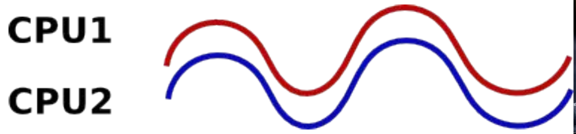
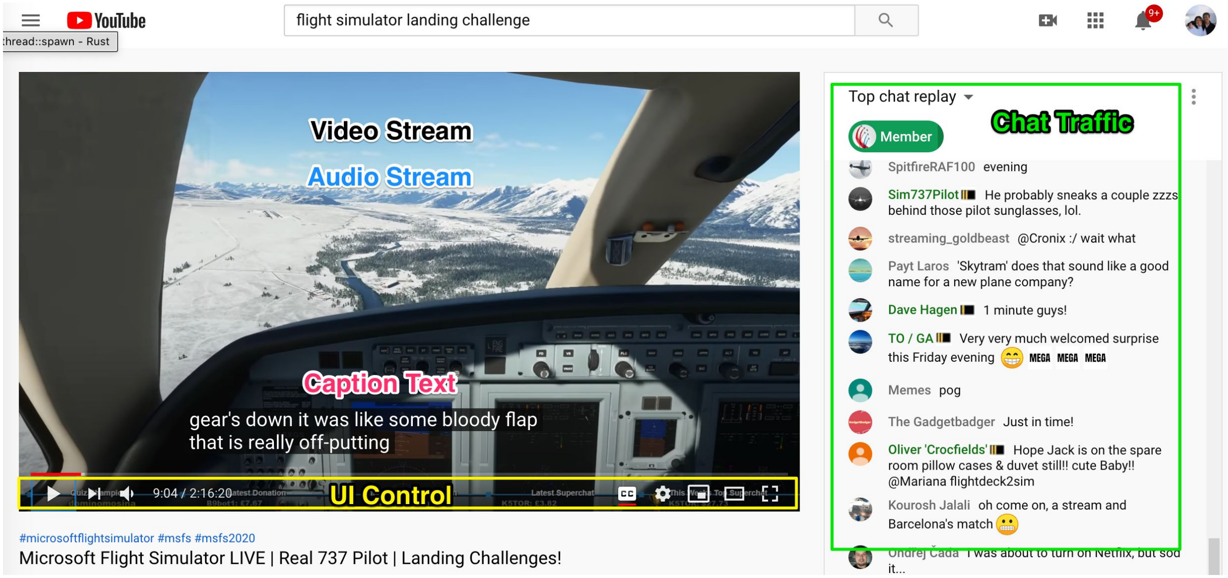
# Threads



---

# Why (Multi) Threads?



gear's down it was like some bloody flap
that is really off-putting

#microsoftflightsimulator #msfs #msfs2020
Microsoft Flight Simulator LIVE | Real 737 Pilot | Landing Challenges!

SpitfireRAF100 evening
Sim737Pilot He probably sneaks a couple zzzs behind those pilot sunglasses, lol.
streaming_goldbeast @Cronix :/ wait what
Payt Laros 'Skytram' does that sound like a good name for a new plane company?
Dave Hagen 1 minute guys!
TO / GA Very very much welcomed surprise this Friday evening 😁 MEGA MEGA MEGA
Memes pog
The Gadgetbadger Just in time!
Oliver 'Crocfields' Hope Jack is on the spare room pillow cases & duvet still!! cute Baby!! @Mariana flightdeck2sim
Kourosh Jalali oh come on, a stream and Barcelona's match 😬
Ondřej Čada I was about to turn on Netflix, but sod it...

absolute butter unbelievable

# Multithreading **Real-World** Examples

- YouTube player
    - UI control thread
    - Audio playback thread
    - Image frames playback threads
    - Network data fetcher thread
    - Caption thread
    - *anything else?*
- Smart IDEs
    - Text editor thread
    - Indexer (for text auto complete)
    - Linter thread
    - Compiler thread
    - Unit tester thread
    - Language Server Protocol in VSCode

# Single Threaded Processes
# vs.
# Multi-Threaded Processes

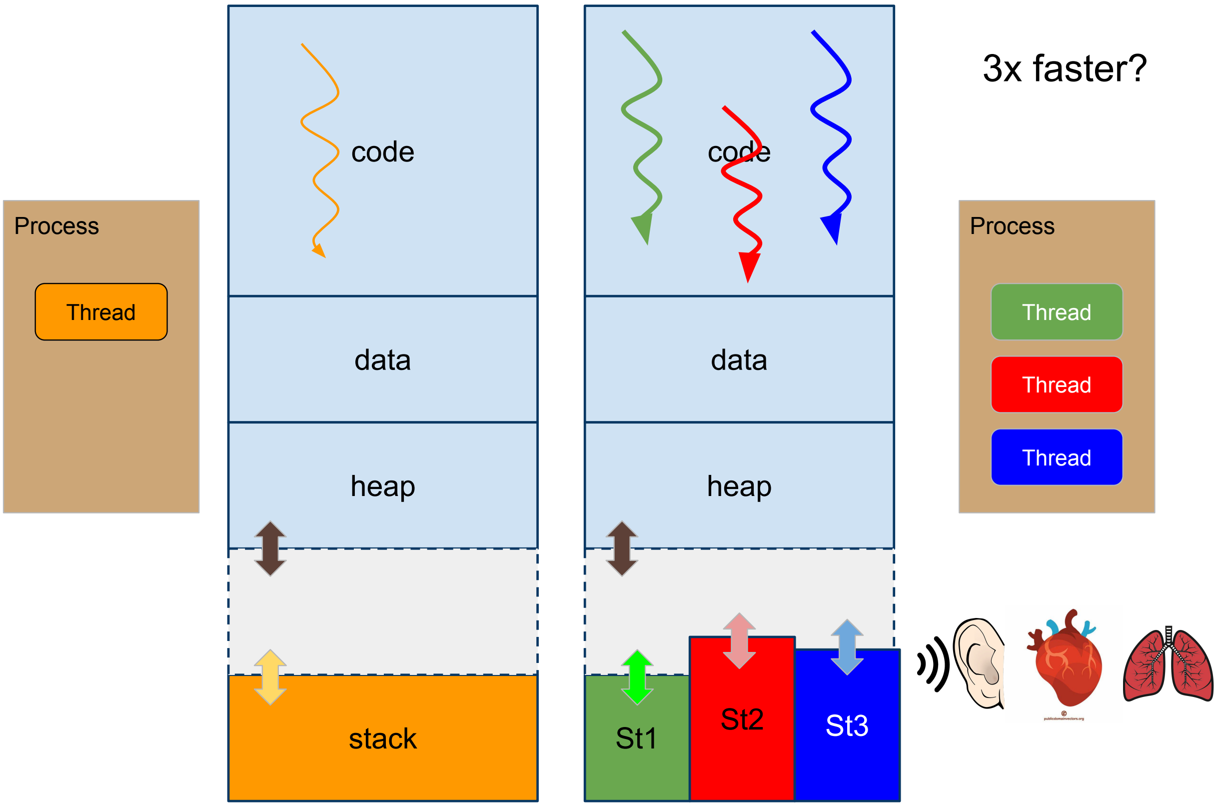
# Parent-Child: Separate Flow of Execution

```
/* parent */
int main() {
  pid_t who = fork();
  if (who == 0) {
     /* Child work begins here */

     // more code not shown

     exit (0xBEEF);
  }
  else {
     /* Parent work begins here */

     // more code not shown

     int status;
     who = wait (&status);
  }
  return 0;
}
```

```
/* child */
int main() {
  pid_t who = fork();
  if (who == 0) {
     /* Child work begins here */

     // more code not shown

     exit (0xBEEF);
  }
  else {
     /* Parent work begins here */

     // more code not shown

     int status;
     who = wait (&status);
  }
  return 0;
```

