CS 310

PROCESSES

LOYOLA
UNIVERSITY CHICAGO
AD · MAJOREM · DEI · GLORIAM
1870

ADMINISTRIVIA

- `shittyshell` In-class assignment this week.
- Shell homework out this week, due Sunday 9/10 11:59 PM.
- Microsoft Teams Join Code `34iz1ae`
- Join link on course website
- VM Download links?

TODAY WE ARE TALKING ABOUT PROCESSES

- Program: combination of instructions & data
- Process: a running program
- App: ?

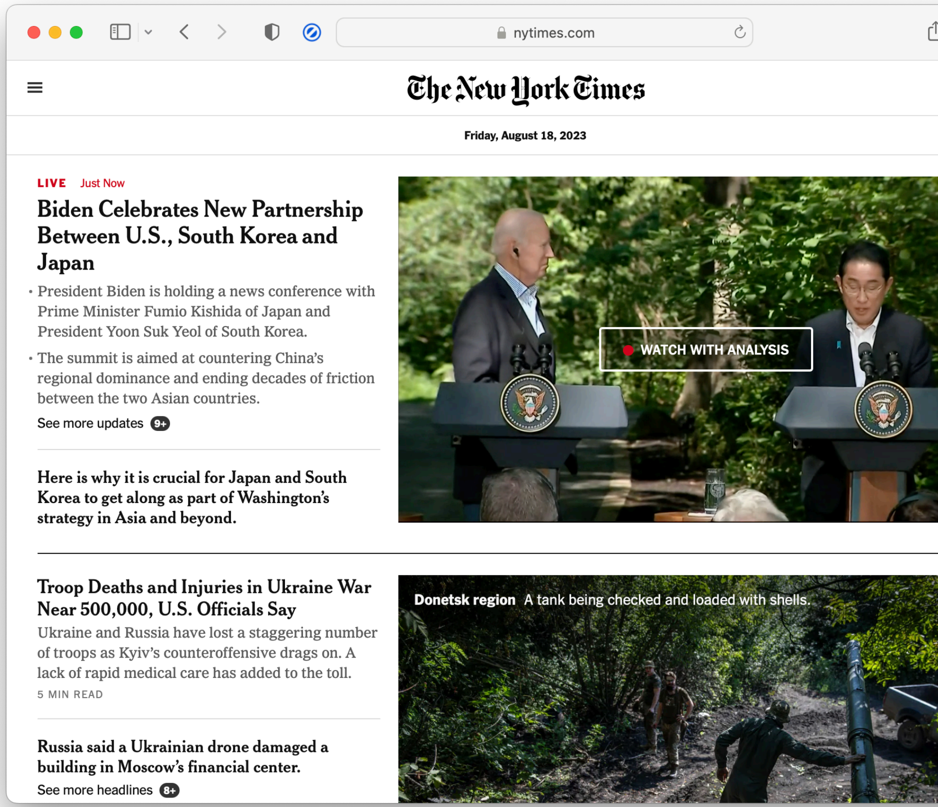
- OS Supports Programs & Processes, so what do you want your program to do?

WHAT DO YOU WANT YOUR PROGRAM TO DO?
EXAMPLE: VIDEO GAME

1. Get input from user 🕹️
2. Recompute location of objects
3. Draw to screen

WHAT DO YOU WANT YOUR PROGRAM TO DO? EXAMPLE: WEB BROWSER

1. Get input from user
2. Request page from server
3. Draw to screen



The New York Times

Friday, August 18, 2023

LIVE Just Now

Biden Celebrates New Partnership Between U.S., South Korea and Japan

- President Biden is holding a news conference with Prime Minister Fumio Kishida of Japan and President Yoon Suk Yeol of South Korea.
- The summit is aimed at countering China's regional dominance and ending decades of friction between the two Asian countries.

See more updates 9+

WATCH WITH ANALYSIS

Here is why it is crucial for Japan and South Korea to get along as part of Washington's strategy in Asia and beyond.

Troop Deaths and Injuries in Ukraine War Near 500,000, U.S. Officials Say

Ukraine and Russia have lost a staggering number of troops as Kyiv's counteroffensive drags on. A lack of rapid medical care has added to the toll.

5 MIN READ

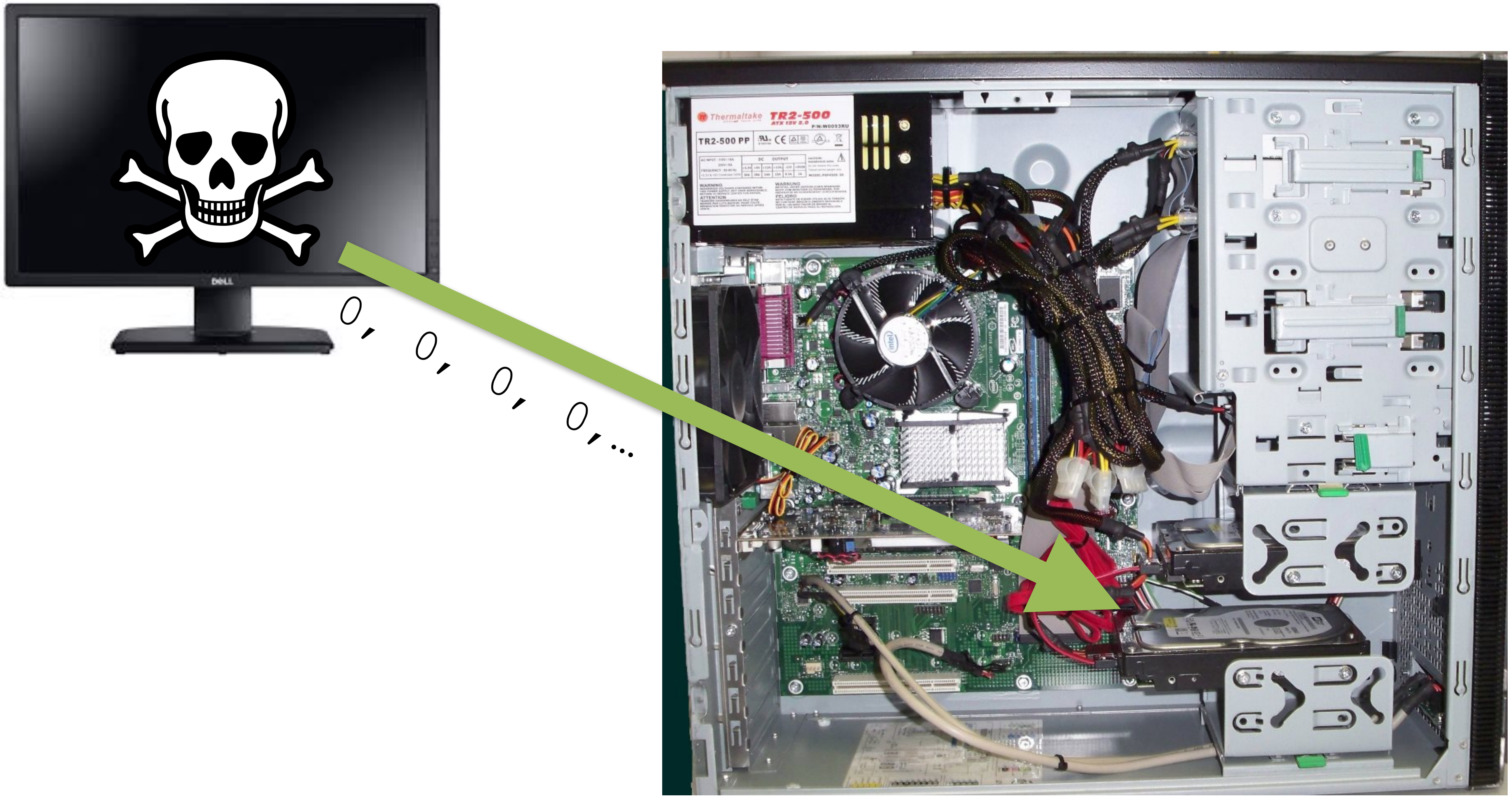
Donetsk region A tank being checked and loaded with shells.

Russia said a Ukrainian drone damaged a building in Moscow's financial center.

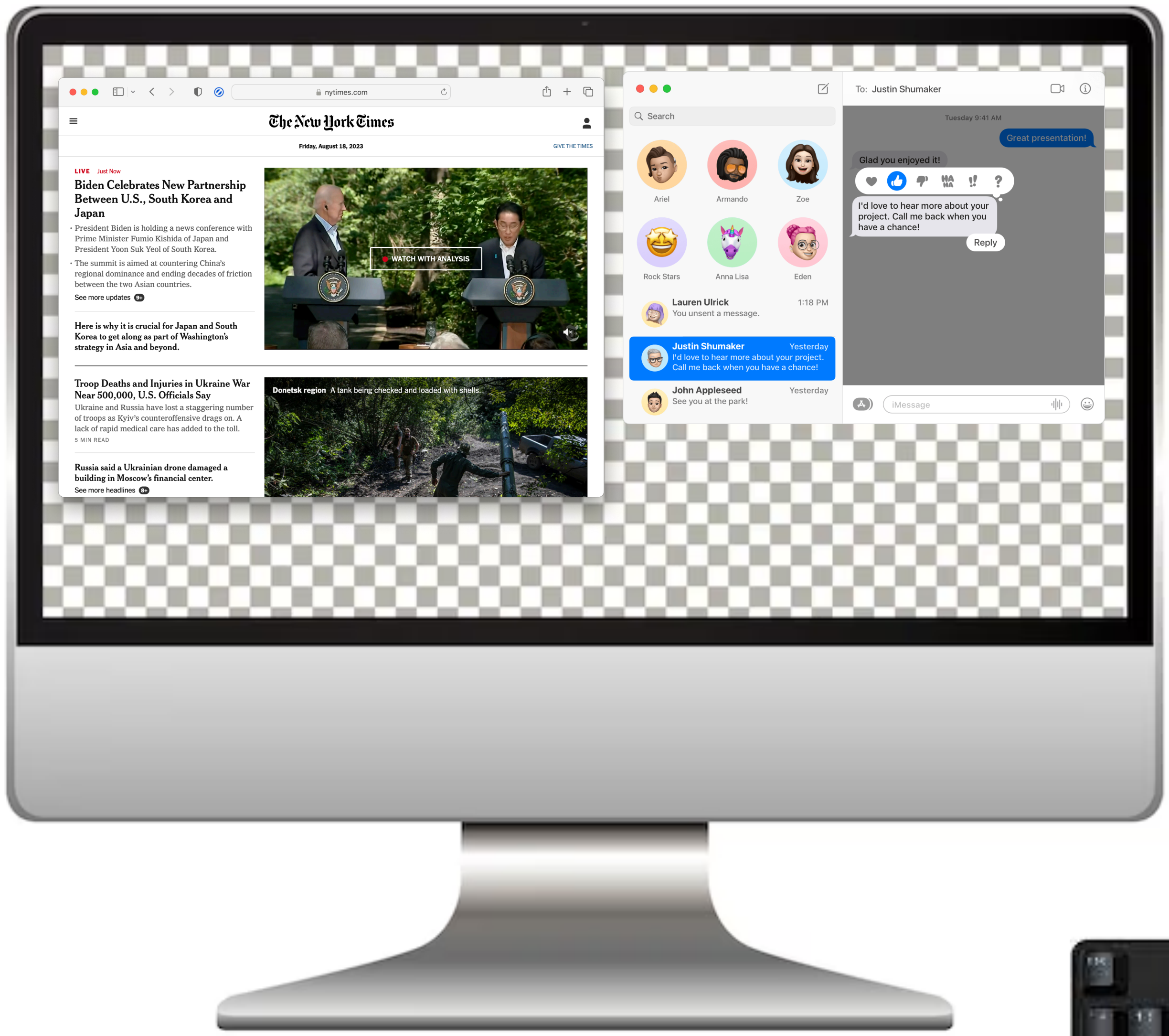
See more headlines 8+

STRATEGY: LET PROGRAMS DO WHATEVER THEY WANT.

- Draw directly to screen
- Read & write directly to network card
- Read & write directly to disk
- etc.


0, 0, 0, 0,…

WHAT IF TWO APPS WANT TO USE THE SAME RESOURCE?


nytimes.com

The New York Times

Friday, August 18, 2023

GIVE THE TIMES

LIVE Just Now

Biden Celebrates New Partnership Between U.S., South Korea and Japan

• President Biden is holding a news conference with Prime Minister Fumio Kishida of Japan and President Yoon Suk Yeol of South Korea.

• The summit is aimed at countering China's regional dominance and ending decades of friction between the two Asian countries.

See more updates

WATCH WITH ANALYSIS

Here is why it is crucial for Japan and South Korea to get along as part of Washington's strategy in Asia and beyond.

Troop Deaths and Injuries in Ukraine War Near 500,000, U.S. Officials Say

Ukraine and Russia have lost a staggering number of troops as Kyiv's counteroffensive drags on. A lack of rapid medical care has added to the toll.

5 MIN READ

Donetsk region A tank being checked and loaded with shells.

Russia said a Ukrainian drone damaged a building in Moscow's financial center.

See more headlines

Search

Ariel, Armando, Zoe, Rock Stars, Anna Lisa, Eden

Lauren Ulrick 1:18 PM
You unsent a message.

Justin Shumaker Yesterday
I'd love to hear more about your project. Call me back when you have a chance!

John Appleseed Yesterday
See you at the park!

To: Justin Shumaker

Tuesday 9:41 AM

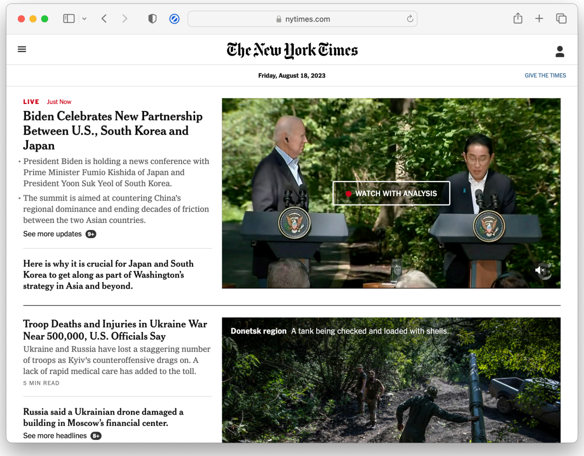
Great presentation!

Glad you enjoyed it!

I'd love to hear more about your project. Call me back when you have a chance!

Reply

iMessage



nytimes.com

The New York Times

Friday, August 18, 2023

GIVE THE TIMES

LIVE Just Now

Biden Celebrates New Partnership Between U.S., South Korea and Japan

- President Biden is holding a news conference with Prime Minister Fumio Kishida of Japan and President Yoon Suk Yeol of South Korea.
- The summit is aimed at countering China's regional dominance and ending decades of friction between the two Asian countries.

See more updates 9+

Here is why it is crucial for Japan and South Korea to get along as part of Washington's strategy in Asia and beyond.

WATCH WITH ANALYSIS

Troop Deaths and Injuries in Ukraine War Near 500,000, U.S. Officials Say

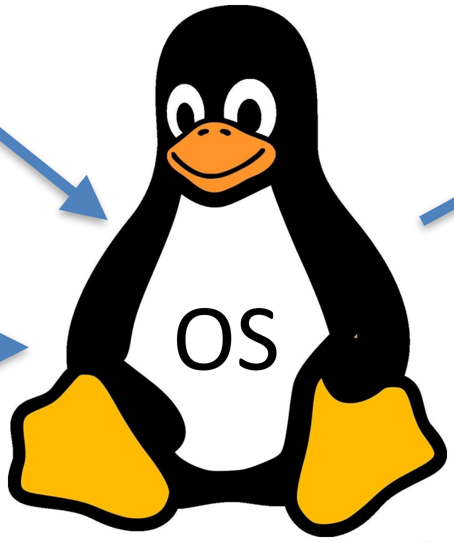
Ukraine and Russia have lost a staggering number of troops as Kyiv's counteroffensive drags on. A lack of rapid medical care has added to the toll.

