

fork() demo

1. fork_demo_1.c : Get the PID of current process
2. fork_demo_2.c : Get the PID of parent process
3. Use ps (man ps to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. fork_demo_3.c: Add sleep to view more details in Activity Monitor
6. fork_demo_4.c: Use fork() to create child process
7. fork_demo_5.c: Add sleep to above and find these processes on Activity Monitor
8. Show the same using ps command

fork() demo

1. fork_demo_1.c : Get the PID of current process
2. fork_demo_2.c : Get the PID of parent process
3. Use ps (man ps to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. fork_demo_3.c: Add sleep to view more details in Activity Monitor
6. fork_demo_4.c: Use fork() to create child process
7. fork_demo_5.c: Add sleep to above and find these processes on Activity Monitor
8. Show the same using ps command
 1. (ps -p 42693 -o pid,ppid)

fork() demo

1. fork_demo_1.c : Get the PID of current process
2. fork_demo_2.c : Get the PID of parent process
3. Use ps (man ps to find more) to find what's the parent process?
4. See the above in Activity Monitor
5. fork_demo_3.c: Add sleep to view more details in Activity Monitor
6. fork_demo_4.c: Use fork() to create child process
7. fork_demo_5.c: Add sleep to above and find these processes on Activity Monitor
8. Show the same using ps command
 1. (ps -p 42693 -o pid,ppid)

9. Fun: Keep finding parent process

The **fork()** System Call

Parent

....

....

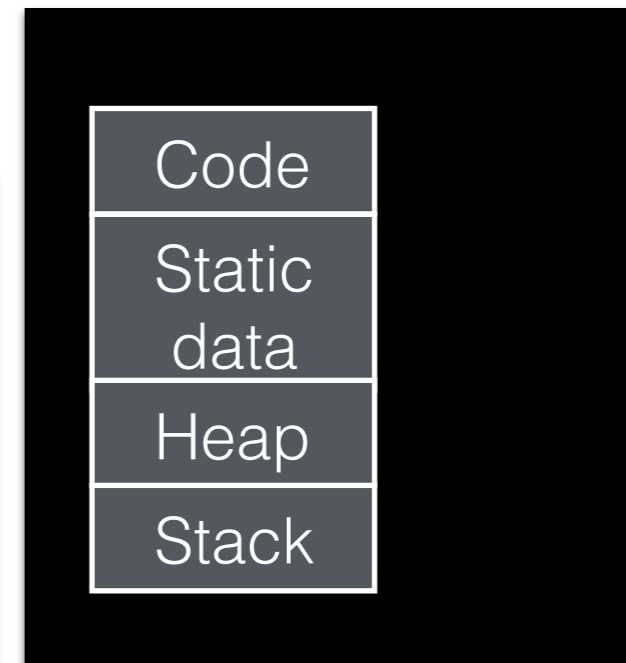
```
main(){  
  fork()
```

....

```
....}
```

The **fork()** System Call

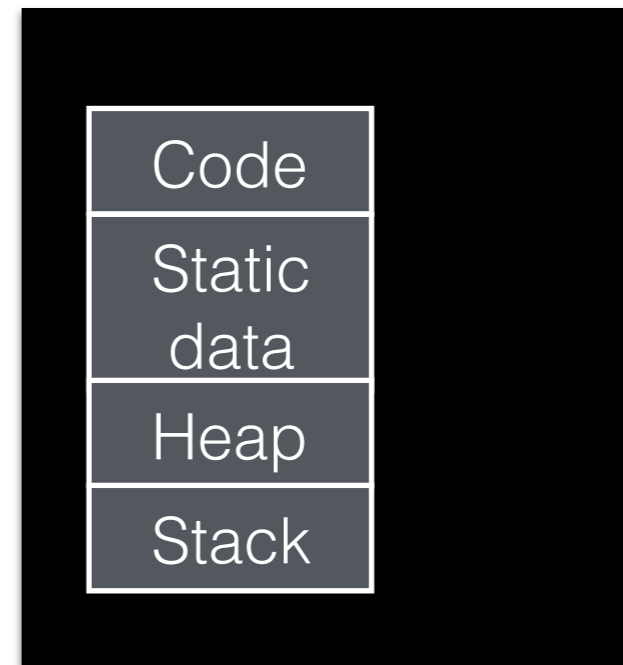
```
Parent
....
....
main(){
fork()
....
....}
```