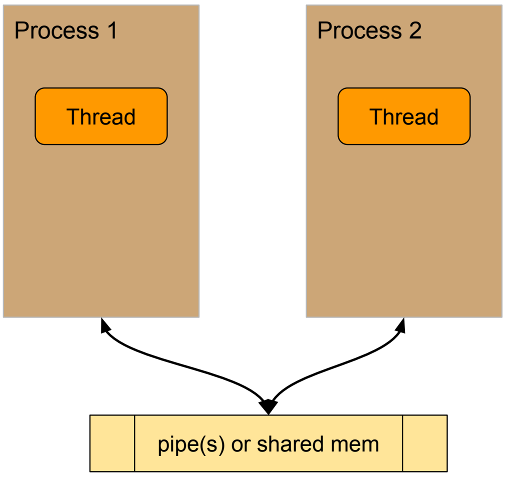
Two **Single-Threaded** Processes

## Single-Threaded vs. Multi-Threaded Process



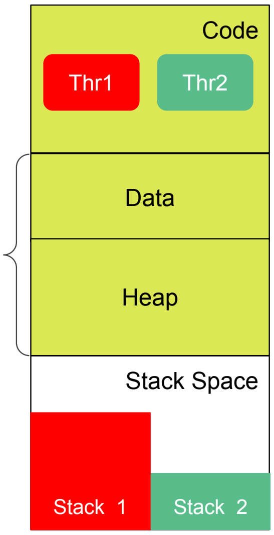
---

## Inter (Process|Thread) Communication



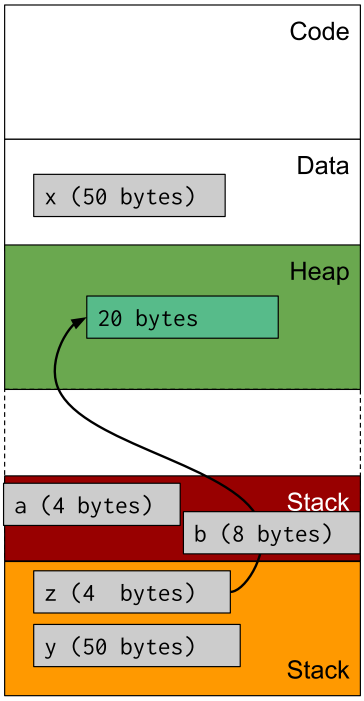
*Processes exchange data via* ***external*** *structure*

*Exchanging data via* ***internal*** *structure (heap and data sections)*



# (Data|Heap|Stack) Sections

```
#include <stdio.h>

char x[50]; // global var

void my_func(int a) {
  double b;
}

int main() {
  char y[50];
  char z*;

  z = malloc(20);
  // my_func();
}
```



---

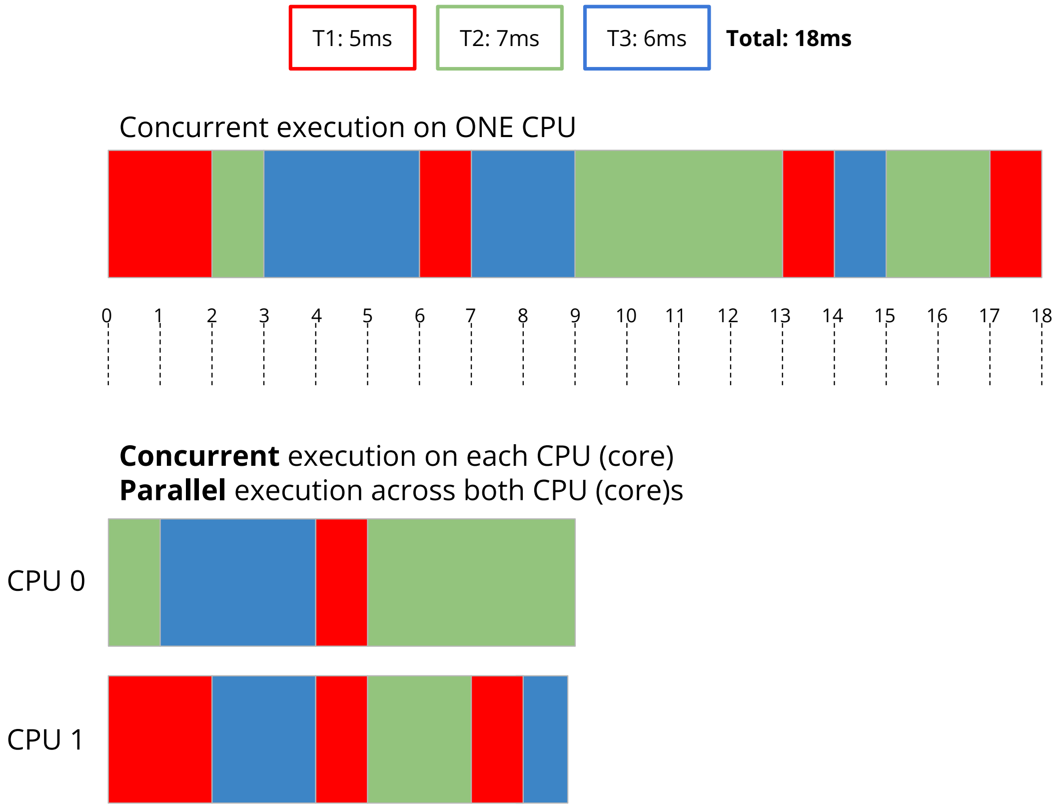
## Parallel ⇒ Concurrent
## Concurrent ⇏ Parallel