5 MIN READ

Donetsk region A tank being checked and loaded with shells.

Russia said a Ukrainian drone damaged a building in Moscow's financial center.

See more headlines 8+



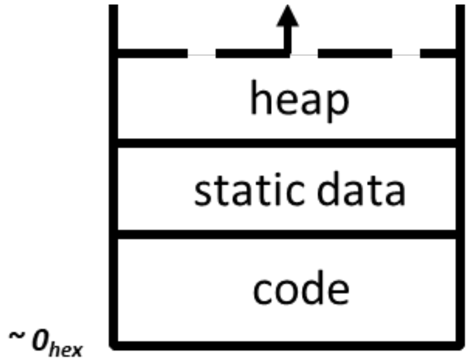
ABSTRACTIONS

- Files, not raw bytes on the disk
- Sockets, not packets on the network interface
- Teletype/Terminal interface, not keyboard scancodes
- etc.

View of a process

- Process: program that is being executed
- Contains code, data, and a thread
    - Thread contains registers, instruction pointer, and stack

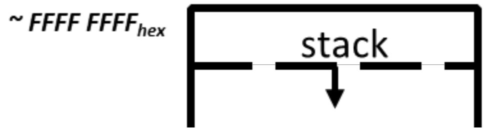
- Code and Data



- Registers

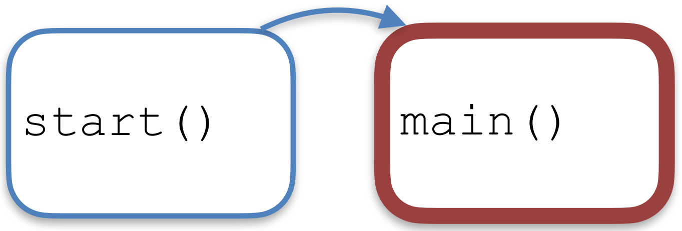
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes
- Stack