Address
Space of
Parent

The **fork()** System Call

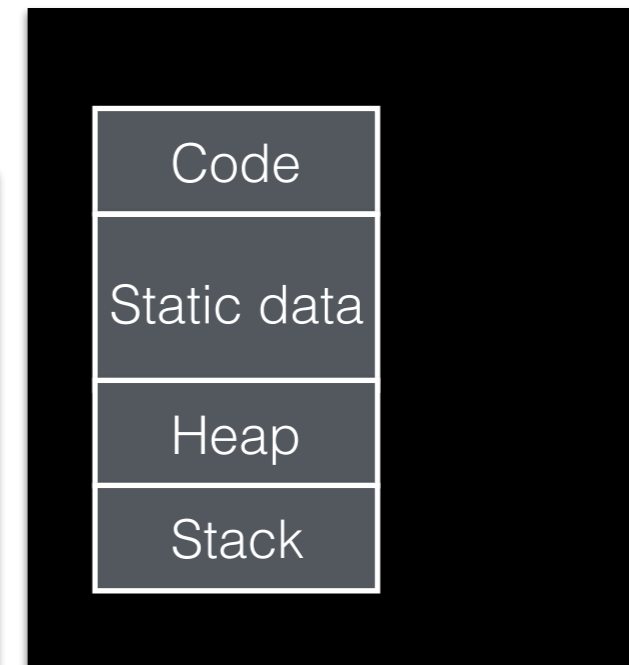
```
Parent
....
....
main(){
  fork()
  ....
....}
```



Address
Space of
Parent

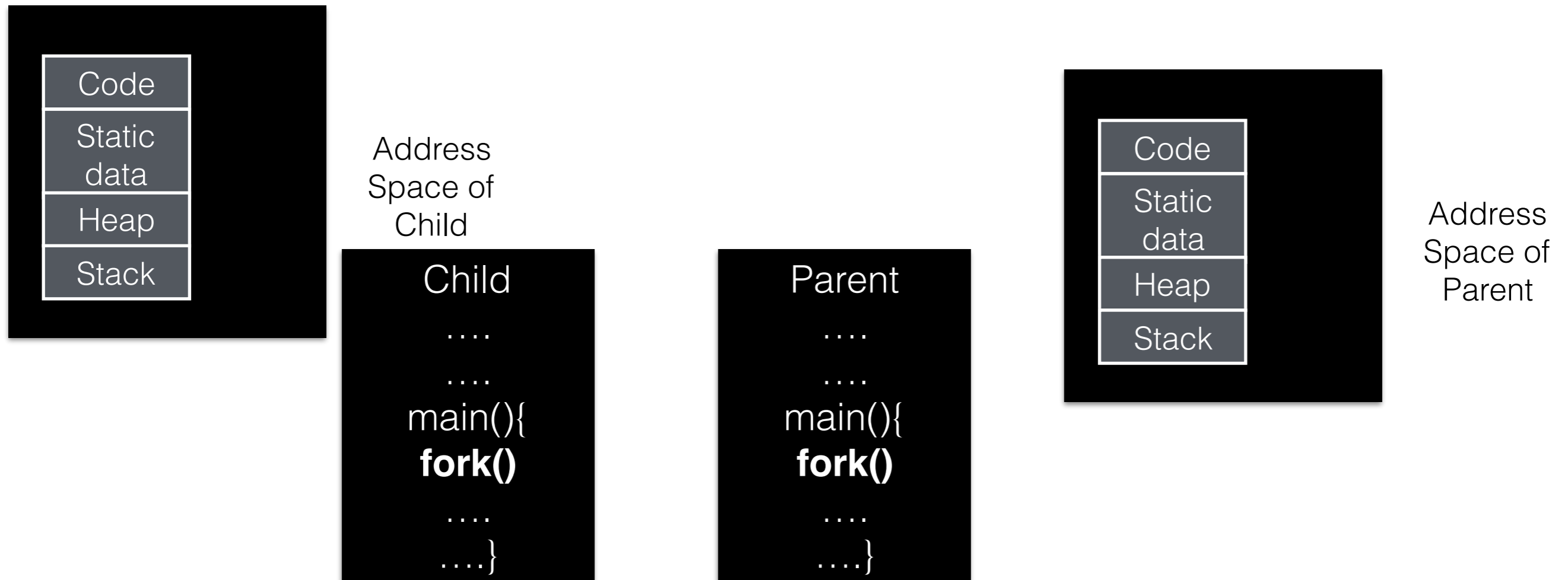
The **fork()** System Call

```
Parent
....
....
main(){
fork()
....
....}
```