*Parallelism* requires *concurrency*

but

*Concurrency* does not guarantee *parallelism*

---



T1: 5ms

T2: 7ms

T3: 6ms

**Total: 18ms**

Concurrent execution on ONE CPU

**Concurrent** execution on each CPU (core)
**Parallel** execution across both CPU (core)s

# Concurrency vs. Parallelism

- Concurrent systems
  - Multiple tasks taking turn to use (**one**) CPU to make progress together
- Parallel systems
  - Multiple tasks running simultaneously on **multiple** CPUs (cores)
  - A program typically consists of **serial** tasks (tasks that can only run on **one CPU**) and **parallel** tasks (tasks that are independent of each other and can run in parallel on **multiple CPUs**)

---

# Concurrency

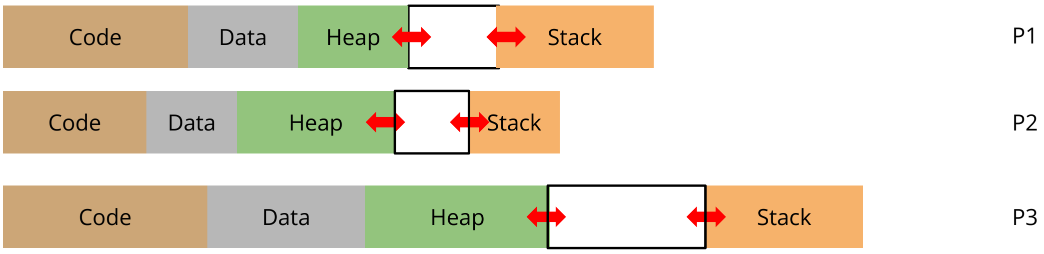
Concurrency with **multiple processes**

- Each process is single-threaded
- Overhead of running multiple-processes (mainly space/memory overhead)
- Requires IPC channels (socket, pipe, signals, files, …) to communicate
- Due to OS protection policy
  - it requires more work to allow these processes share data
  - It is **easier** to write *safe* concurrent code
- Processes can be distributed across multiple distinct machines

Concurrency with **multiple threads**

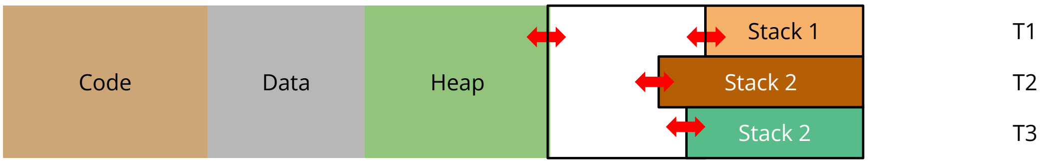
- Several flows of execution sharing the *same* process image (same code, same data, same heap, but ***different stacks***)
- Does not require communication channels for exchanging data
- Concurrent code may be unsafe (**race conditions**)

# Concurrent Processes



3 threads across 3 (single-threaded) processes

# Concurrent Threads



3 threads within one process

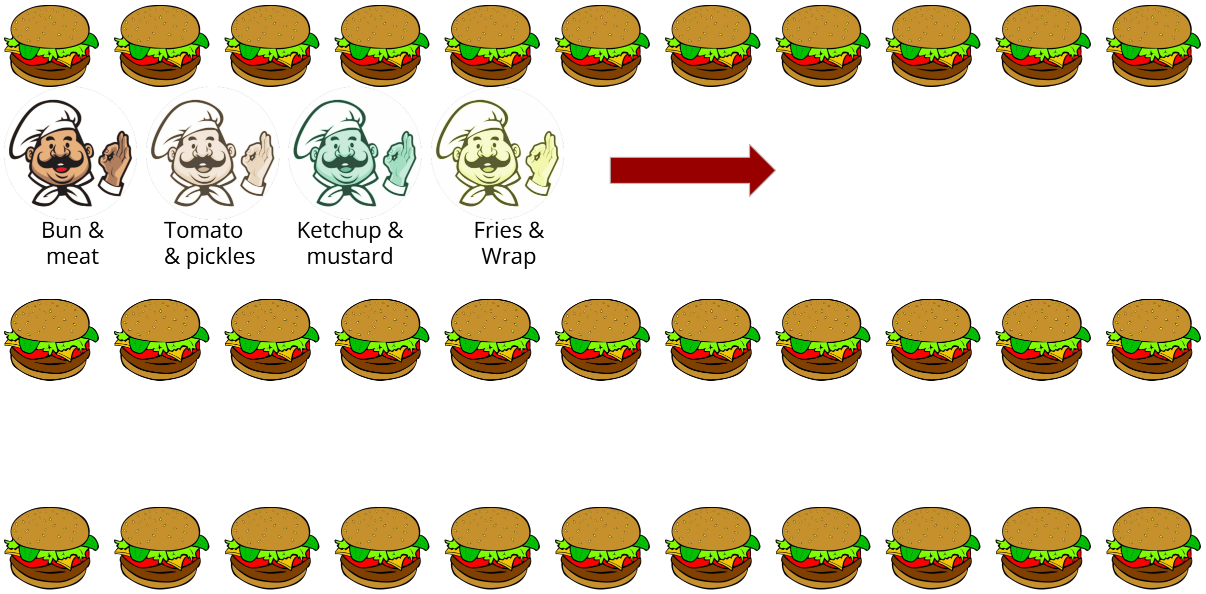
---

# Design for Parallelism

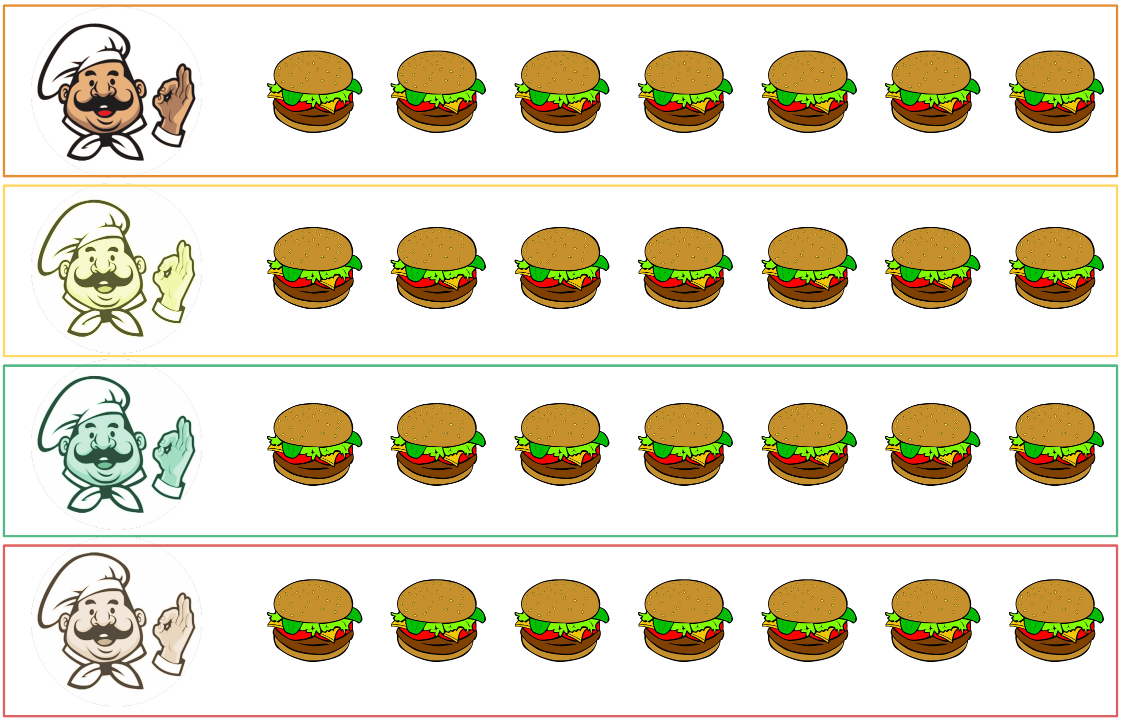
# Time for lunch?