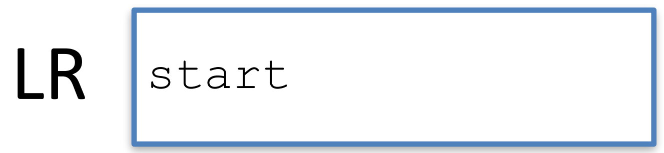


STACK REVIEW

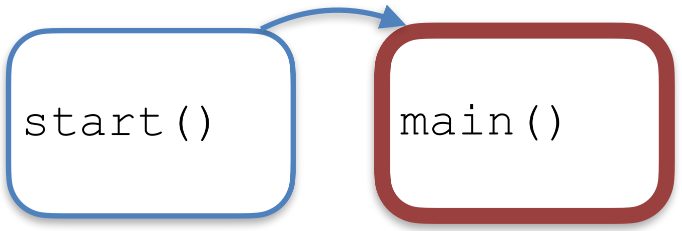
SAME WORKS FOR A FUNCTION CALL TREE

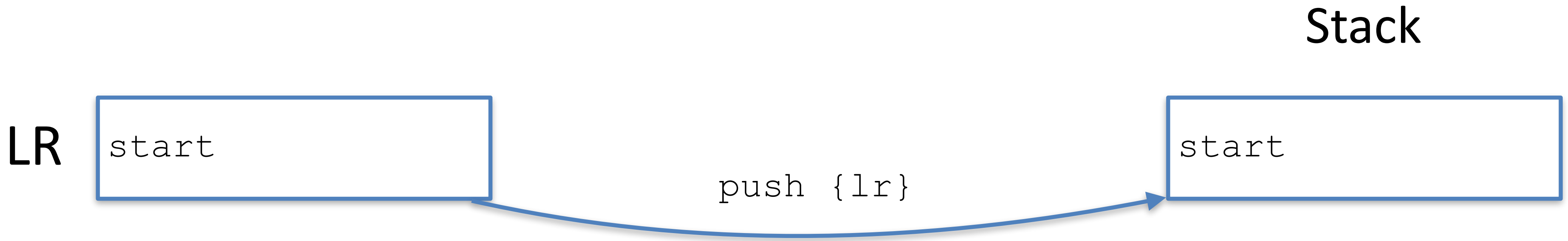


Stack

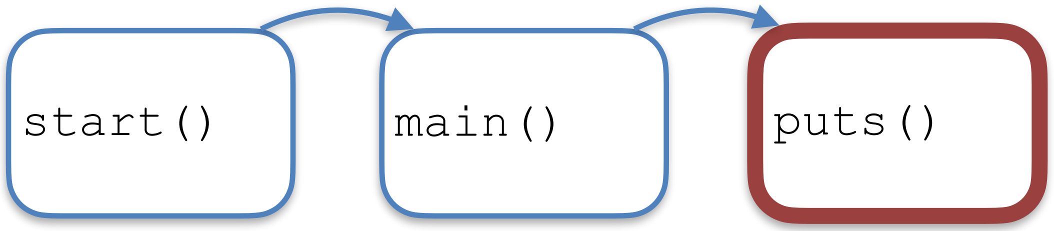


SAME WORKS FOR A FUNCTION CALL TREE

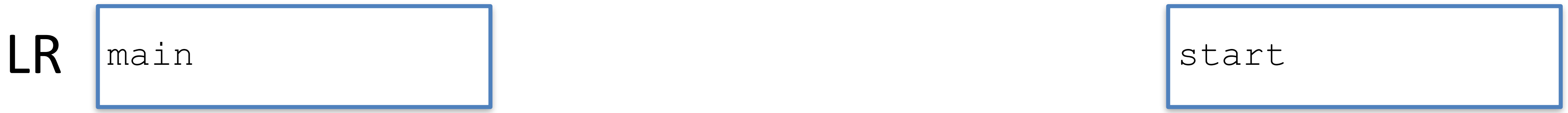




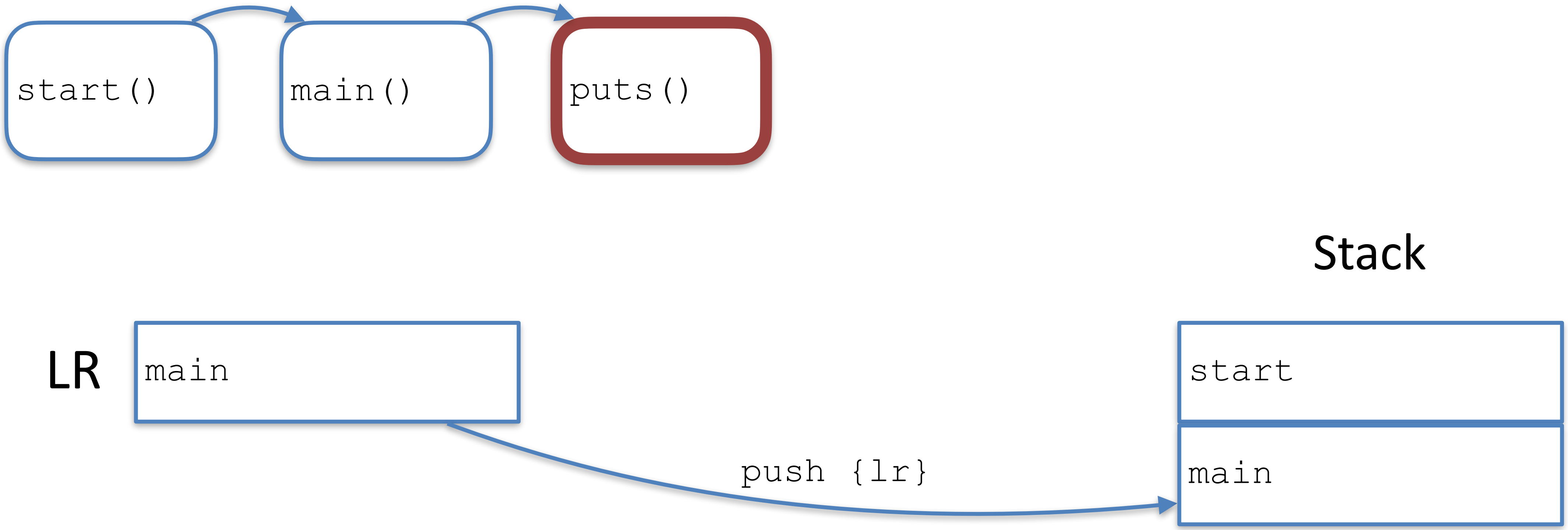
SAME WORKS FOR A FUNCTION CALL TREE



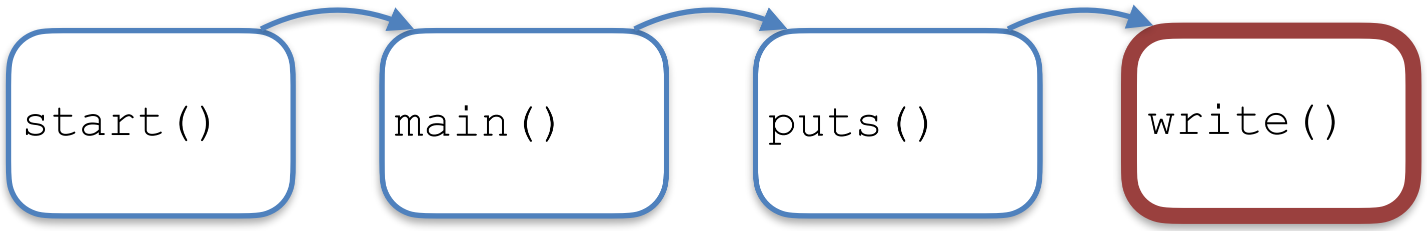
Stack

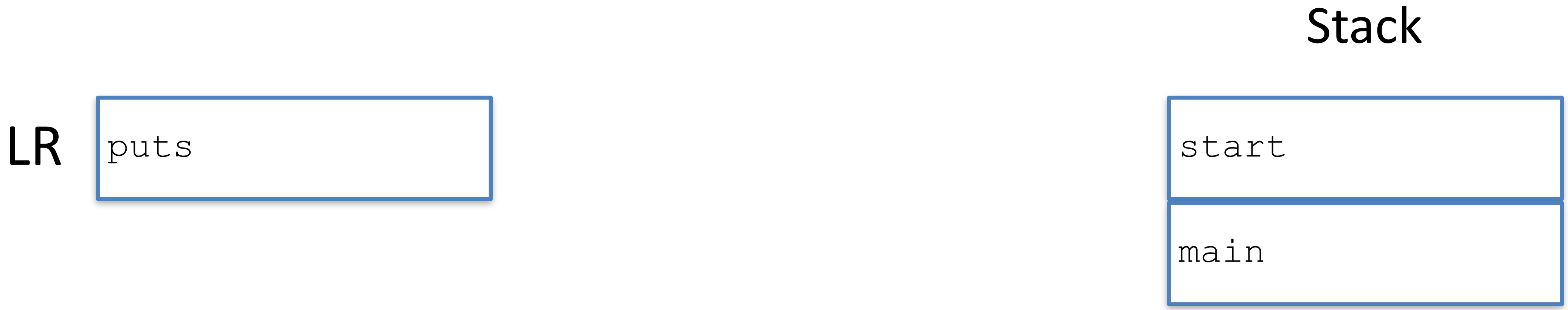


SAME WORKS FOR A FUNCTION CALL TREE

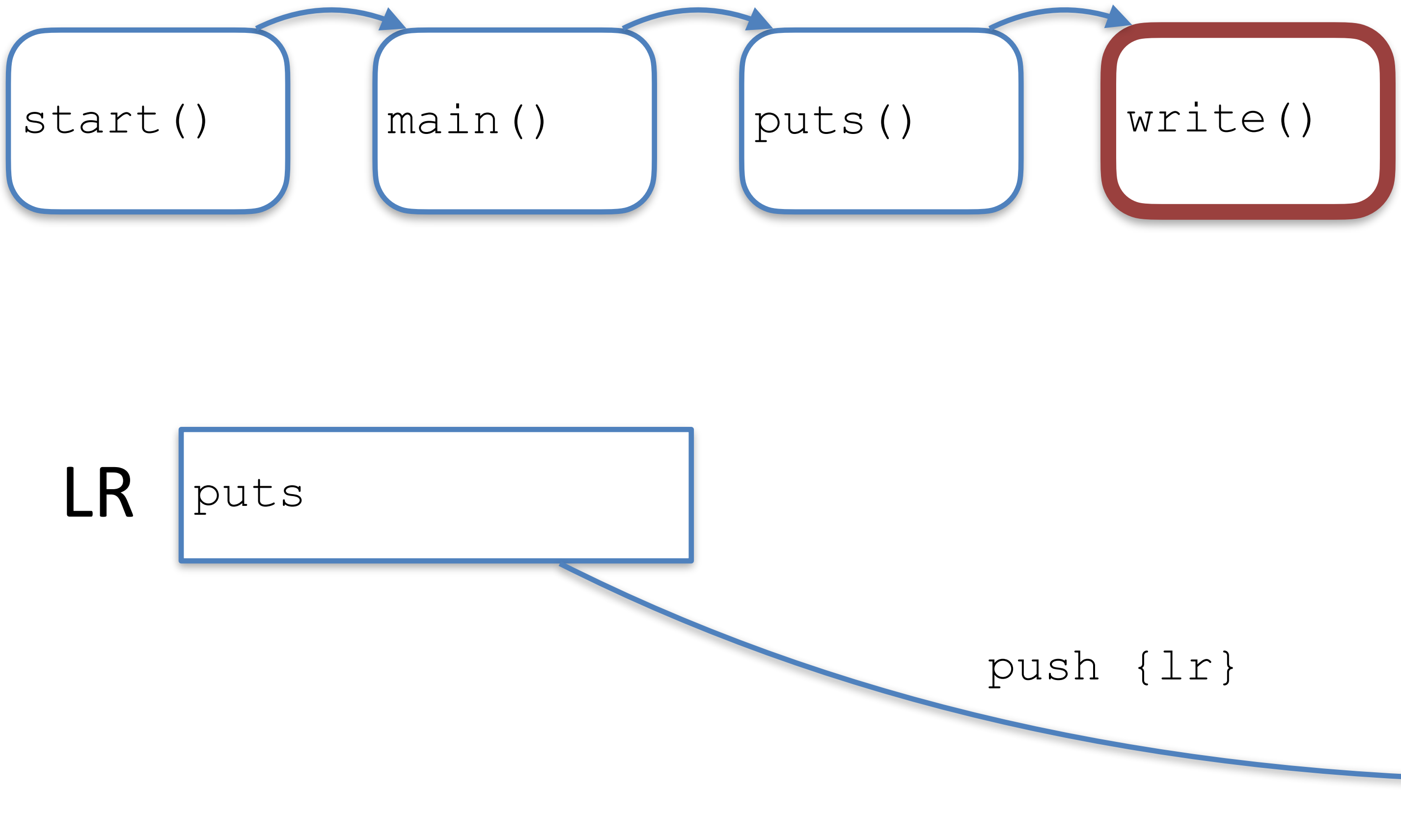


SAME WORKS FOR A FUNCTION CALL TREE

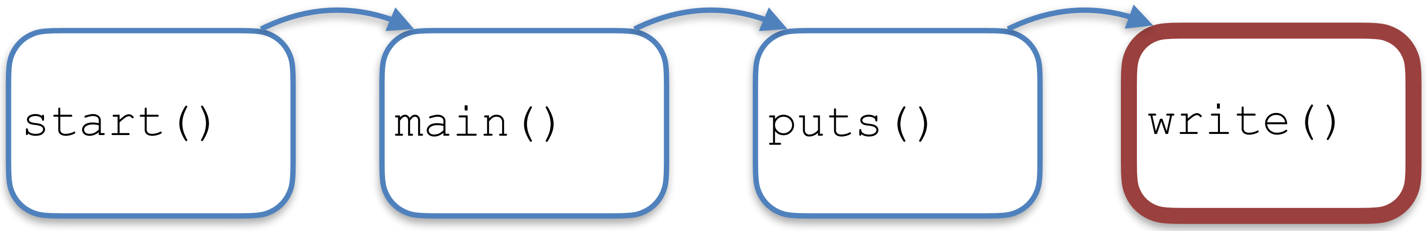


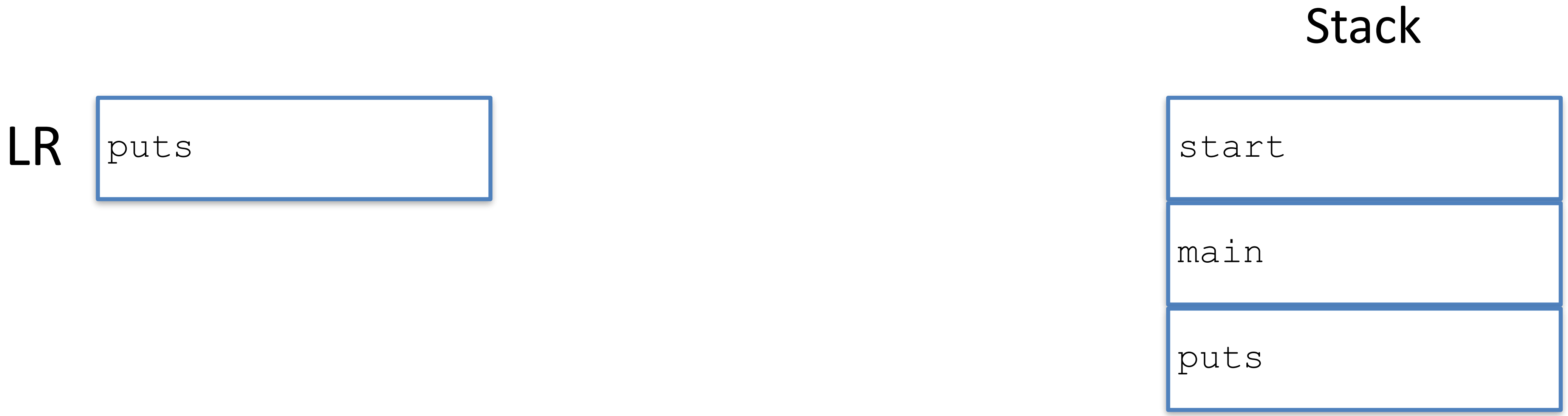


SAME WORKS FOR A FUNCTION CALL TREE

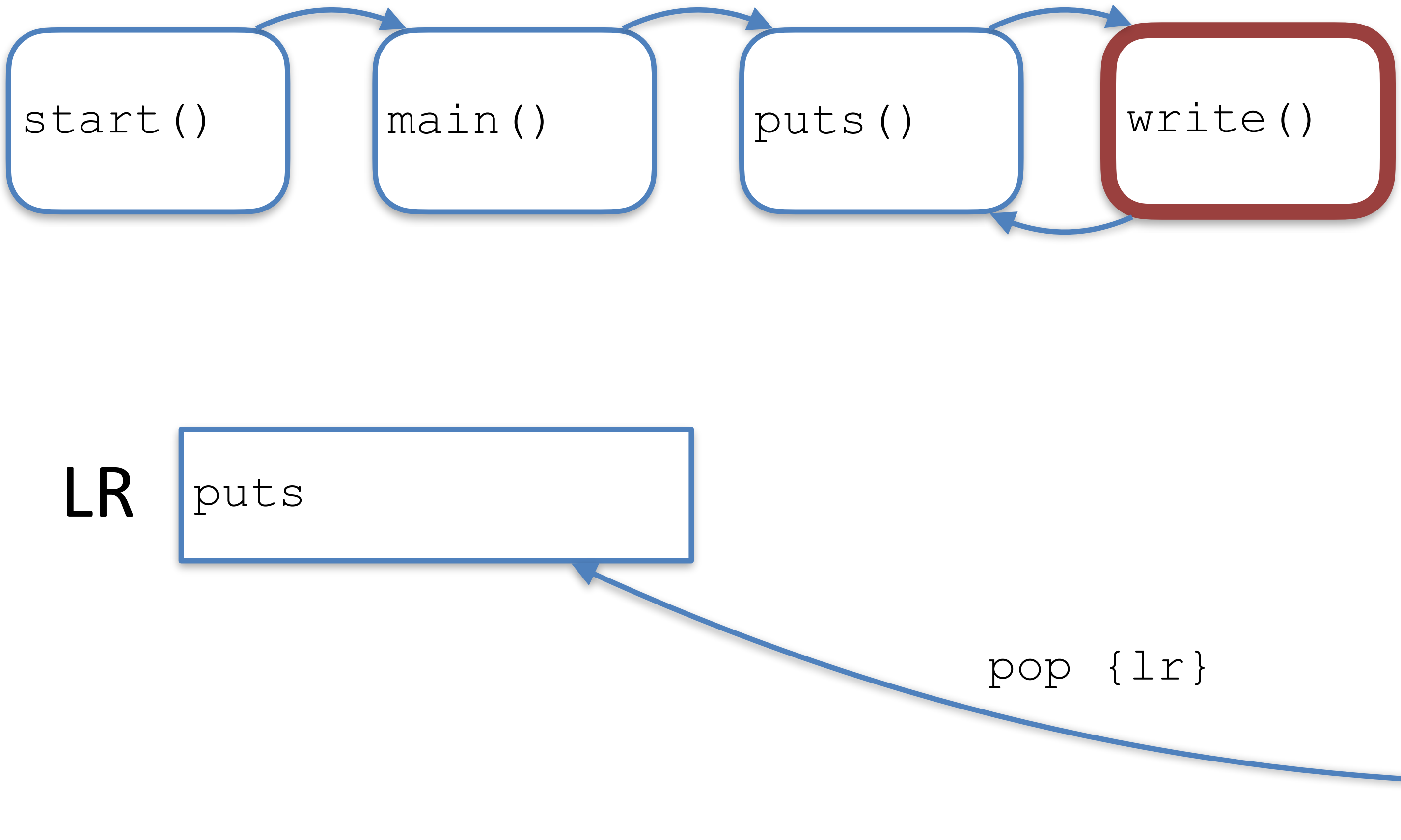


SAME WORKS FOR A FUNCTION CALL TREE

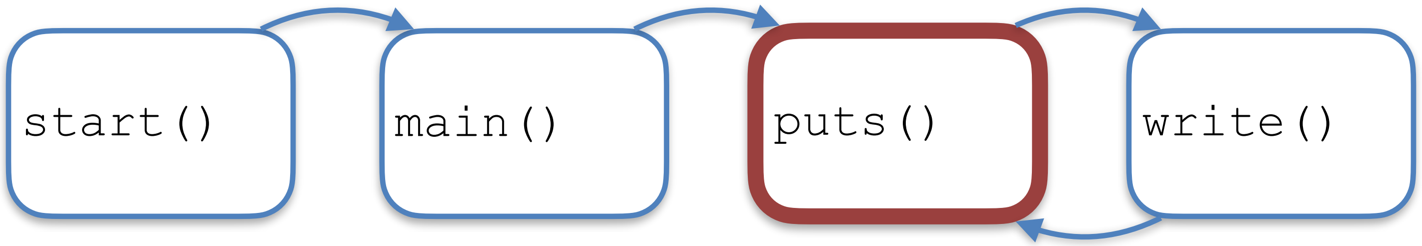


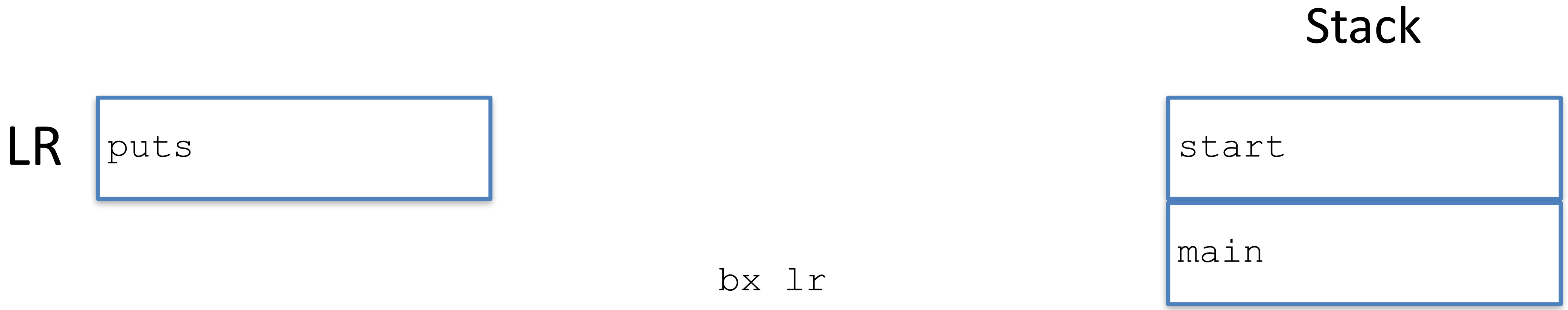


SAME WORKS FOR A FUNCTION CALL TREE



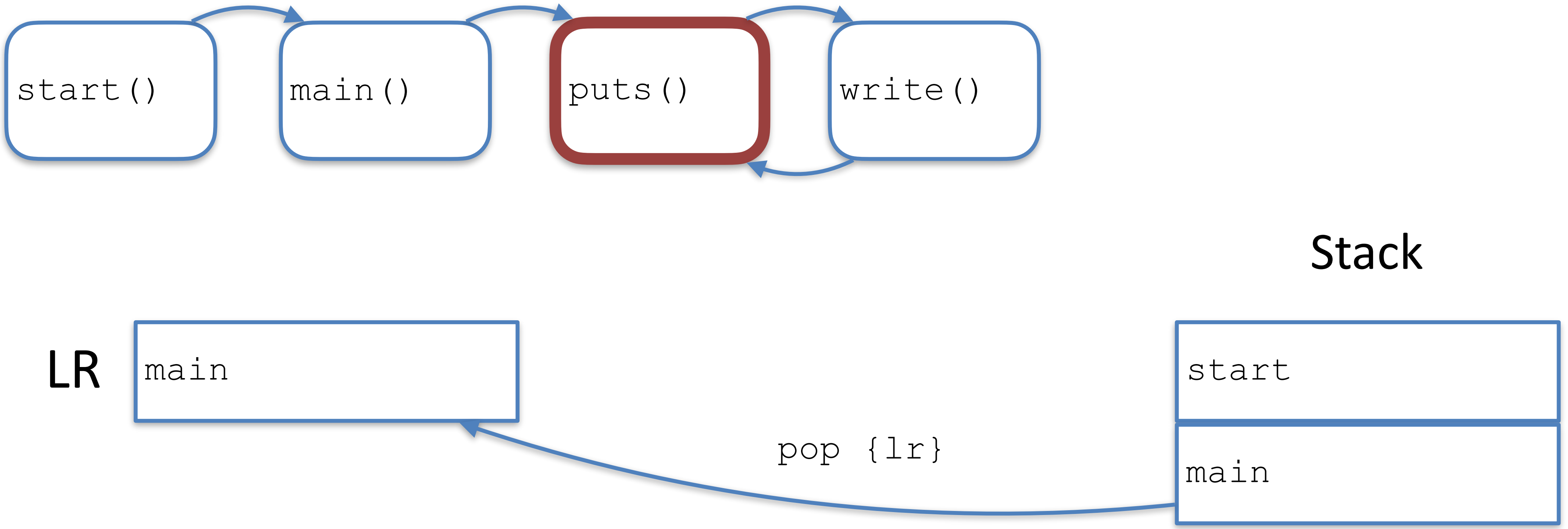
SAME WORKS FOR A FUNCTION CALL TREE



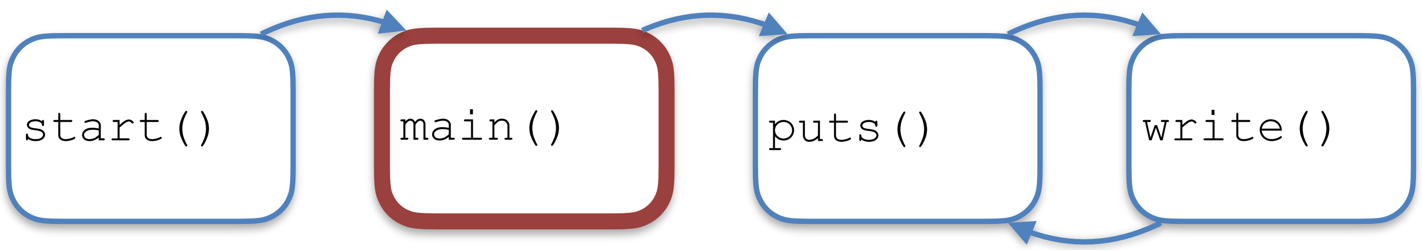


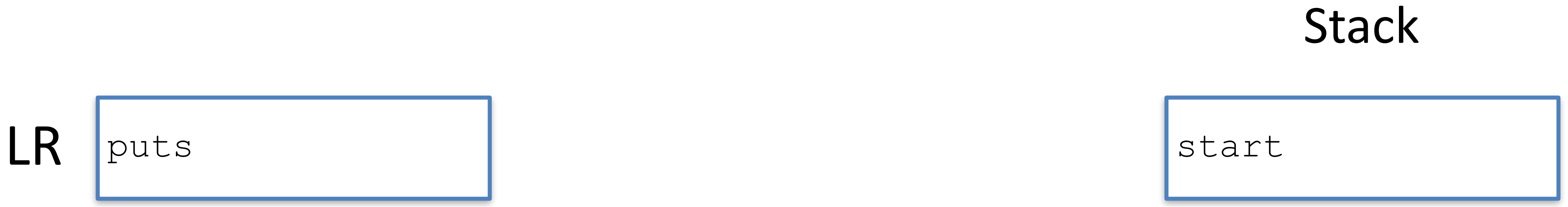
```
bx lr
```

SAME WORKS FOR A FUNCTION CALL TREE



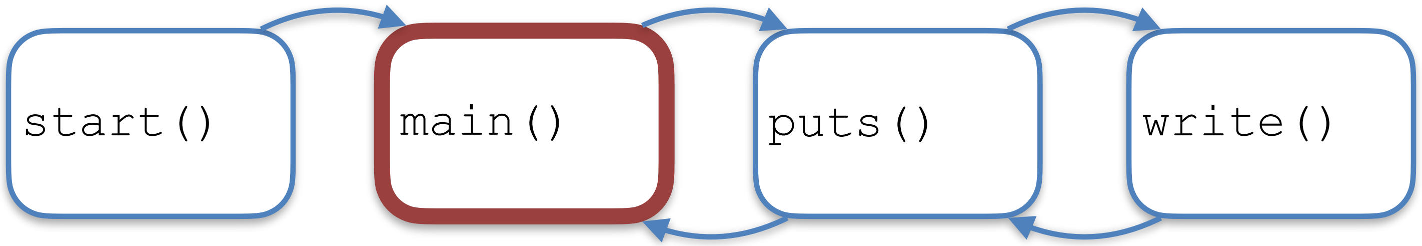
SAME WORKS FOR A FUNCTION CALL TREE

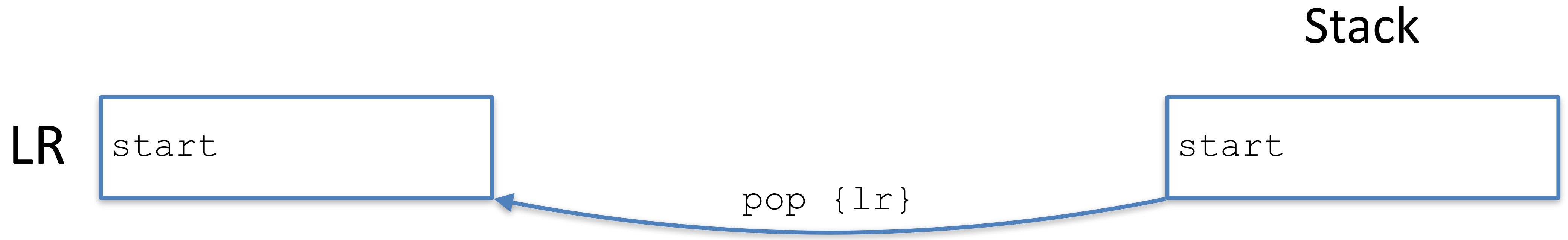




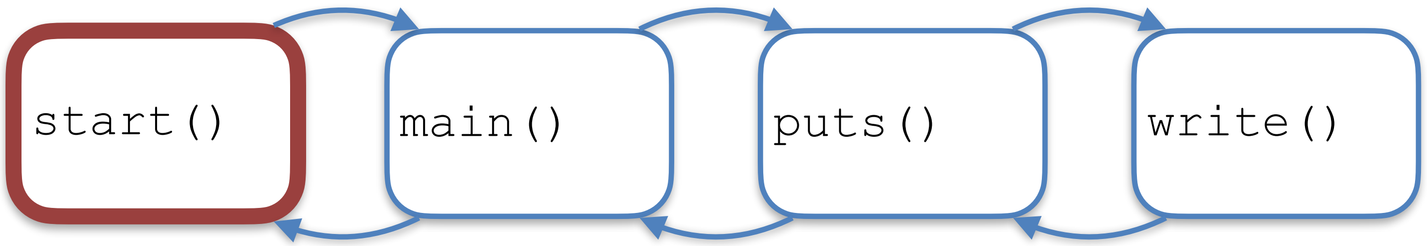
```
bx lr
```

SAME WORKS FOR A FUNCTION CALL TREE

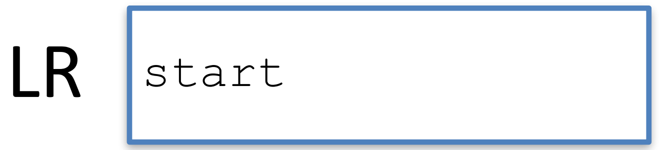




SAME WORKS FOR A FUNCTION CALL TREE



Stack



`bx lr`

PROLOGUE AND EIPLOGUE

```
function:
   push {lr}        ← Prologue

   …do some stuff…

   pop {lr}         ← Epilogue
   bx lr
```

STACK FRAMES

CLOBBERED REGISTERS

```
main:
  push {lr}
  ldr r4,=10       ; Init local variable
  ldr r0,=string   ; strlen(string)
  bl strlen        ; Compute length of string
  pop {lr}
  bx lr
strlen:
  push {lr}
  ldr r4,=0        ; Init iterator
  …compute string length…
  pop{lr}
  bx lr
```

CLOBBERED REGISTERS

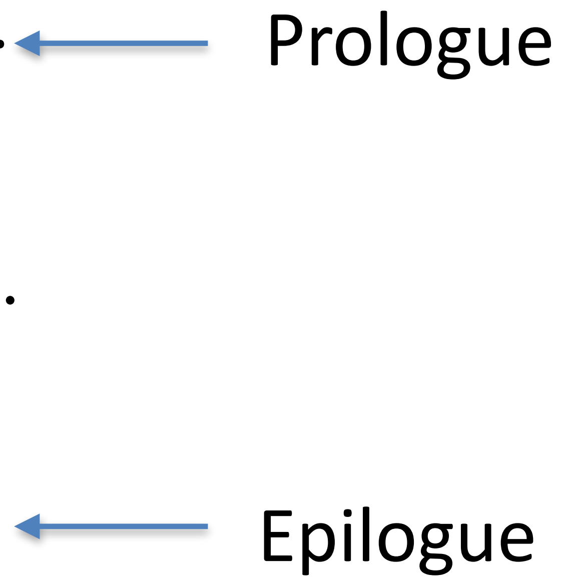
```
main:
  push {lr}
  ldr r4,=10          ; Init local variable
  ldr r0,=string      ; strlen(string)
  bl strlen           ; Compute length of string
  pop {lr}
  bx lr
strlen:
  push {lr}
  ldr r4,=0           ; Init iterator
  …compute string length…
  pop{lr}
  bx lr
```

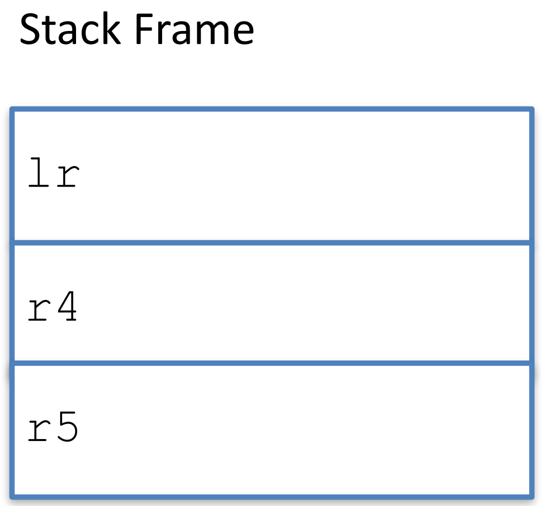
`main` and `strlen` both use `r4` for local variables

`strlen` overwrites `main`’s local variable!

PROLOGUE AND EIPLOGUE

```
function:
    push {lr,r4,r5}

    ...do some stuff...

    pop {lr,r4,r5}
    bx lr
```

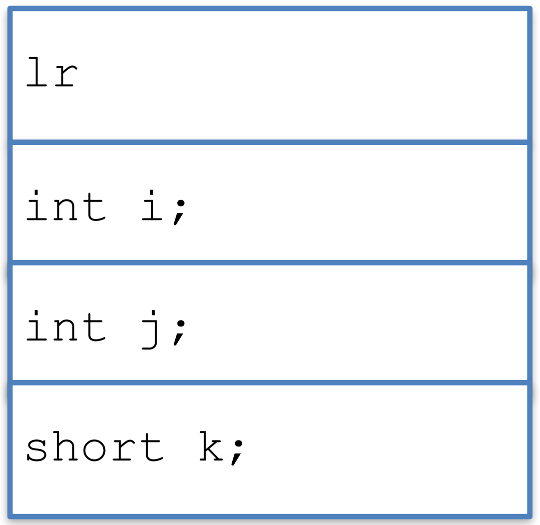




LOCAL VARIABLES

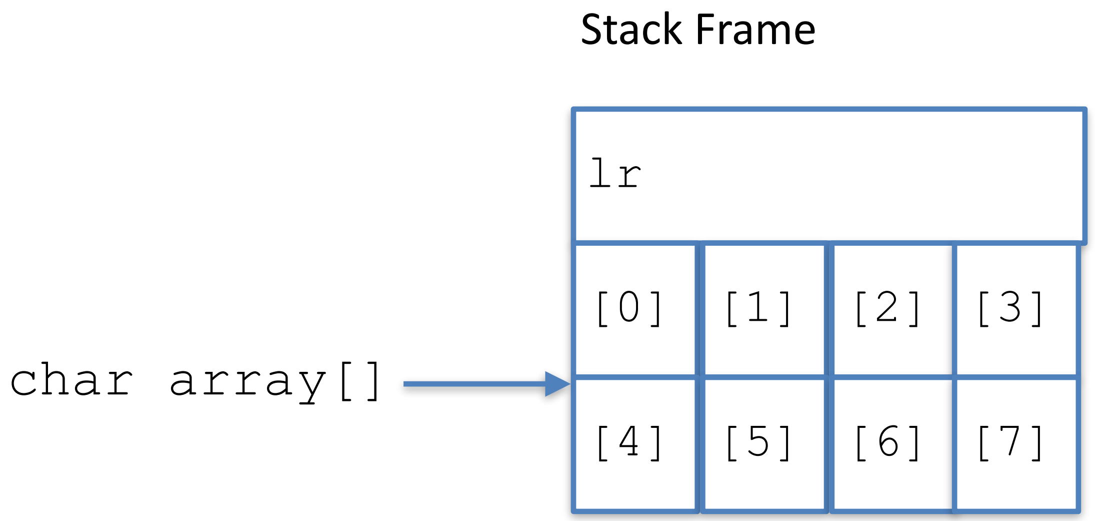
```
function() {
    int i, j;
    short k;
    …
}
```

Stack Frame



LOCAL VARIABLES

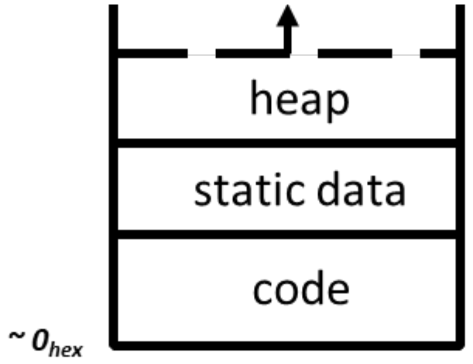
```
function() {
    char array[8];
    …
}
```



View of a process

- Process: program that is being executed
- Contains code, data, and a thread
  - Thread contains registers, instruction pointer, and stack

- Code and Data

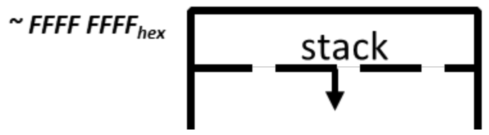


- Registers

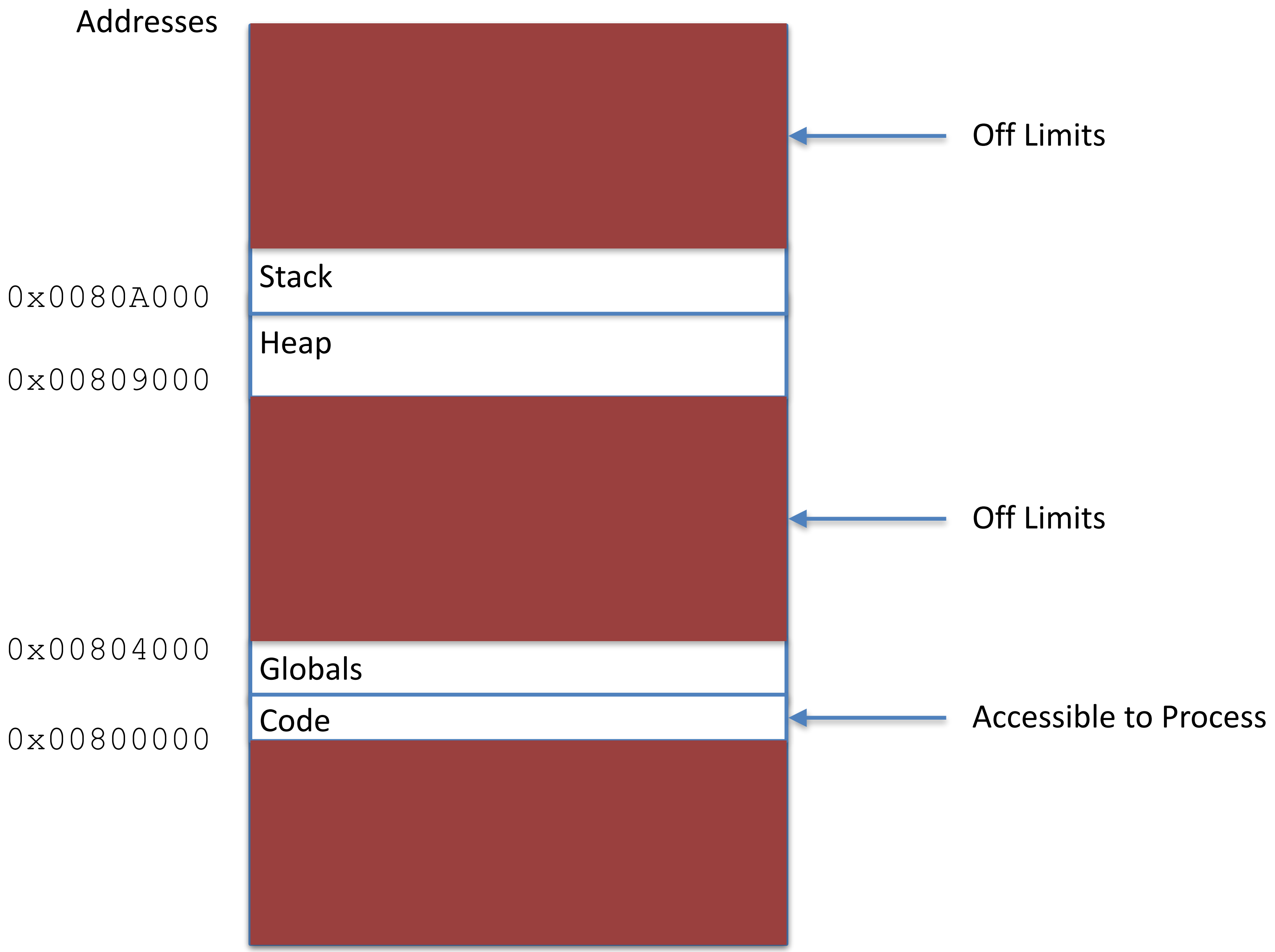
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes

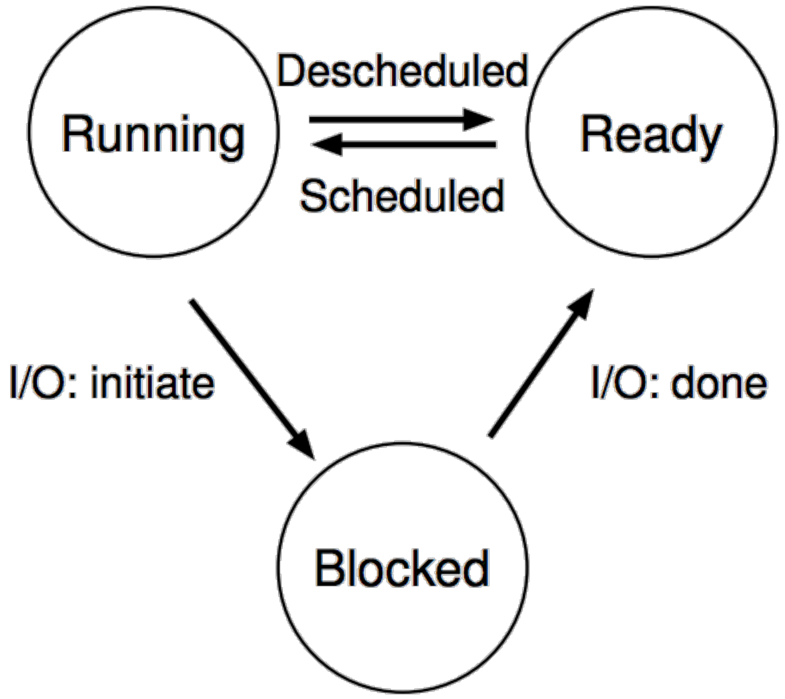
- Stack



PROCESS MEMORY ISOLATION



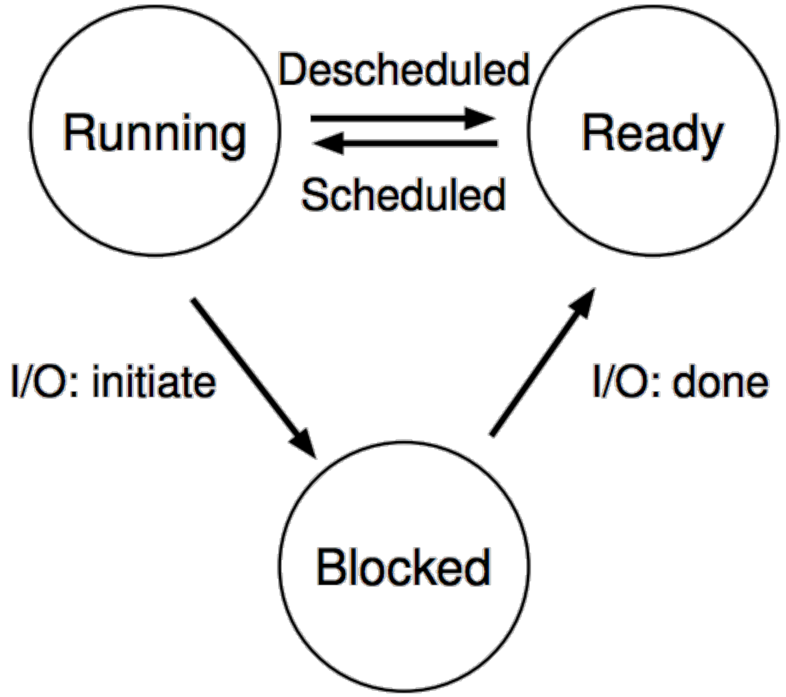
The three basic process states:



- OS ***schedules*** processes
  - Decides which of many competing processes to run.
- A ***blocked*** process is not ready to run.
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke, updating display.
  - While waiting for results, the process often cannot do anything, so it **blocks**, and the OS schedules a different process to run.