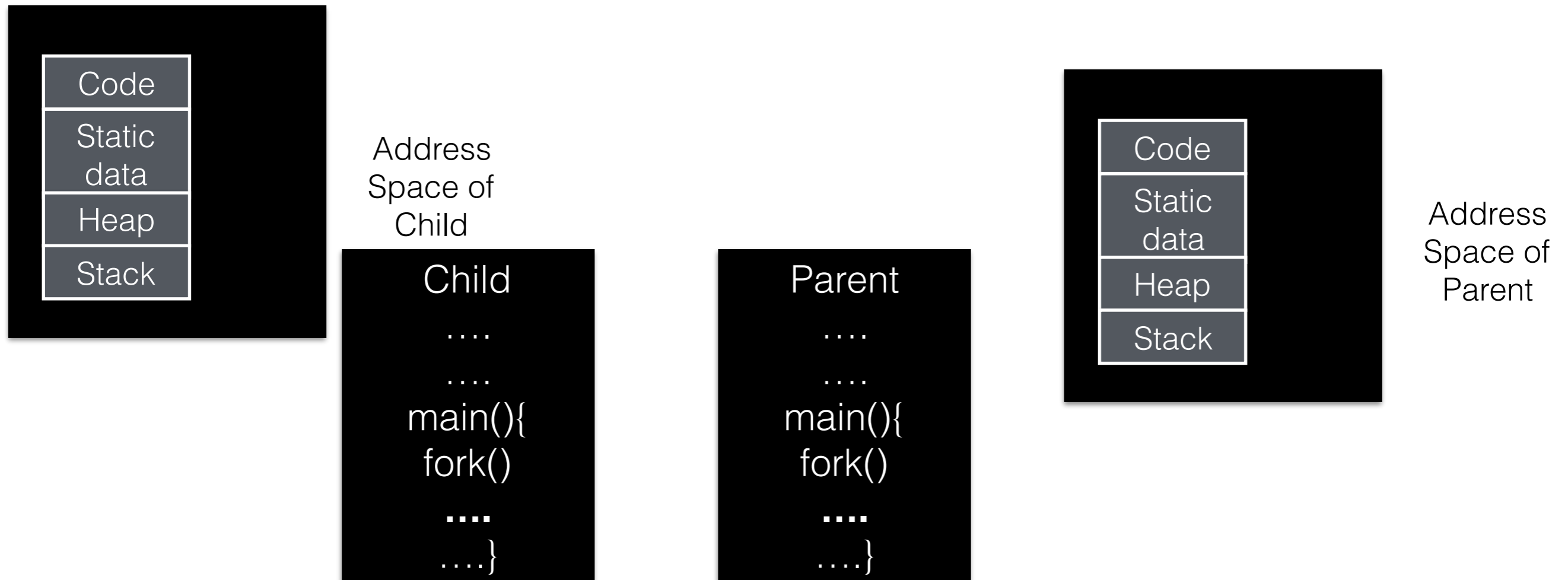


Address
Space of
Parent

The **fork()** System Call



The **fork()** System Call



fork() Code Usage in Linux repo

← → ↺

GitHub, Inc. [US] | https://github.com/torvalds/linux/search?q=fork&unscoped_q=fork