---

# Task Parallelism



# Data Parallelism



---

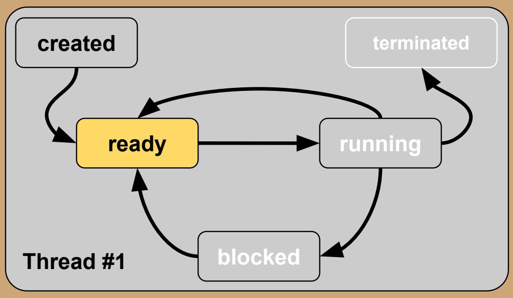
# Potential Conflicts?
# Data Parallelism vs Task Parallelism

# Design for Parallelism

- Task Parallelism (Separation of Concerns)
  - Several “independent” modules which run in parallel on separate CPUs
  - Each module runs a different program/set of instructions
  - **Examples**: *music streaming* (one thread reads the song bytes from the net, one thread plays the music on the audio device, one thread responds to UI actions)
- Data Parallelism (Increased Performance)
  - To handle massive amount of data, smaller subsets of data deployed to one CPU
  - Each CPU runs the same program / set of instructions
  - **Examples**:
    - *mergesort* (each CPU runs the same algorithm but on a smaller set of data),
    - *graphics shaders* (each fragment processor runs the same function to determine the final shade of one pixel)

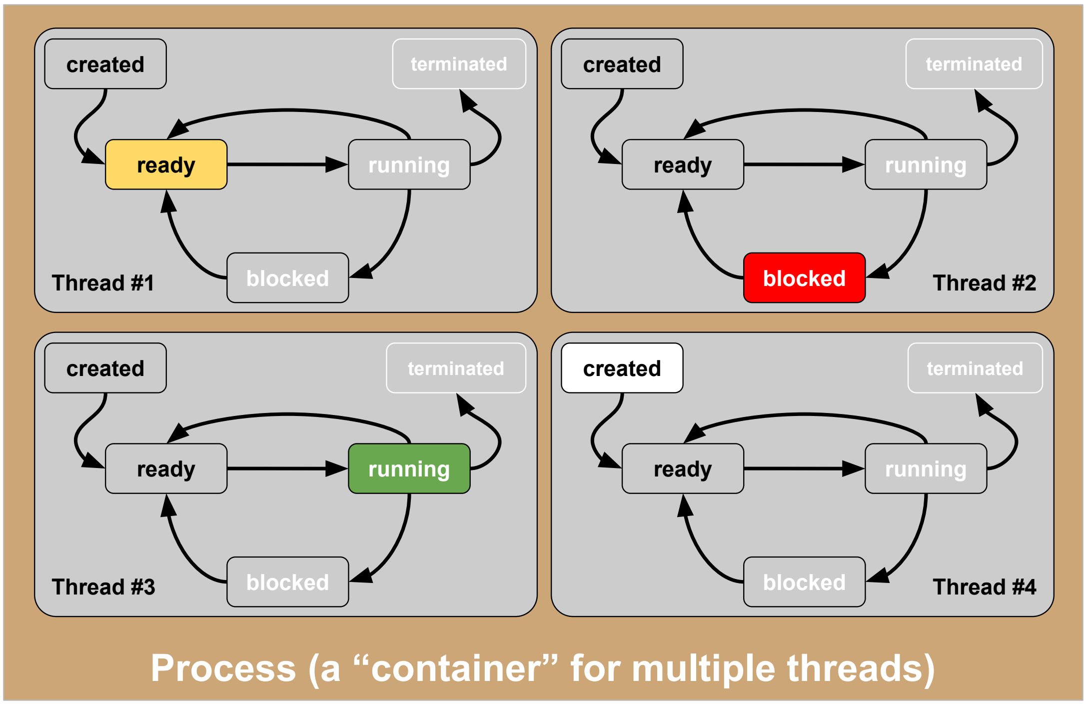
---

# Process state vs. Thread State?



**SIngle-Threaded Process**

# Process state vs. Thread State?



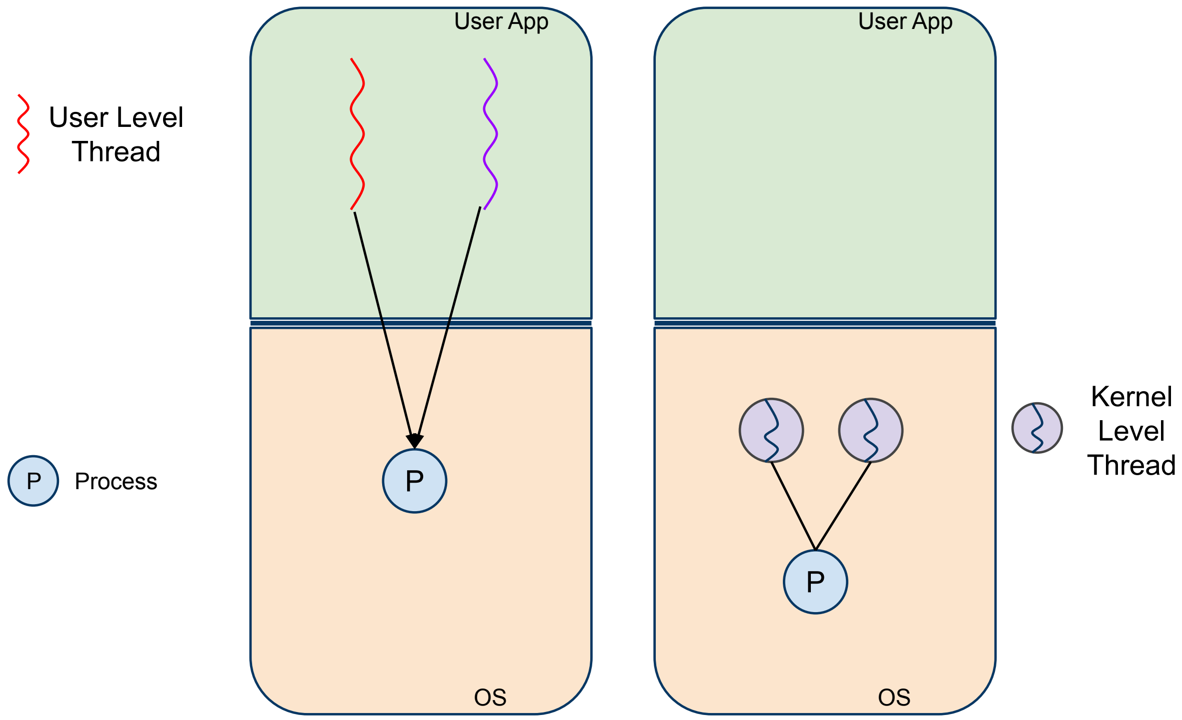
Process (a “container” for multiple threads)