Multiprogramming processes

The three basic process states:



- When one process is Blocked, OS can schedule a different process that is Ready
- Even with a single processor, the OS can provide the illusion of many processes running simultaneously
- OS usually sets a maximum runtime before switching limit for processes (timeslice)

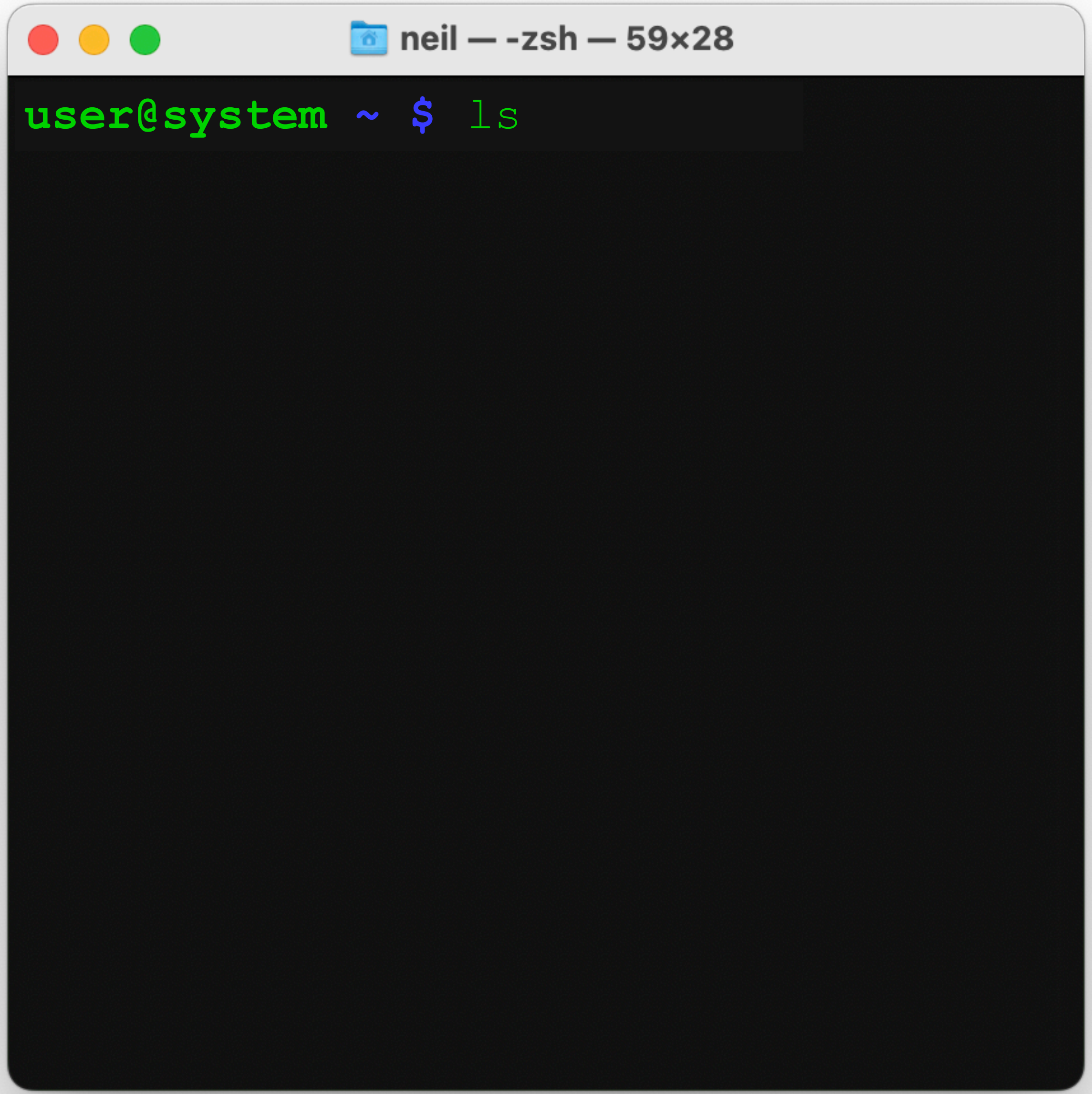
Key difference between kernel and processes: privilege

- Processes have limited access to the computer
  - Hardware supports different “modes” of execution (kernel and user)
  - Kernel mode has access to physical memory and special instructions

- They run when the OS lets them
- They have access to the memory the OS gives them

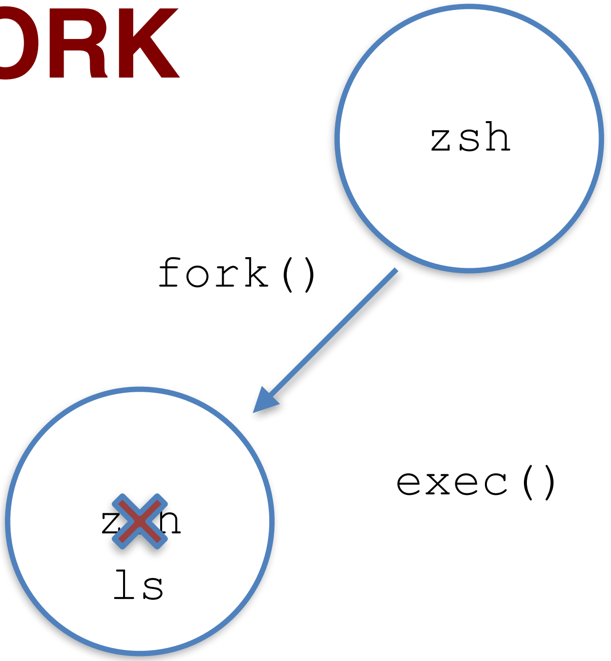
- They cannot access many things directly
  - Must ask the OS to do so for them

CREATING A NEW PROCESS: FORK



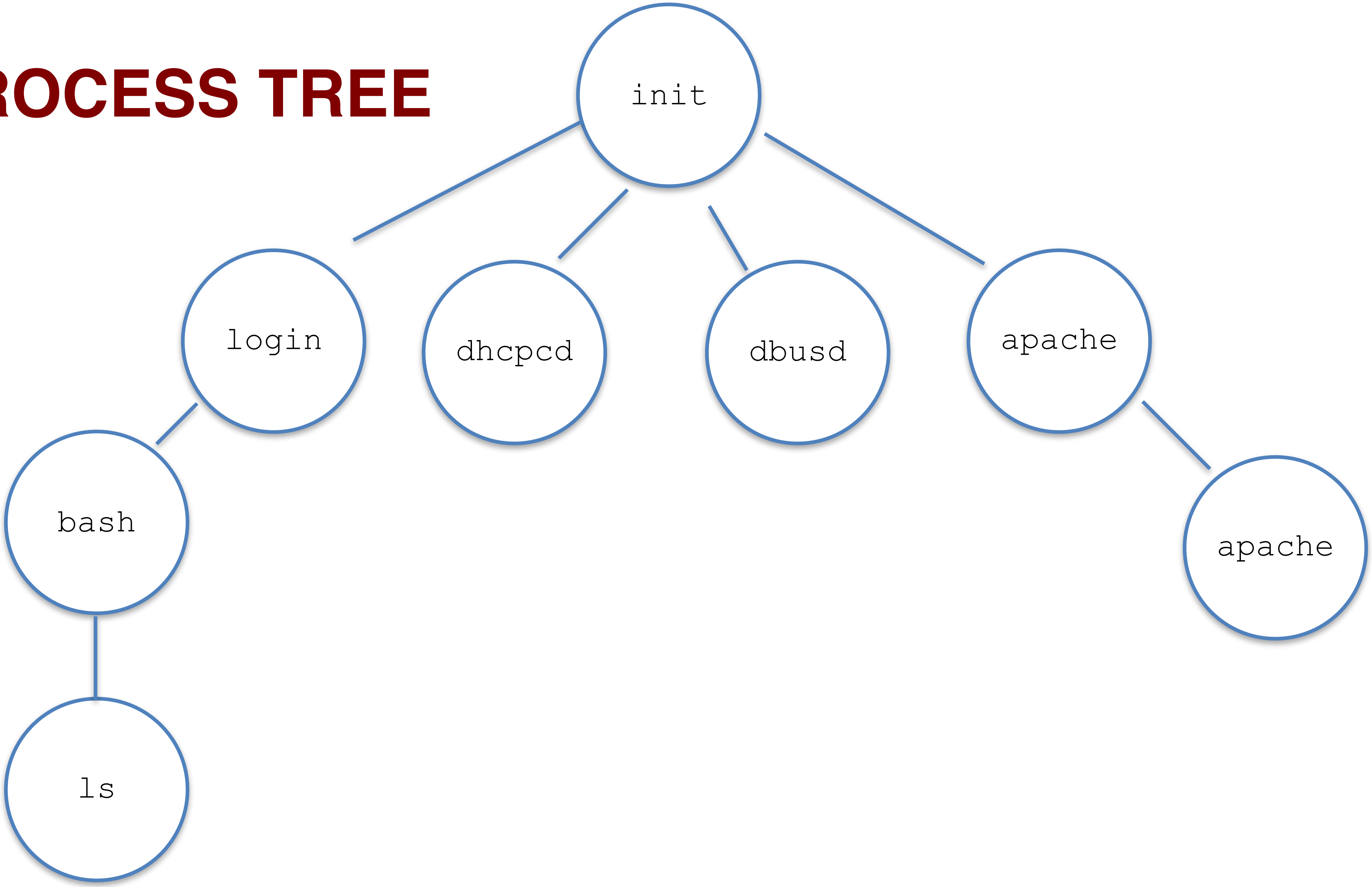
neil — -zsh — 59×28

```
user@system ~ $ ls
```



$$\begin{tikzcd}
\text{zsh} \arrow[d, "\text{fork()}"'] \\
\text{\cancel{zsh}}\ \text{ls} \arrow[r, phantom, "\text{exec()}"] & {}
\end{tikzcd}$$

PROCESS TREE



PROCESS SYSCALLS

```
pid_t fork(void);
```

- Create a new process that is a copy of the current one
- Returns either PID of child process (parent) or 0 (child)

```
void _exit(int status);
```

- Exit the current process (exit(), the library call cleans things up first)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Suspends the current process until a child (*pid*) terminates

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- Execute a new program, replacing the existing one

CREATING A NEW PROCESS

```
#include <stdio.h>
#include <unistd.h>

int main() {
  if( fork() == 0) {
    printf("CHILD\n");
  } else {
    printf("PARENT\n");
  }
  printf("BOTH\n");
  return 0;
}
```

CREATING A NEW PROCESS & WAITING FOR CHILD

```
#include <stdio.h>
#include <unistd.h>

int main() {
  int status;
  if( fork() == 0) {
    printf(“CHILD\n”);
  } else {
    printf(“PARENT\n”);
    wait(&status);   // Parent waits for child to finish
  }                  // and gets its return code in status.
  printf(“BOTH\n”);
  return 0;
}
```

EXECUTING A NEW PROGRAM

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    execve("/bin/python3", ...);
  } else {
    printf("Parent!\n");
  }

  printf("Only parent!\n");
  return 0;
}
```

FORK BOMB

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

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

- Creates a new process
  - Then each process creates a new process
  - Then each of those creates a new process…
- Known as a Fork bomb!
  - Machine eventually runs out of memory and processing power and will stop working

- Defense: limit number of processes per user

FORK BOMB

- Python fork bomb

```
import os
while 1:
  os.fork()
```

- Rust fork bomb

```
#[allow(unconditional_recursion)]
fn main() {
    std::thread::spawn(main);
    main();
}
```

- Bash fork bomb

```
:(){ :|:& };:
```

- With spacing and a clearer function name

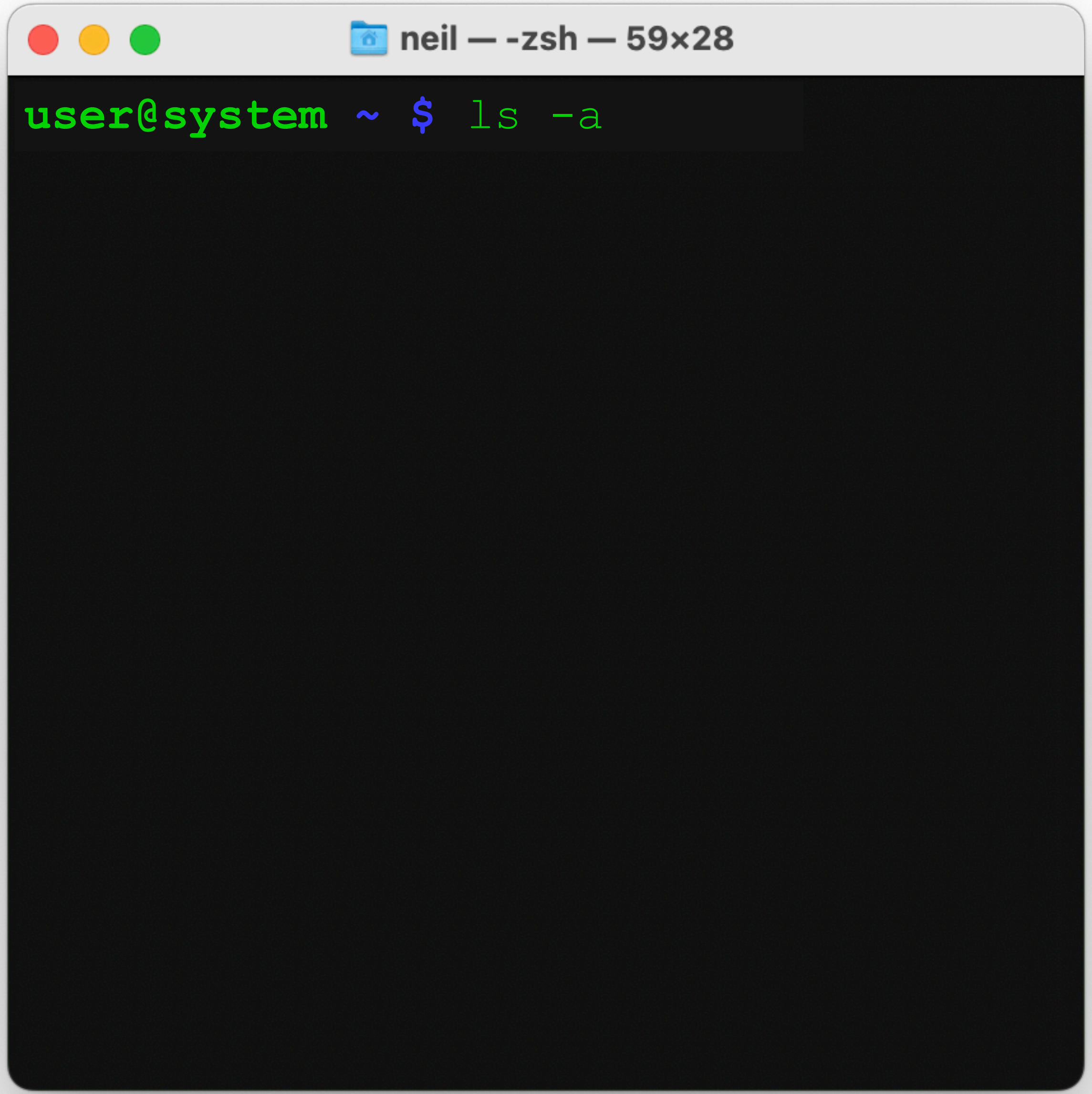
```
fork() {
    fork | fork &
}
fork
```

SHELL

ADMINISTRIVIA

- `shittyshell` In-class assignment today.
- Microsoft Teams Join Code `34iz1ae`
- Join link on course website
- Homework 1 Due Next Wednesday

WHAT DOES THE SHELL DO