Apps

Work


New folder

New folder

Google Analytics

Outlook Web App

imac - Google Sear...



fork

/

Pull requests

Issues

Marketplace

Explore

Code

Commits

Issues

562

2K

11

Languages

C	418
Unix Assembly	44
Text	32
Shell	19
reStructuredText	16
Makefile	5
C++	3
Objective-C	2
Roff	1
Perl	1

Advanced search

Cheat sheet

562 code results in [torvalds/linux](#) or view [all results on GitHub](#)

Sort: Best match ▾

[tools/testing/selftests/powerpc/benchmarks/.gitignore](#)

Showing the top match Last indexed on Apr 8

```
1  gettimeofday
2  context_switch
3  fork
4  exec_target
5  mmap_bench
6  futex_bench
7  null_syscall
```

[tools/testing/selftests/ftrace/test.d/trigger/trigger-filter.tc](#)

Showing the top two matches Last indexed on Nov 18, 2017

```
19  echo "event tracing is not supported"
20  exit_unsupported
21  fi
22
23  if [ ! -f events/sched/sched_process_fork/trigger ]; then
...
33  echo 'traceoff if child_pid == 0' > events/sched/sched_process_fork/trigger
34  ( echo "forked")
35  if [ `cat tracing_on` -ne 1 ]; then
36  fail "traceoff trigger on sched_process_fork did not work"
```

Shell

[tools/testing/selftests/powerpc/tm/tm-fork.c](#)

C

The **wait()** System Call

Run fork_demo_4 again. Different order?

> man 2 fork

```
WAIT(2)                                BSD System Calls Manual                                WAIT(2)

NAME
    wait, wait3, wait4, waitpid -- wait for process termination

SYNOPSIS
    #include <sys/wait.h>

    pid_t
    wait(int *stat_loc);

    pid_t
    wait3(int *stat_loc, int options, struct rusage *rusage);

    pid_t
    wait4(pid_t pid, int *stat_loc, int options, struct rusage *rusage);

    pid_t
    waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION
    The wait() function suspends execution of its calling process until stat_loc information is available for a terminated child process, or a signal is received. On return from a successful wait() call, the stat_loc area contains termination information about the process that exited as defined below.

    The wait4() call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accumulated by child processes, or that require options. The other wait functions are implemented using wait4().

    The pid parameter specifies the set of child processes for which to wait. If pid is -1, the call waits for any child process. If pid is 0, the call waits for any child process in the process group of the caller. If
```

The **wait()** System Call

> man 2 fork

Wait? Why does man fork not work?