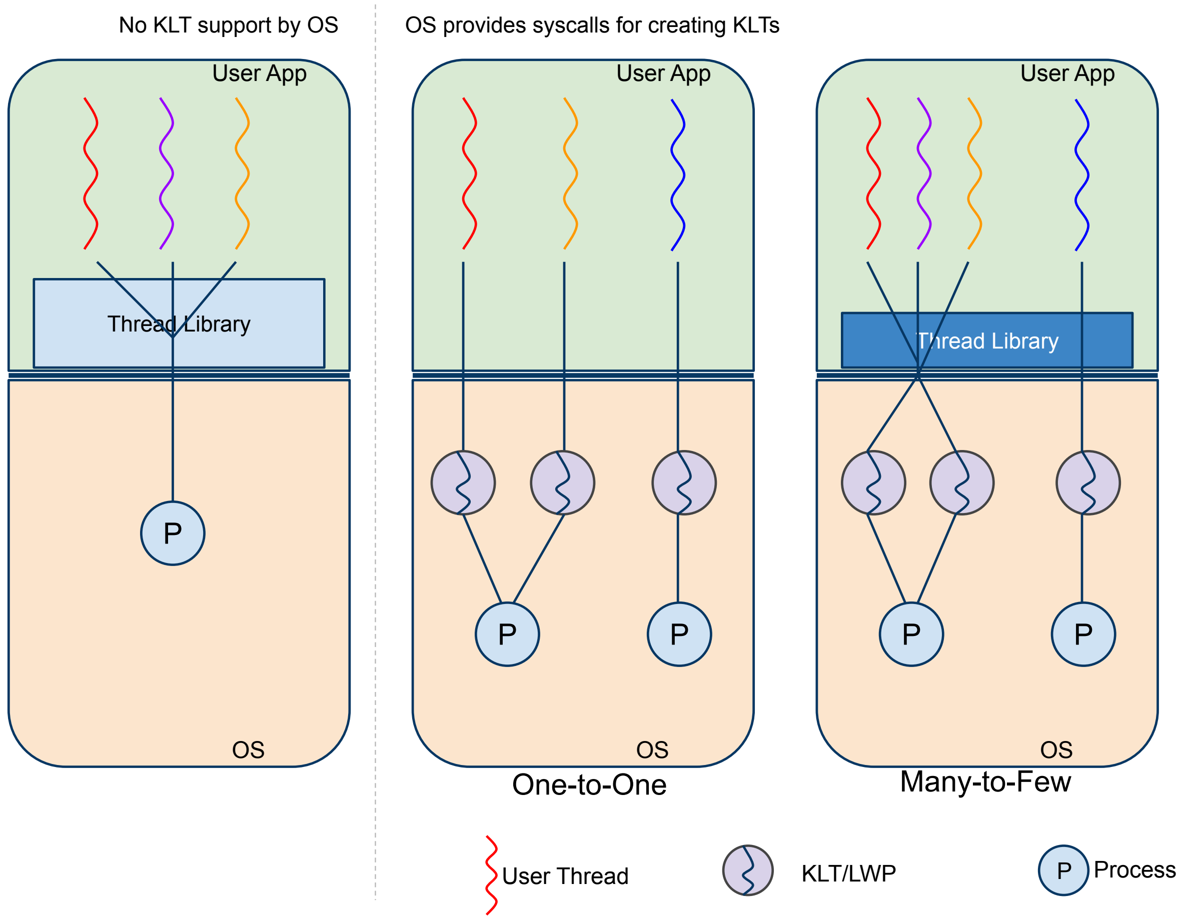
---

# Thread Implementation

- OS has *native support* to manage threads
  - Threads scheduling by the OS
  - OS provides system calls to create/destroy threads
  - User-Level (ULT) and Kernel-Level Threads (KLT)
- OS **does not** support threads natively
  - Threads are created and managed by a library
  - Thread scheduling by the library
  - OS can schedules only processes
  - User-Level Threads (ULT) only

# ULT vs KLT


User Level Thread
Process
User App
P
OS
User App
P
OS
Kernel Level Thread


No KLT support by OS
OS provides syscalls for creating KLTs
User App
Thread Library
P
OS
User App
P
P
OS
One-to-One
User App
Thread Library
P
P
OS
Many-to-Few
User Thread
KLT/LWP
P
Process

# ULT Demo: thread-manager.c
# swap_context()    [EOS]

---

# ULT to KLT Mapping

- Many ULTs ⇒ One Process (when OS does not support KLTs)
    - Thread management by thread library in user space
    - Multiple user threads cannot run in parallel
- Many ULTs ⇒ One KLT
    - Thread management by thread library in user space
    - Multiple user threads cannot run in parallel
- One ULTs ⇒ One KLT
    - Multiple user threads can run in parallel, each thread is scheduled directly by OS
- Many ULTs ⇒ Few KLTs
    - Many ser threads are multiplexed to smaller or equal number of kernel threads
    - Can be used by the system puts a limit on max KLTs users can create
    - Multiplexed ULTs vs bound ULTs

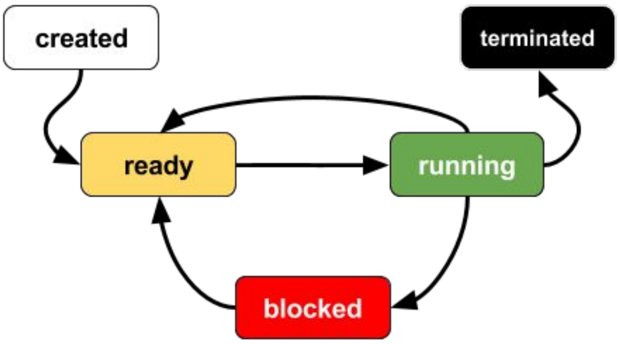
# Misconceptions

- UL threads are faster to run
- UL threads run (only) in user-mode

- KL threads do not have to be associated with a process
- KL threads run (only) in kernel-mode
- KL threads are needed to execute system calls