```
user@system ~ $ ls -a
```

1. Get a command (`ls -a`)
2. Parse the command
   - First token is the name of the binary to run. Need to convert to a full path on disk.
   - `ls` → `/bin/ls`
3. Build an `argv` for the new process.
4. `fork()` / `execve()`

PARSING THE COMMAND

Command Buffer:

`l`	`s`		`-`	`a`	`\0`

1. Is this a complete path to a binary on disk?
   - If yes, we don't need to do anything because the user told us what binary they want to run.
   - If no, we need to find the complete path to the binary on the disk.
   - You can tell if it's a complete path by looking at the first character: if it's a '/', then it's a complete path.
2. If not a complete path, use the PATH variable to find the binary.
   - PATH = "`/bin/`" command = "`ls`"
   - Concatenate PATH with command: "`/bin/ls`"

BUILDING THE ARGUMENT VECTOR

Command Buffer (char array):

`l`	`s`		`-`	`a`	`\0`

Full Path to Binary (char array):

`/`	`b`	`i`	`n`	`/`	`l`	`s`	`\0`

Argument Vector (char* array):

`&fullpath[0]`	`&cmdbuf[3]`	`NULL`

1. **Argument vector is an array of pointers, not array of chars.**
2. **Each element of the argument vector array holds the address of a char.**
   - **& in C means "address of"**

TODO: ADD A PARAMETER TO ARGV

```
#include <stdio.h>
#include <unistd.h>

void main() {
    char cmd[] = "/bin/ls";
    char *argv[] = {&cmd[0], NULL};
    execv(cmd,argv);
}
```

SIGNALS

ALERTING PROCESSES OF EVENTS

- How do we let a process know there was an event?
  - Errors
  - Termination
  - User commands (like CTRL-C or CTRL-\)
- Events could happen whenever
  - Need to interrupt process control flow and run an event handler
  - Linux mechanism to do so is called “signals”

SIGNALS ARE ASYNC MESSAGES TO PROCESSES

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called "Signals"

```
 1) SIGHUP      2) SIGINT    3) SIGQUIT    4) SIGILL   5) SIGTRAP
 6) SIGABRT     7) SIGBUS    8) SIGFPE     9) SIGKILL 10) SIGUSR1
11) SIGSEGV    12) SIGUSR2 13) SIGPIPE   14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT   19) SIGSTOP 20) SIGTSTP
21) SIGTTIN    22) SIGTTOU 23) SIGURG    24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO    30) SIGPWR
31) SIGSYS     ...
```

SIGNALS ARE ASYNC MESSAGES TO PROCESSES

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called "Signals"

Process Errors

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      ...
```

SIGNALS ARE ASYNC MESSAGES TO PROCESSES

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called “Signals”

Process Termination

```
 1) SIGHUP      2) SIGINT    3) SIGQUIT    4) SIGILL   5) SIGTRAP
 6) SIGABRT     7) SIGBUS    8) SIGFPE     9) SIGKILL 10) SIGUSR1
11) SIGSEGV    12) SIGUSR2  13) SIGPIPE   14) SIGALRM 15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD  18) SIGCONT   19) SIGSTOP 20) SIGTSTP
21) SIGTTIN    22) SIGTTOU  23) SIGURG    24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO   30) SIGPWR
31) SIGSYS     ...
```

SENDING SIGNALS

- OS sends signals when it needs to
- Processes can ask the OS send signals with a system call
  - `int kill(pid_t pid, int sig);`
- Users send signals through OS from command line or keyboard
  - Shell command: `kill -9 pid` (SIGKILL)
  - CTRL-C (SIGINT)

HANDLING SIGNALS

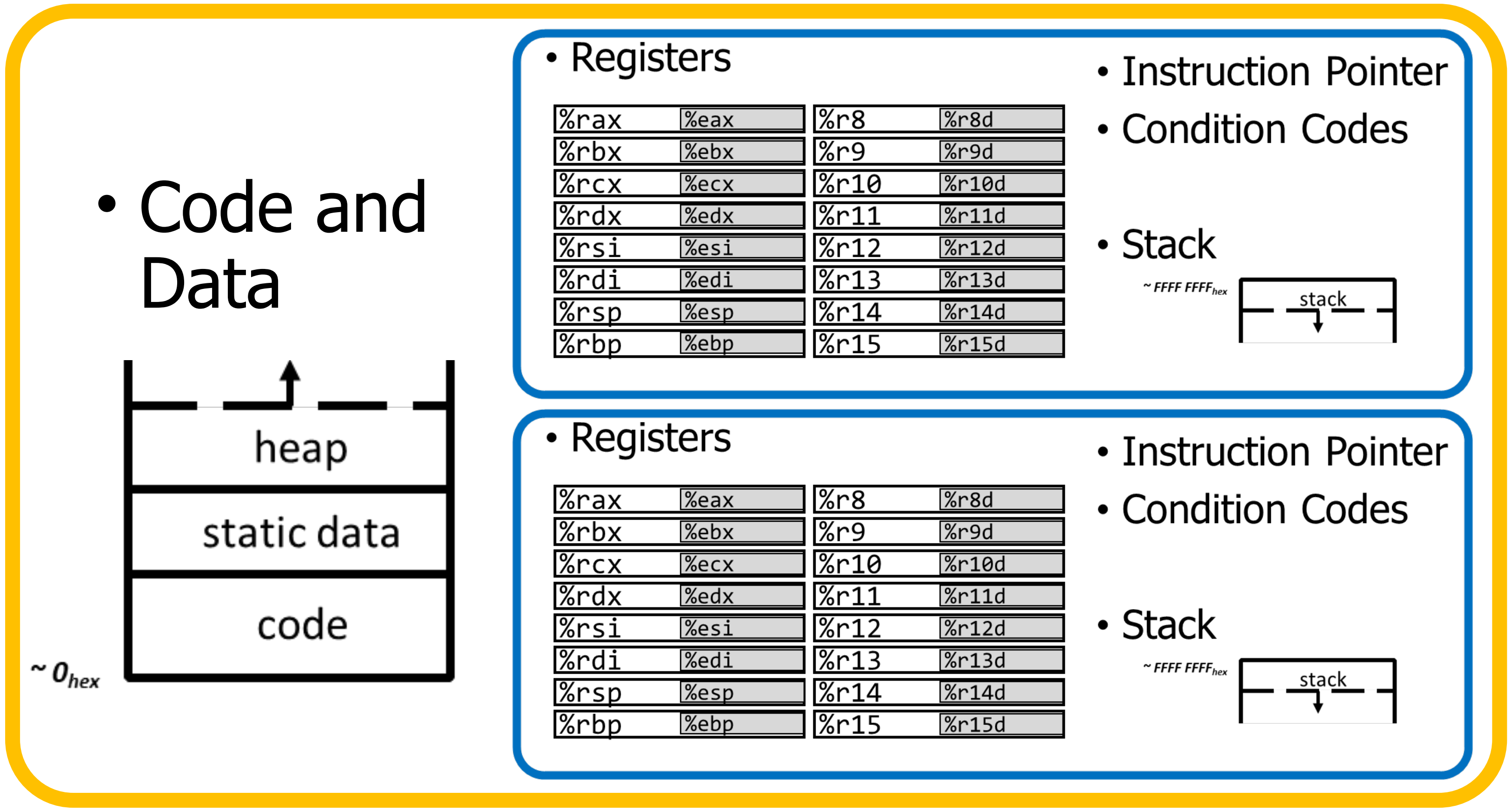
- Programs can register a function to handle individual signals
  - `signal(int sig, sighandler_t handler);`
- OS keeps track of signal handlers for each signal
  - Calls that function when a signal occurs
- What is the process supposed to do about it?
  - Do some quick processing to handle it
  - Reset the process and try again
  - Quit the process (default handler)

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sighandler (int signum) {
   printf("HA HA You can't kill me!\n");
}
void main (void) {
   signal(SIGINT, sighandler);
   printf("Starting\n");
   while(1) {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
}
```

THREADS

Alternate view of a process

- A process could have multiple threads
  - Each with its own registers and stack



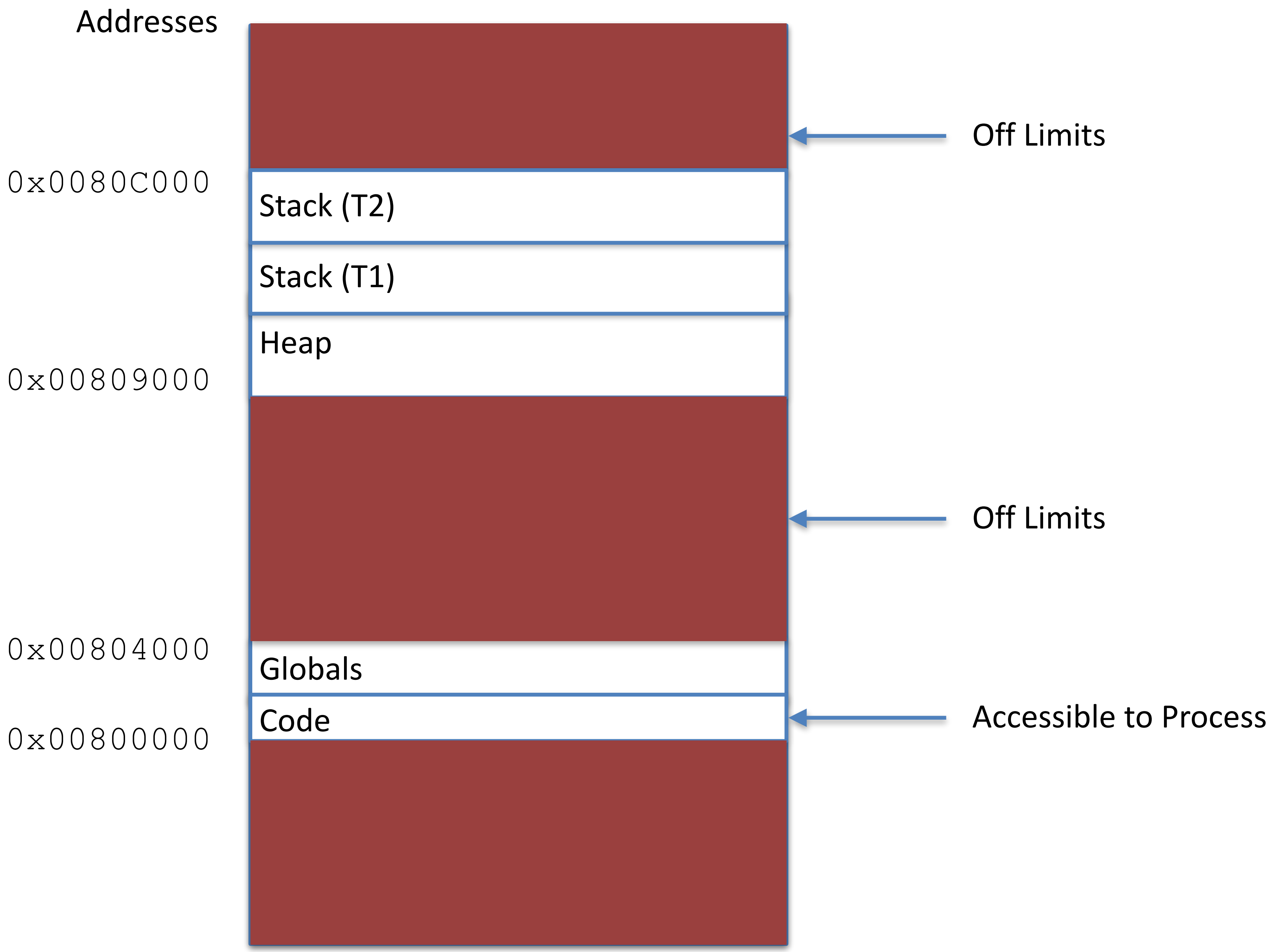
Threads have separate:

- Instruction Pointer
- Registers
- Stack Memory
- Condition Codes

Threads share:

- Code
- Global variables

PROCESS WITH MULTIPLE THREADS



Thread use case: web browser

Let's say you're implementing a web browser:

You want a tab for each web page you open:

- The same code loads each website (shared code section)
- The same global settings are shared by each tab (shared data section)
- Each tab does have separate state (separate stack and registers)

Disclaimer: Actually, modern browsers use separate processes for each tab for a variety of reasons including performance and security. But they used to use threads.

Thread use case: user interfaces

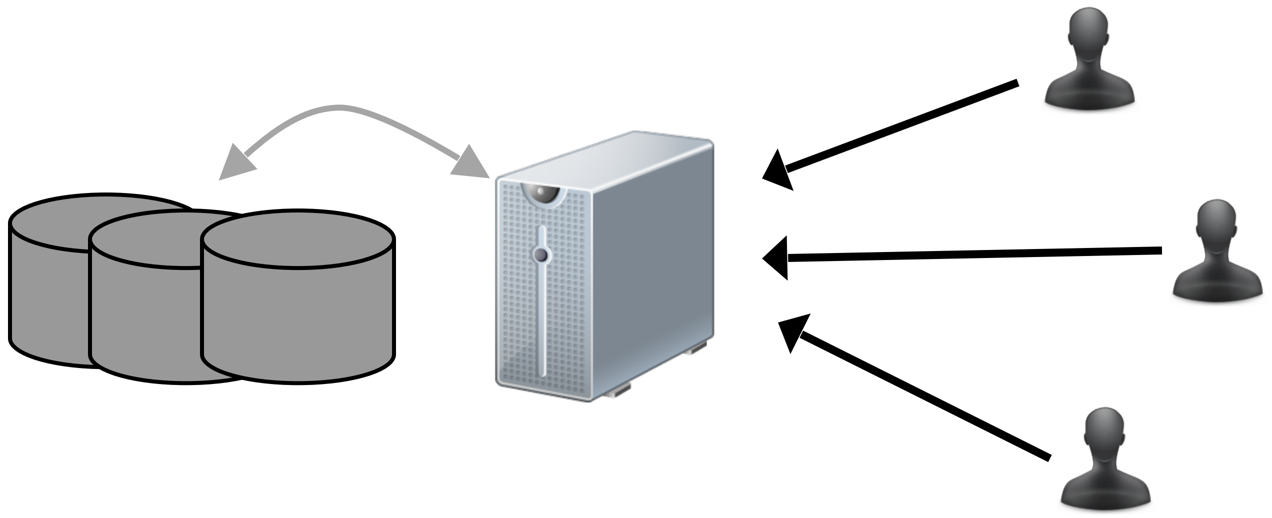
- Even if there is only a single processor core, threads are useful

- Single-threaded User Interface
  - While processing actions, the UI is frozen