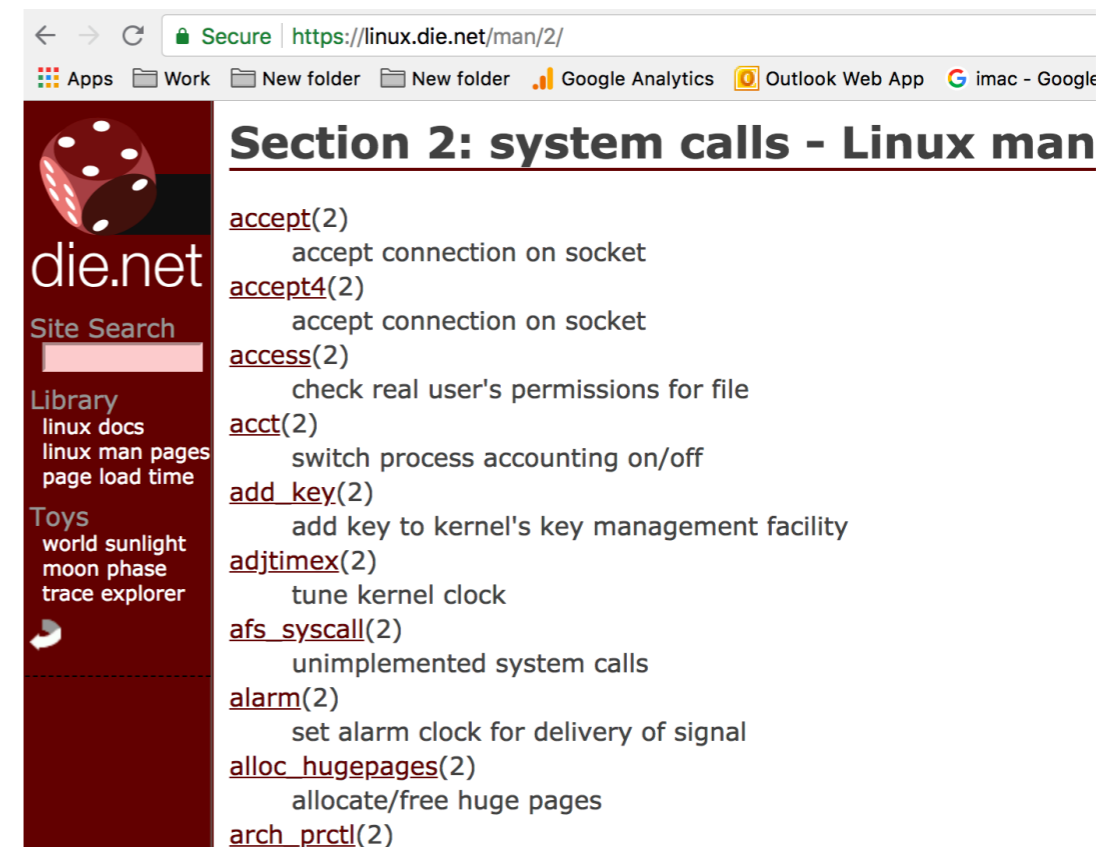
← → ↻ Secure | <https://linux.die.net/man/1/>

Apps Work New folder New folder Google Analytics Outlook Web App imac - Google Sear...

Section 1: user commands - Linux man

- [0alias\(1\)](#)
create quick scripts to run 0launch
- [0desktop\(1\)](#)
add programs to desktop environment
- [0install\(1\)](#)
decentralised software installation system
- [0launch\(1\)](#)
download/run programs by URL
- [0store\(1\)](#)
manage implementation cache
- [0store-secure-add\(1\)](#)
add implementation to system cache
- [3ddesk\(1\)](#)
activates 3D-Desktop, 3D desktop switcher
- [3ddeskd\(1\)](#)
starts daemon for 3D-Desktop, 3D desktop switcher

die.net
Site Search
Library
linux docs
linux man pages
page load time
Toys
world sunlight
moon phase
trace explorer



← → ↻ Secure | <https://linux.die.net/man/2/>

Apps Work New folder New folder Google Analytics Outlook Web App imac - Google Sear...

Section 2: system calls - Linux man

- [accept\(2\)](#)
accept connection on socket
- [accept4\(2\)](#)
accept connection on socket
- [access\(2\)](#)
check real user's permissions for file
- [acct\(2\)](#)
switch process accounting on/off
- [add_key\(2\)](#)
add key to kernel's key management facility
- [adjtimex\(2\)](#)
tune kernel clock
- [afs_syscall\(2\)](#)
unimplemented system calls
- [alarm\(2\)](#)
set alarm clock for delivery of signal
- [alloc_hugepages\(2\)](#)
allocate/free huge pages
- [arch_prctl\(2\)](#)

die.net
Site Search
Library
linux docs
linux man pages
page load time
Toys
world sunlight
moon phase
trace explorer

The **wait()** System Call

▲ The difference is that the `wait()` in `<sys/wait.h>` is the one you should use.