---

# Thread state vs Process state (diagram)



created ↠ ready: the process just created, ready to use the CPU
ready ↠ running: the process is dispatched by the OS to use the CPU
running ↠ ready: the process time slice is expired
ready ↠ blocking: the process made a blocking system call (read(), sleep, ….)
blocked ↠ ready: the blocking system call completed, the process is ready to use the CPU again
running ↠ terminated: the process exit normally (or with error)

# Thread Implementations

- POSIX Threads (either user space or kernel space)
    - C
    - C++
- Windows (kernel lib)
- Java Threads (running on JVM)
    - JVM on Linux depends on POSIX Threads
    - JVM on Windows depends on Windows Kernel Lib

---

# POSIX Threads

# POSIX Thread vs. Process APIs

| POSIX Threads | Description | Process Equivalent |
|---|---|---|
| `pthread_create()` | Create a new thread | `fork()` |
| `pthread_self()` | Return the thread ID of the caller | `getpid()` |
| `pthread_cancel()` | Send a request to cancel a thread. | `???` |
| `pthread_detach()` | Detach a thread (make it unjoinable) | *"orphan"* |
| `pthread_exit()` | Terminate the calling thread | `exit()` |
| `pthread_kill()` | Deliver a signal to a thread | `kill()` |
| `pthread_join()` | Join with a terminated thread | `wait()` |

*When the parent process dies, the "orphan" will also die*

---

# fork() vs pthread_create()

- fork(): **both** parent and child processes resume at the **next statement** following fork() call
- pthread_create():
  - Parent thread resumes at the next statement
  - Child thread resumes at a function

# Examples

- Three examples on GitHub gist
- Java (`happy.java`)
    - implements Runnable
    - extends Thread
- C (`happy-pthr.c`)
    - pthread library
- C++11 (`happy.cpp`)
    - `#include <thread>`
    - `#include <future>`
    - `std::async`
    - `std::future`

# Posix Threads: Basic Example

C

```c
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
  printf ("Hello C Thread\n");
  return NULL;
}

int main() {
  printf ("From main thread\n");
  pthread_t one;
  pthread_create(&one, NULL, hello, NULL);
  pthread_join(one, NULL);
  printf ("About to exit\n");
  return 0;
}
```

C++11

```cpp
#include <thread>
using namespace std;

void hello() {
   cout << "Hello C++ thread\n";
}

int main() {
   cout << "From main thread\n";

   thread one(hello);

   one.join();
   cout << "About to exit\n";
   return 0;
}
```

# Posix Threads: Passing Argument(s)

C

```c
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
  int *num = (int *) arg;
  printf ("Hello C Thread %d\n", *num);
  return NULL;
}

int main() {
  printf ("From main thread\n");
  pthread_t one;
  int val = 53;
  pthread_create(&one, NULL, hello, &val);
  pthread_join(one, NULL);
  printf ("About to exit\n");
  return 0;
}
```

C++11

```cpp
#include <thread>
using namespace std;

void hello(int num) {
   cout << "Hello C++ thread" << num;
}

int main() {
   cout << "From main thread\n";

   thread one(hello, 53);

   one.join();
   cout << "About to exit\n";
   return 0;
}
```

---

# Posix Threads: Return Result

C

```c
#include <pthread.h>
#include <stdio.h>

void* hello(void* arg) {
  printf ("Hello C Thread\n");
  return (void *) 71;
}

int main() {
  printf ("From main thread\n");
  pthread_t one;
  pthread_create(&one, NULL, hello, NULL);
  int val;
  pthread_join(one, (void *) &val);
  printf ("About to exit %d\n", val);
  return 0;
}
```

C++11

```cpp
#include <future>
#include <iostream>

using namespace std;

int hello() {
  cout << "Hello C++ thread\n";
  return 71;
}

int main() {
  cout << "From main thread\n";
  auto one = async (hello);
  cout << "About to exit\n";
  cout << one.get();
  return 0;
}
```

async/future = *run asynchronously now, get the result later (a time in the future)*

# Multi-Process vs. Multi-Thread

- In an multi-process application, the processes are **isolated** from each other. Data manipulation errors in one process won't affect the other processes
- In an MT application, the threads share the same data. Data manipulation errors by one thread can **easily spread** to the others

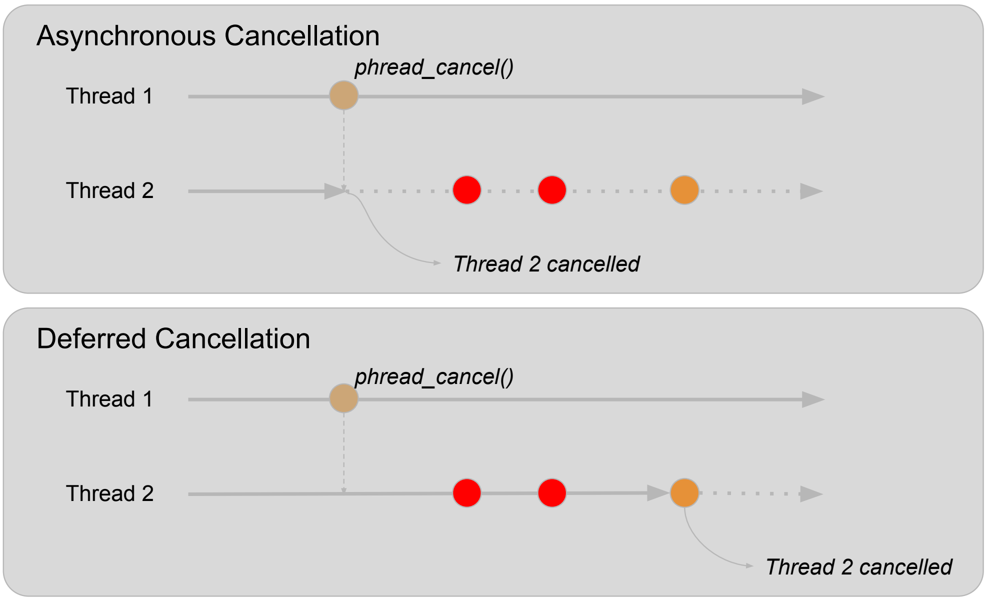
- Potential bugs in MT-app
  - Sharing local variables created in a thread with other threads
  - Deallocating a resource by one thread while the other threads are using it
  - Race conditions
  - Debugging is hard (opportunity for you to make a MT debugger)