```
main() {
   while(true) {
         check_for_UI_interactions();
         process_UI_actions(); // UI freezes while
processing
   }
}
```

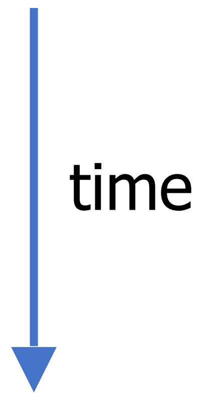
Thread use case: web server

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



Web server option 1: handle one request at a time

Request 1 arrives
Server reads in request 1
Server starts disk I/O for request 1
Request 2 arrives
Disk I/O for request 1 finishes
Server responds to request 1
Server reads in request 2



- Easy to program, but slow
  - Can’t overlap disk requests with computation
  - Can’t overlap either with network sends and receives

Web server option 1: event-driven model

- Issue I/Os, but don't wait for them to complete

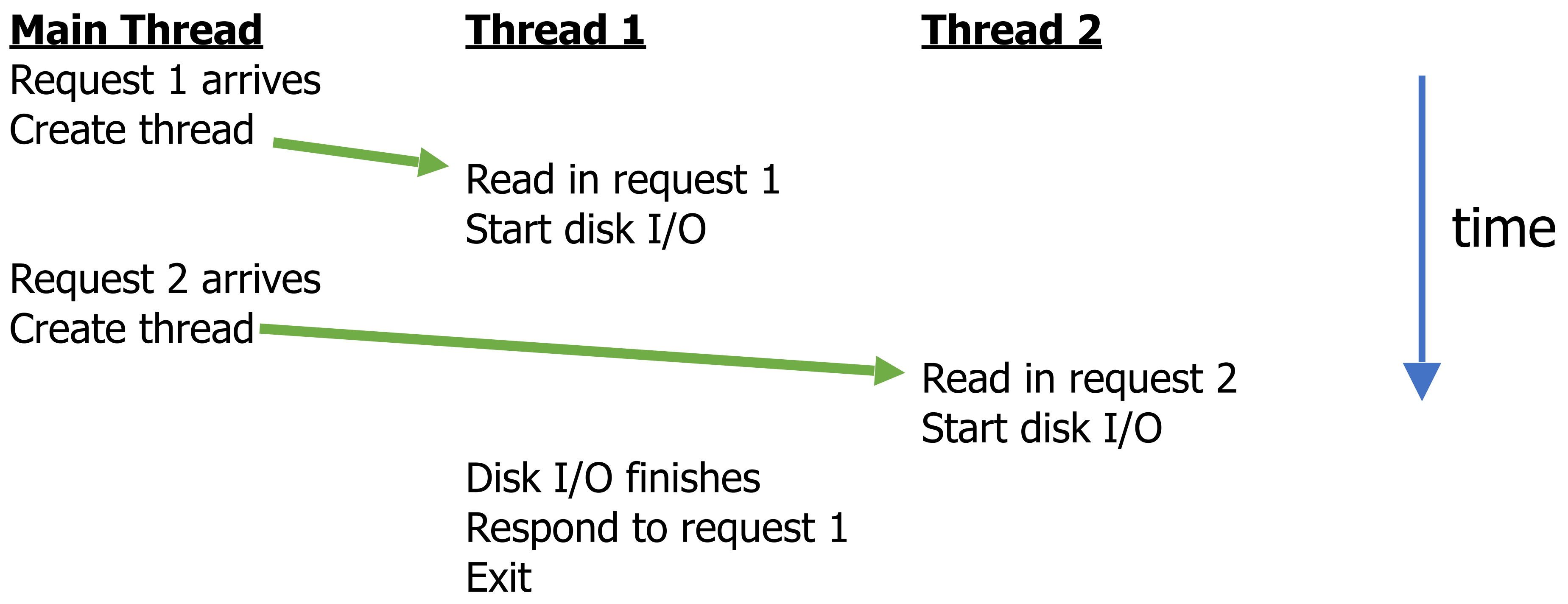
  Request 1 arrives
  Server reads in request 1
  Server starts disk I/O for request 1
  Request 2 arrives
  Server reads in request 2
  Server starts disk I/O for request 2
  Disk I/O for request 1 completes
  Server responds to request 1

time

- Fast, but hard to program
  - Must remember which requests are in flight and which I/O goes where
  - Lots of extra state

Web server option 3: multi-threaded web server

- One thread per request. Thread handles only that request.



- Easy to program (maybe), and fast!
    - State is stored in the stacks of each thread and the thread scheduler
    - Simple to program if they are independent…

More Practical Motivation

Back to Jeff Dean's "Numbers Everyone Should Know"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Handle I/O in separate thread, avoid blocking other progress

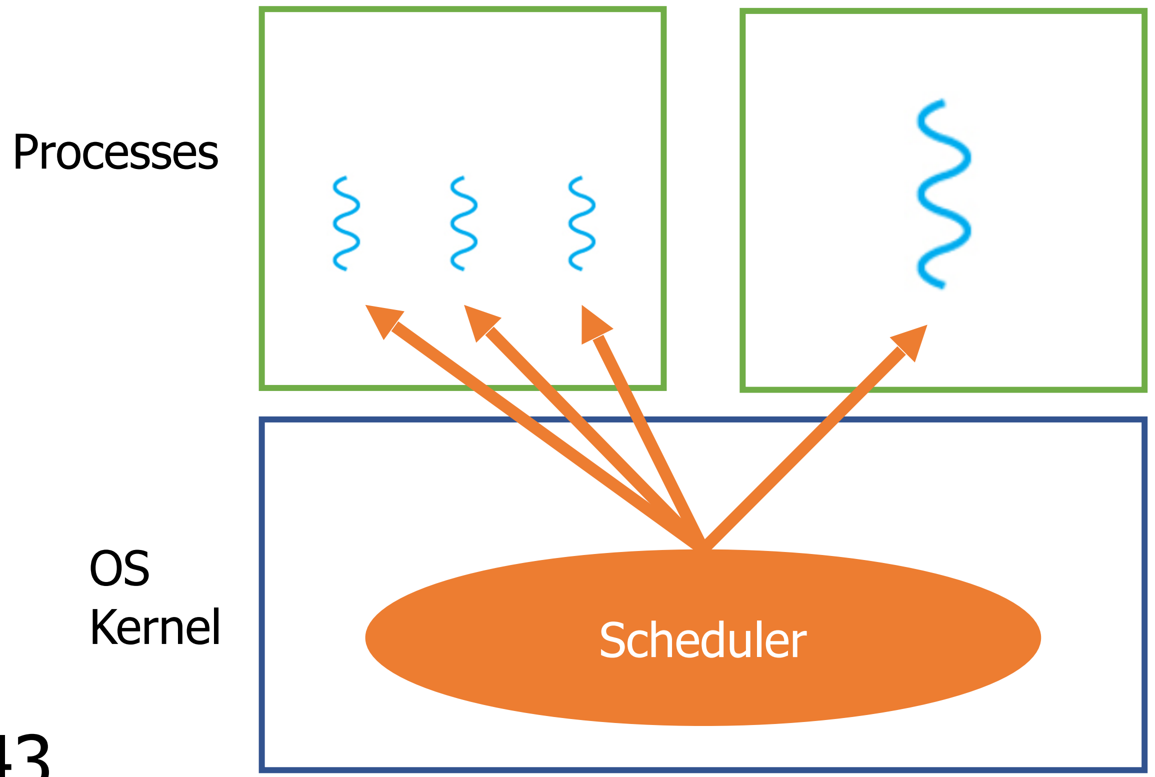
Models for thread libraries: **Kernel Threads**

- Thread scheduling is implemented by the operating system
    - OS manages the threads within each process

- Upsides
    - Other threads can continue while one blocks on I/O
    - No additional scheduler

- Downsides
    - Higher overhead

- This is what we'll focus on in CS343



Threads versus Processes

Threads

- **`pthread_create()`**
  - Creates a thread
  - **Shares** all memory with all threads of the process.
  - Scheduled independently of parent
- **`pthread_join()`**
  - Waits for a particular thread to finish
- Can communicate by reading/ writing (shared) global variables.

Processes

- **`fork()`**
  - Creates a single-threaded process
  - **Copies** all memory from parent
    - Can be quick using copy-on-write
  - Scheduled independently of parent
- **`waitpid()`**
  - Waits for a particular child process to finish
- Can communicate by setting up shared memory, pipes, reading/ writing files, or using sockets (network).

POSIX Threads Library: pthreads

- https://man7.org/linux/man-pages/man7/pthreads.7.html

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

  - thread is created executing start_routine with arg as its sole argument.
  - return is implicit call to pthread_exit

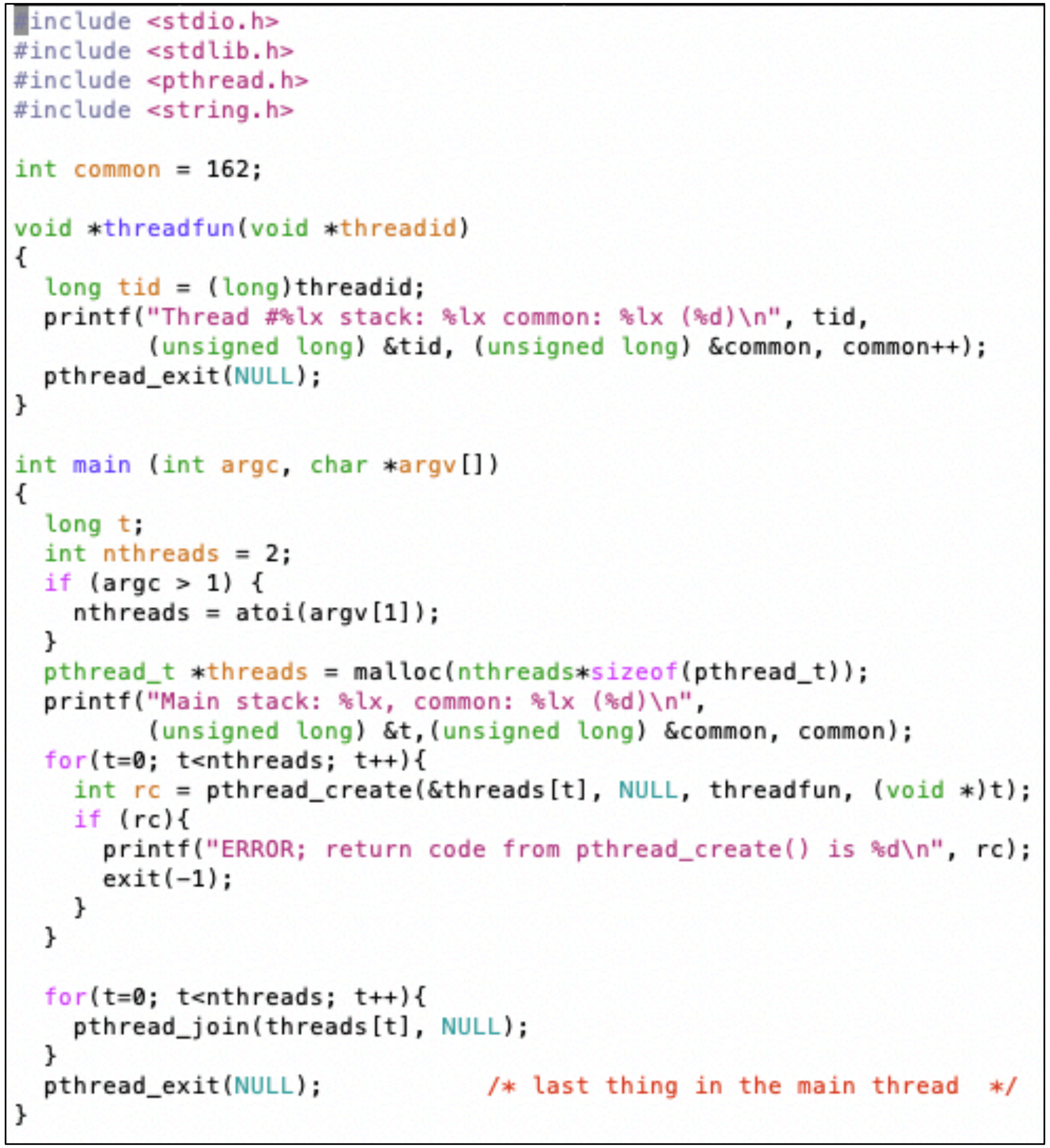
```
void pthread_exit(void *value_ptr);
```

  - terminates the thread and makes value_ptr available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

  - suspends execution of the calling thread until the target thread terminates.
  - On return with a non-NULL value_ptr the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr.

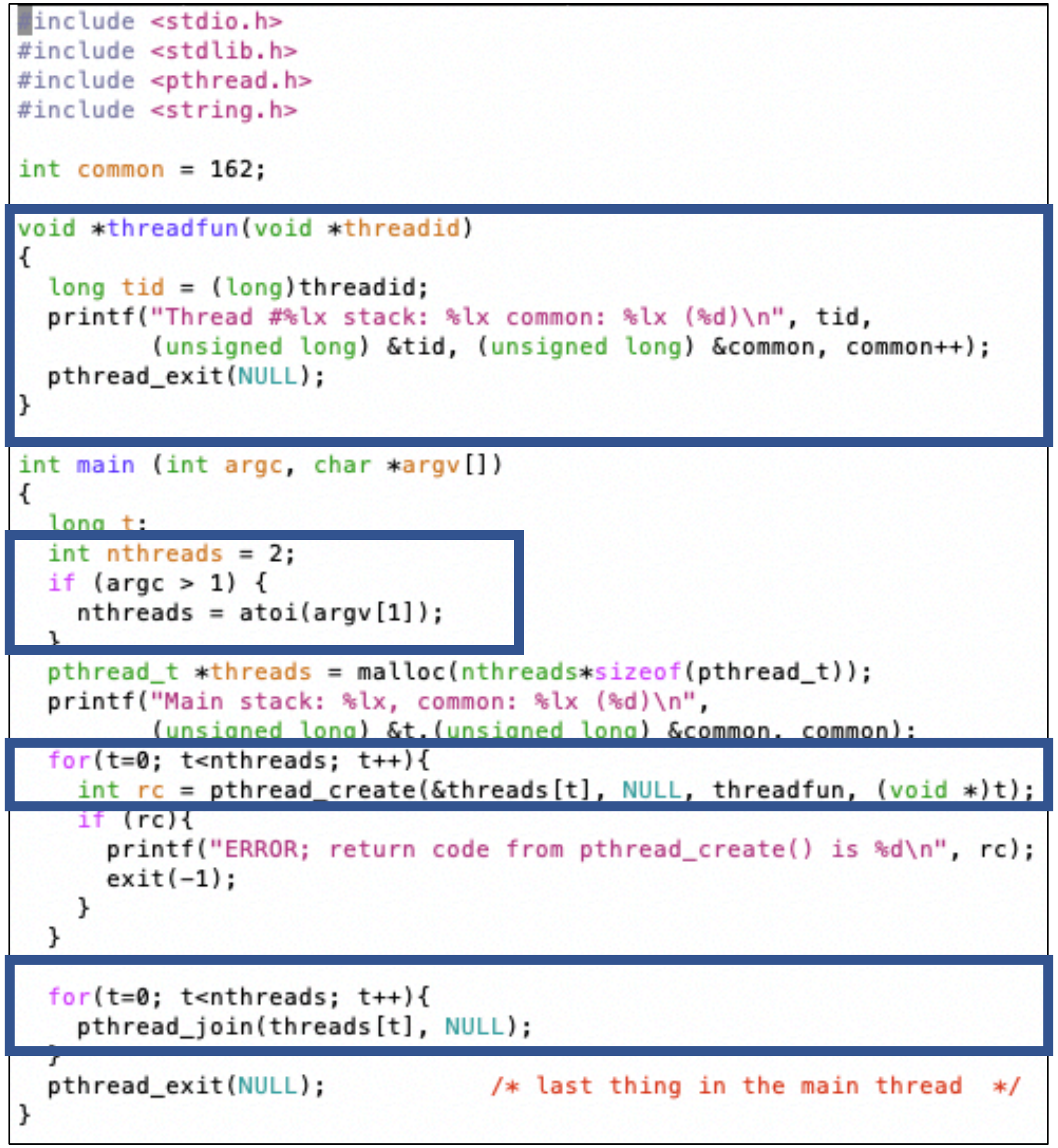
Threads Example



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

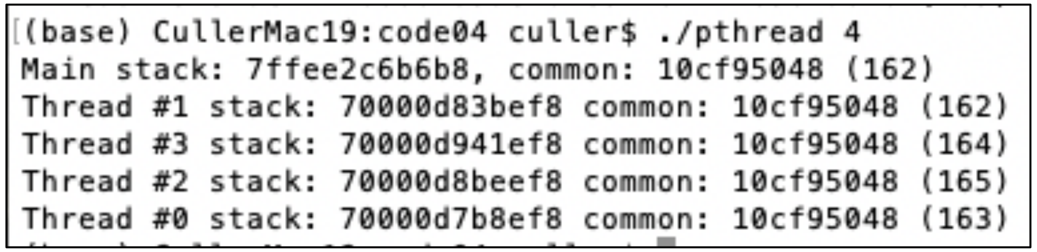
Threads Example

- Reads N from process arguments
- Creates N threads
- Each one prints a number, then increments it, then exits
- Main process waits for all of the threads to finish

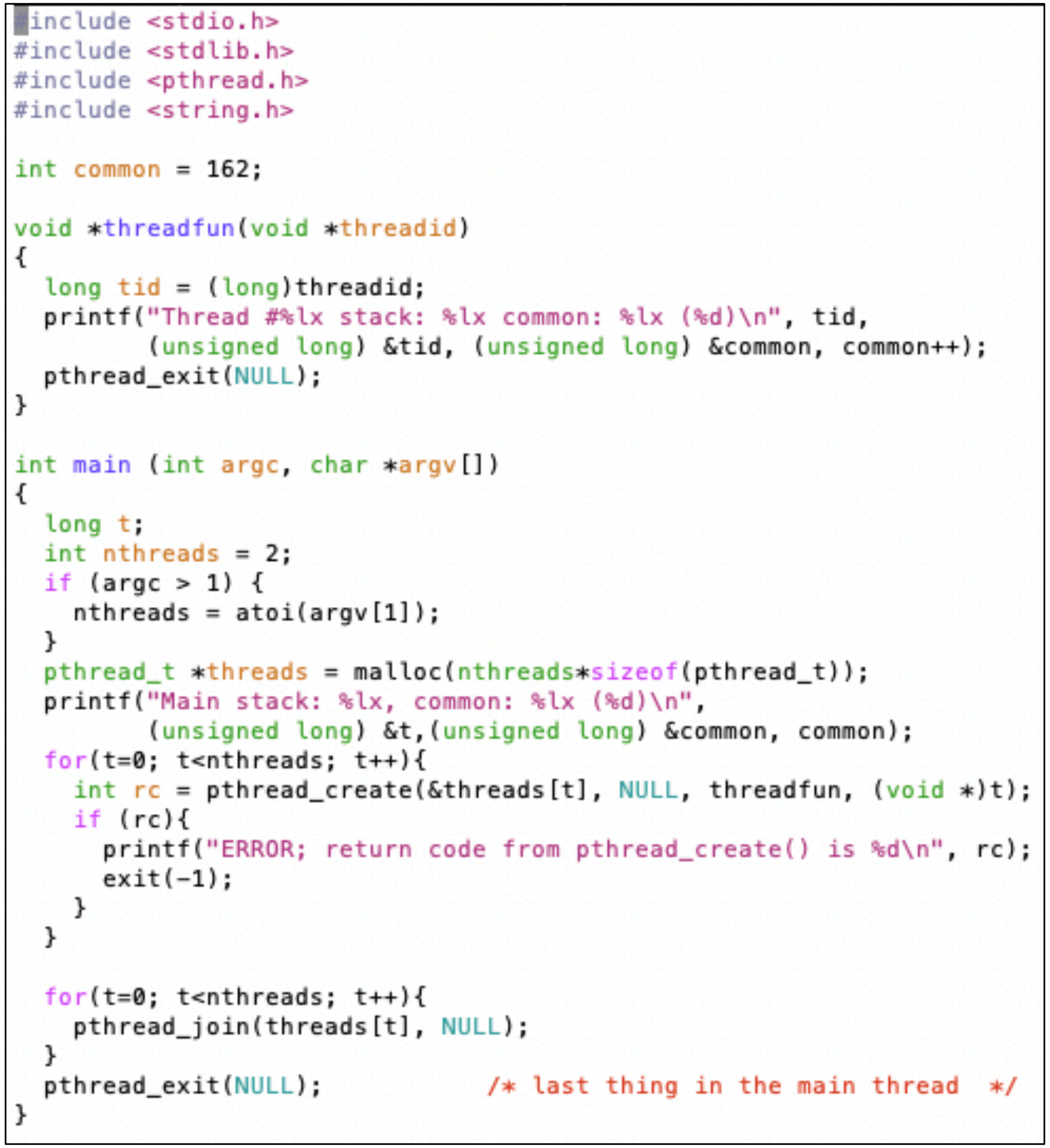