2

From the `wait(3)` man page:

▼ SYNOPSIS

`#include <sys/types.h>`
`#include <sys/wait.h>`

`pid_t wait(int *status);`

✓ The `wait` function isn't defined by the ISO C standard, so a conforming C implementation isn't allowed to declare it in `<stdlib.h>` (because it's legal for a program to use the name `wait` for its own purposes). gcc with glibc apparently does so in its default non-conforming mode, but if you invoke it with `gcc -ansi -pedantic` or `gcc -std=c99 -pedantic`, it doesn't recognize the function name `wait` or the type `pid_t`.

The **wait()** System Call

1. wait_demo_1.c :wait for child to exit



The difference is that the `wait()` in `<sys/wait.h>` is the one you should use.

2

From the `wait(3)` man page:



SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
```



```
pid_t wait(int *status);
```

The `wait` function isn't defined by the ISO C standard, so a conforming C implementation isn't allowed to declare it in `<stdlib.h>` (because it's legal for a program to use the name `wait` for its own purposes). gcc with glibc apparently does so in its default non-conforming mode, but if you invoke it with `gcc -ansi -pedantic` or `gcc -std=c99 -pedantic`, it doesn't recognize the function name `wait` or the type `pid_t`.

The **wait()** System Call

1. wait_demo_1.c :wait for child to exit
2. But wait, which wait to use?



The difference is that the `wait()` in `<sys/wait.h>` is the one you should use.

2

From the `wait(3)` man page:



SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
```



```
pid_t wait(int *status);
```

The `wait` function isn't defined by the ISO C standard, so a conforming C implementation isn't allowed to declare it in `<stdlib.h>` (because it's legal for a program to use the name `wait` for its own purposes). gcc with glibc apparently does so in its default non-conforming mode, but if you invoke it with `gcc -ansi -pedantic` or `gcc -std=c99 -pedantic`, it doesn't recognize the function name `wait` or the type `pid_t`.

The **wait()** System Call

The **wait()** System Call

1. `wait_demo_1.c` :wait for child to exit

The **wait()** System Call

1. `wait_demo_1.c` :wait for child to exit
2. But wait, which wait to use?

The **wait()** System Call

1. wait_demo_1.c :wait for child to exit
2. But wait, which wait to use?
3. Run wait_demo_2.c

The **wait()** System Call

1. wait_demo_1.c :wait for child to exit
2. But wait, which wait to use?
3. Run wait_demo_2.c
4. Run wait_demo_3.c and find out NULL/0,'\0'

The **exec()** System Call

> man 3 exec

```
execvp(const char *file, char *const argv[]);
```

```
int  
execvP(const char *file, const char *search_path, char *const argv[]);
```

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function **execve(2)**. (See the manual page for **execve(2)** for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The const char *arg0 and subsequent ellipses in the **execl()**, **execvp()**, and **execle()** functions can be thought of as arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a NULL pointer.

The **execv()**, **execvp()**, and **execvP()** functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **exec()** System Call

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments
3. `exec_demo_3.c`: Pass arguments around!

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments
3. `exec_demo_3.c`: Pass arguments around!
4. `exec_demo_4.c`: Get wc of `exec_demo_4.c`

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments
3. `exec_demo_3.c`: Pass arguments around!
4. `exec_demo_4.c`: Get wc of `exec_demo_4.c`
5. `exec_demo_5.c`: Get wc of any file

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments
3. `exec_demo_3.c`: Pass arguments around!
4. `exec_demo_4.c`: Get wc of `exec_demo_4.c`
5. `exec_demo_5.c`: Get wc of any file
6. `man wc` to understand what we get

The **exec()** System Call

1. `exec_demo_1.c` : Execute other process
2. `exec_demo_2.c` : Execute other process with arguments
3. `exec_demo_3.c`: Pass arguments around!
4. `exec_demo_4.c`: Get wc of `exec_demo_4.c`
5. `exec_demo_5.c`: Get wc of any file
6. `man wc` to understand what we get
7. `exec_demo_6.c`: Local variables accessible in child!