---

# `pthread_cancel()`

- Thread cancelability **state**: enabled (default) or disabled
- Thread cancelability **type**: deferred (default) or asynchronous
  - A thread with async cancelability can be cancelled anytime!
- A **deferred cancelation**: postpone termination until the thread reaches a "cancellable library call / system call".
  - Refer to `man 7 pthreads` for the list of cancellation points

# Thread Cancellation



Asynchronous Cancellation

Thread 1

phread_cancel()

Thread 2

Thread 2 cancelled

Deferred Cancellation

Thread 1

phread_cancel()

Thread 2

Thread 2 cancelled

● *Non-cancellable lib/syscalls*

● *Cancellable lib/syscalls*

---

# `man 7 pthreads`

- Thread-safe functions: functions that can be safely called from a multi-threaded program
- List of thread cancellable functions (a.k.a cancellable points)

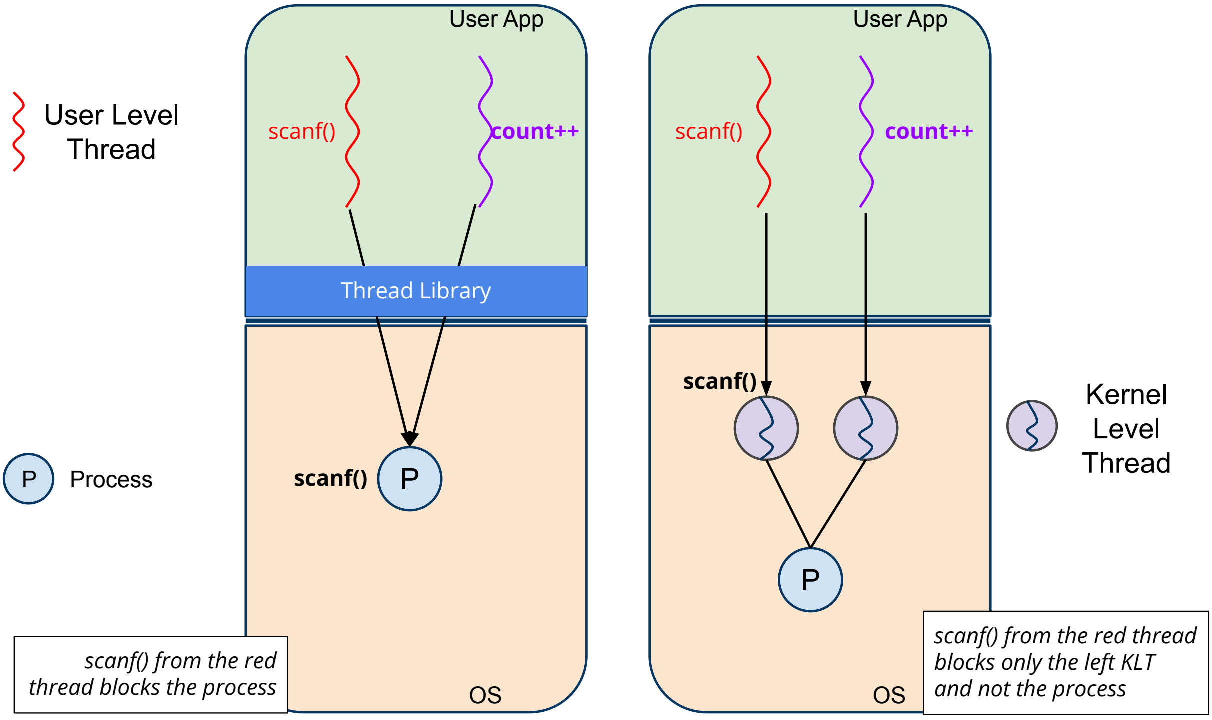
# System Calls in MT Thread Processes

Invoking the following system calls in a multi-threaded process may affect *other threads* within the same process:

- *Blocking system calls*
- fork()
- exec*()
- signal() and kill()

---

# Blocking SysCalls from User App



*scanf() from the red thread blocks the process*

*scanf() from the red thread blocks only the left KLT and not the process*

# Design Issues: Blocking System Calls

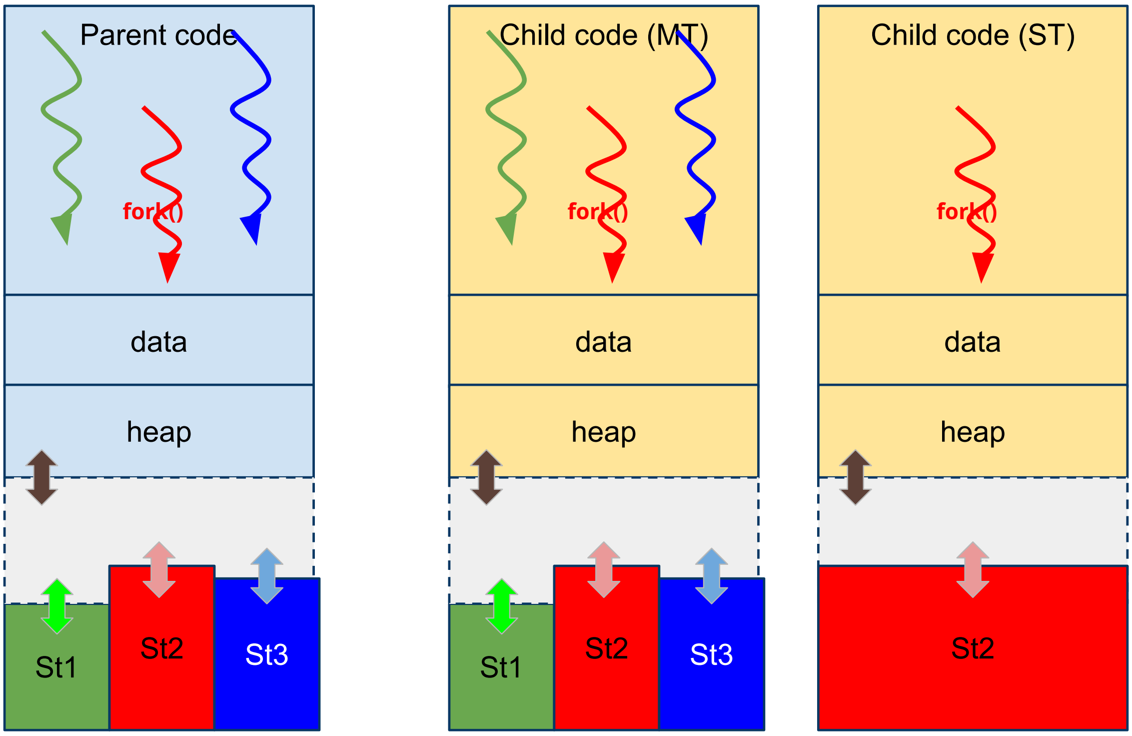
- In a ULT only implementation, blocking calls issued by a single thread will place the entire process into a blocked state
  - *Solution*: replace blocking system calls with non-blocking thread library service calls, so the thread library can *postpone* the actual system call until the "time is right"
- Kernel-Thread implementation does not suffer from this issue