```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Threads Example

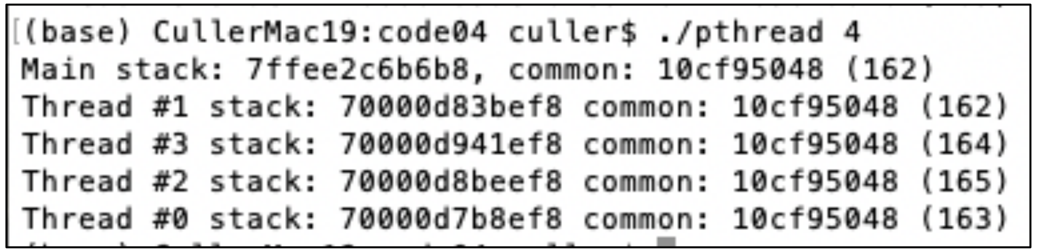


```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```



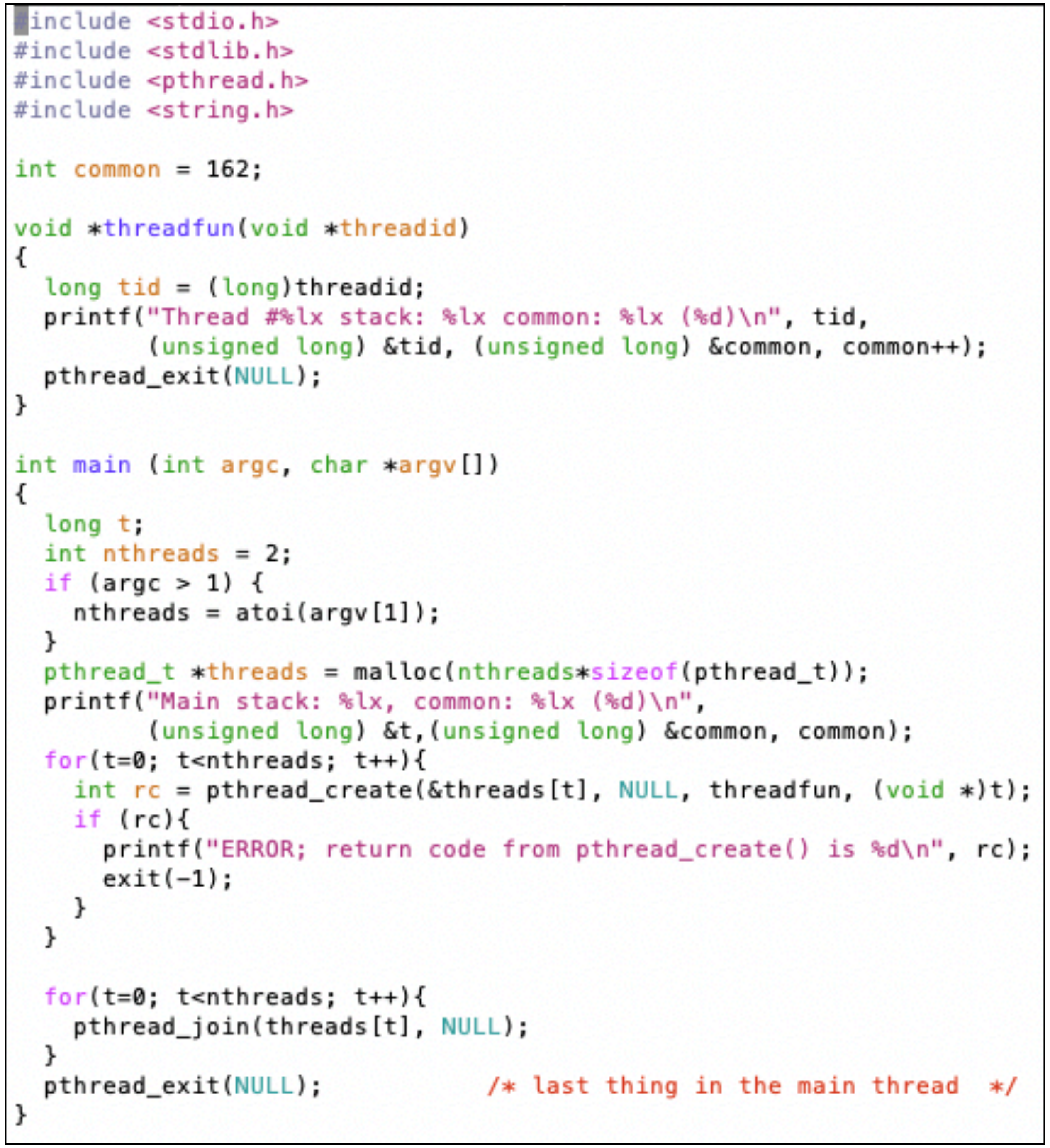
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Check your understanding



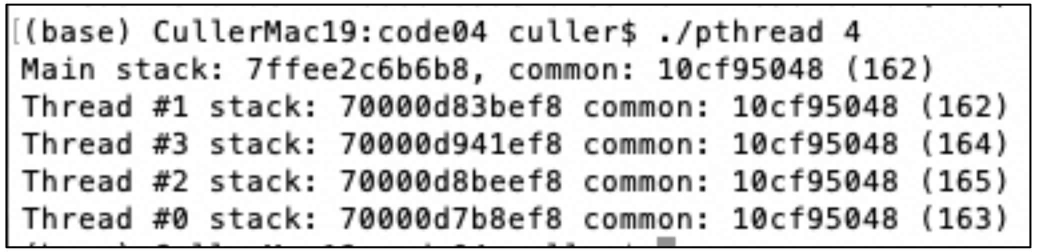
```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program?
2. Does the main thread join with the threads in the same order that they were created?
3. Do the threads exit in the same order they were created?
4. If we run the program again, would the result change?



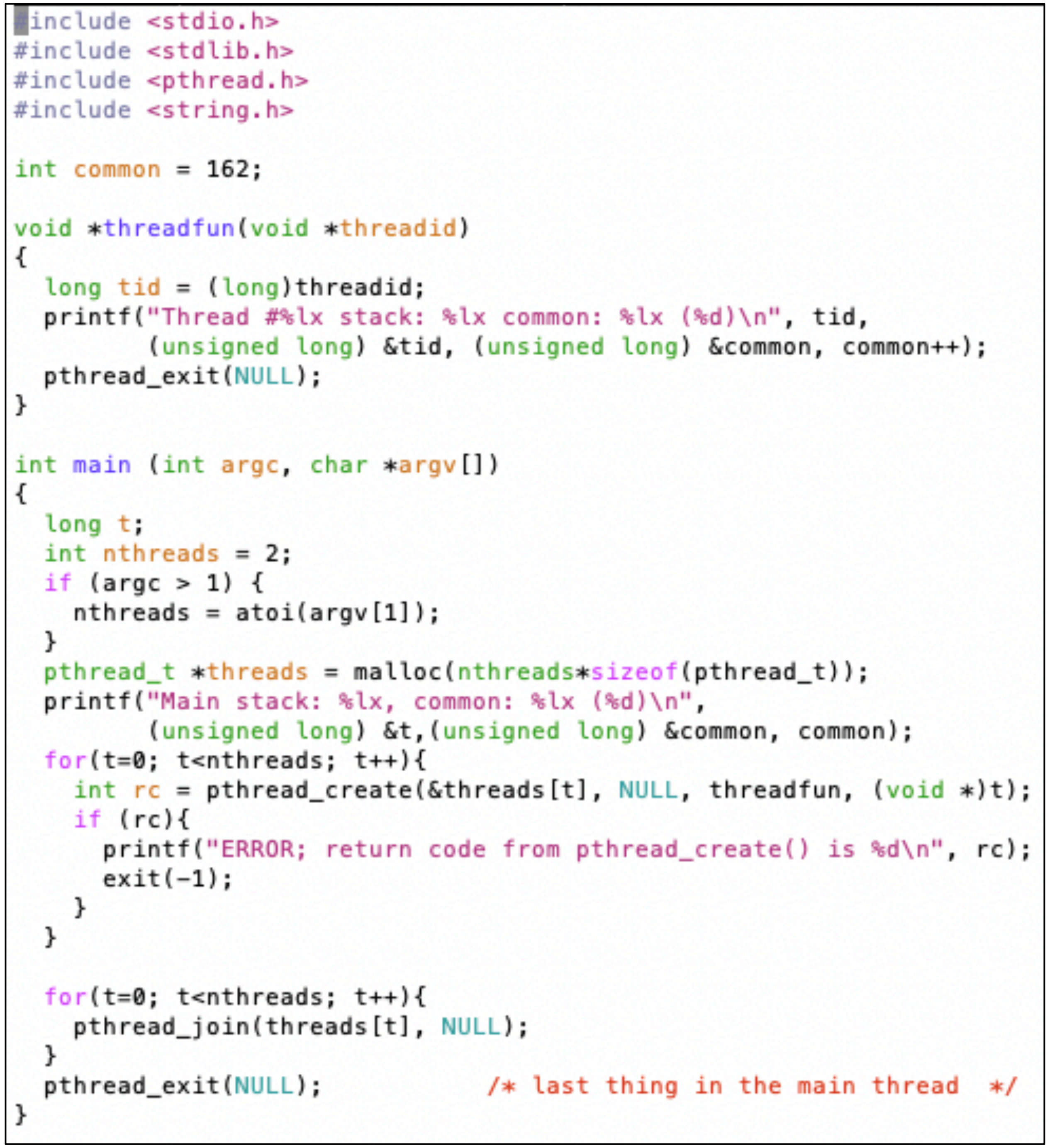
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Check your understanding



```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program? **Five**
2. Does the main thread join with the threads in the same order that they were created? **Yes**
3. Do the threads exit in the same order they were created? **Maybe??**
4. If we run the program again, would the result change? **Possibly!**



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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```