---

# System Call: fork()

- What to duplicate when fork() is issued by a thread?
  - Do we duplicate all the threads?
  - Do we duplicate just the thread that calls fork()?
- Linux fork() creates a single-threaded child process

# fork(): is the child ST or MT?

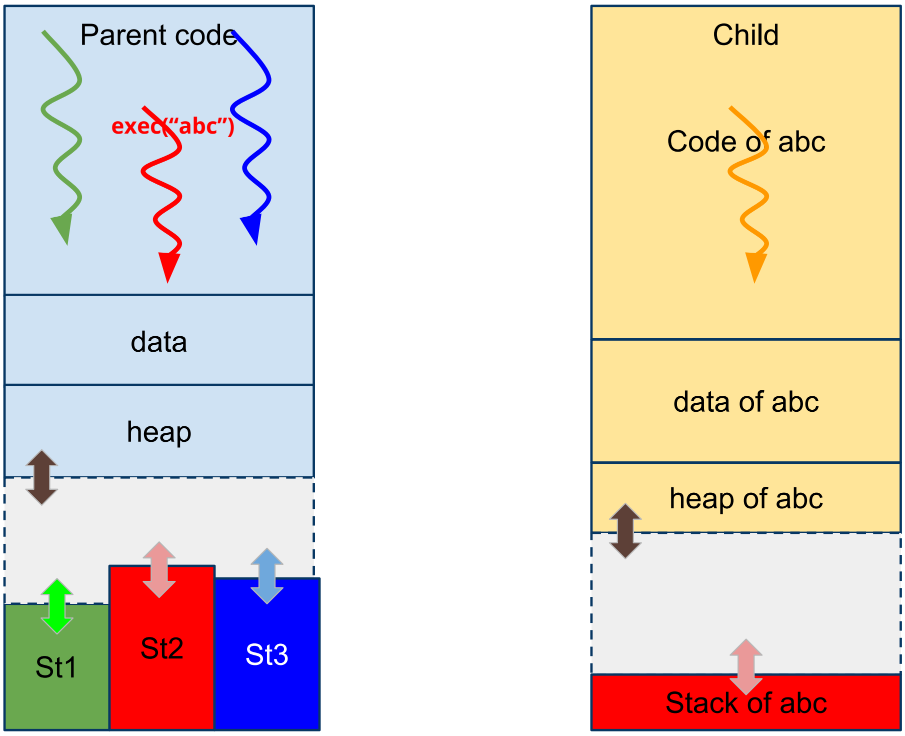


---

# System Call: exec()

- `exec()` on MT process behaves similarly to ST process
- The entire process image is replaced
  - All the [other] threads in the process will disappear
  - After a successful `exec()` the new process is **always** single-threaded (*until that process creates more threads*)

# exec(): the child is always ST



---

# System Call: signal()/sigaction()/kill()

- Which thread(s) should receive the signal?
  - Deliver to the thread to which the signal applies?
    - example: `SIGSEGV`, `SIGFPE`
  - Deliver to all the threads in the process?
    - example: `SIGINT`
  - Deliver to a specific thread?
    - When signal is raised by `pthread_kill(____, _____);`

# Amdahl's Law

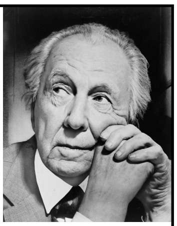
- A program of M instructions running on a CPU with speed of V insts/sec
  - Time to complete when the entire program runs on ONE CPU: M/V second
- Assume s is the *fraction* of the **serial task** therefore p = (1- s) is the *fraction* of the **parallel task**.

When N CPUs are available:

- The serial portion (s) of the program runs on ONE CPU will complete in sM/V
- The parallel portion (1-s) runs on N cpus will complete in (1-s)M/(NV)

---

# Amdahl Speed Up



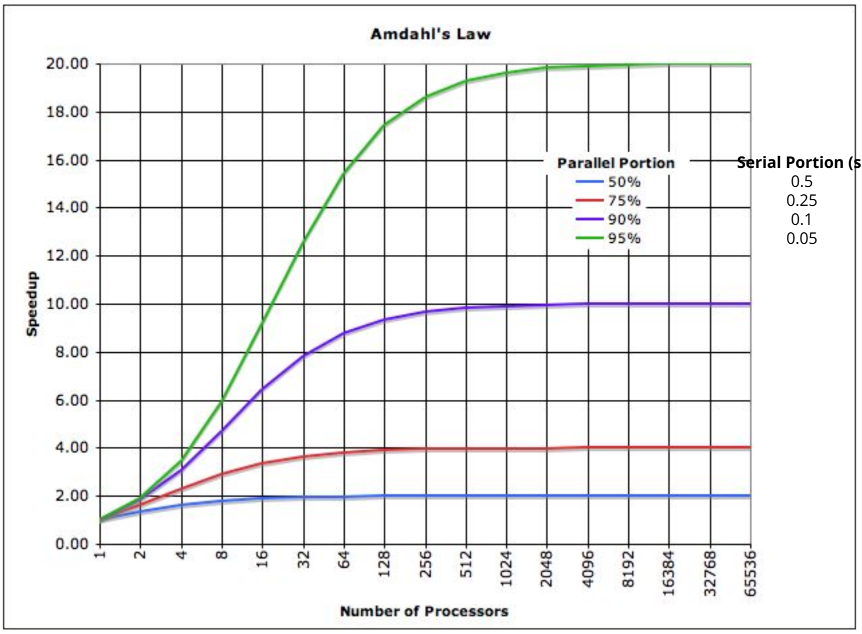
$$\text{Speedup} = \frac{\text{time on 1 CPU}}{\text{time on N CPUs}} = \frac{M/V}{s\ M/V+(1-s)\,\frac{M}{NV}}$$

$$\text{Speedup} = \frac{1}{s+\frac{(1-s)}{N}}$$

$$\text{Max Speedup} = \lim_{N\to\infty} \frac{1}{s+\frac{(1-s)}{N}} = \frac{1}{s}$$

# Amdahl Graph

**Live Graph on Desmos**



| Serial Portion (s) | Max Speedup (1/s) |
|---|---|
| 0.5 | 2x |
| 0.25 | 4x |
| 0.1 | 10x |
| 0.05 | 20x |

---

# Demo: Co-routines (JavaScript/Kotlin/Python/C)