Lecture 2: Processes and Threads

CS343 – Operating Systems

Branden Ghena – Fall 2020

Some slides borrowed from:
Stephen Tarzia (Northwestern), Jaswinder Pal Singh (Princeton), Harsha Madhyastha (Michigan), and UC Berkeley CS61C and CS162

Northwestern

Today's Goals

- Understand the operating system's view of a process.
- How does a process communicate with the OS?
- Explore a few process creation system calls.
- What are threads and why are they useful?

Outline

- Processes
- System Calls
- Process Creation Calls
- Threads

Outline

- **Processes**
- System Calls
- Process Creation Calls
- Threads

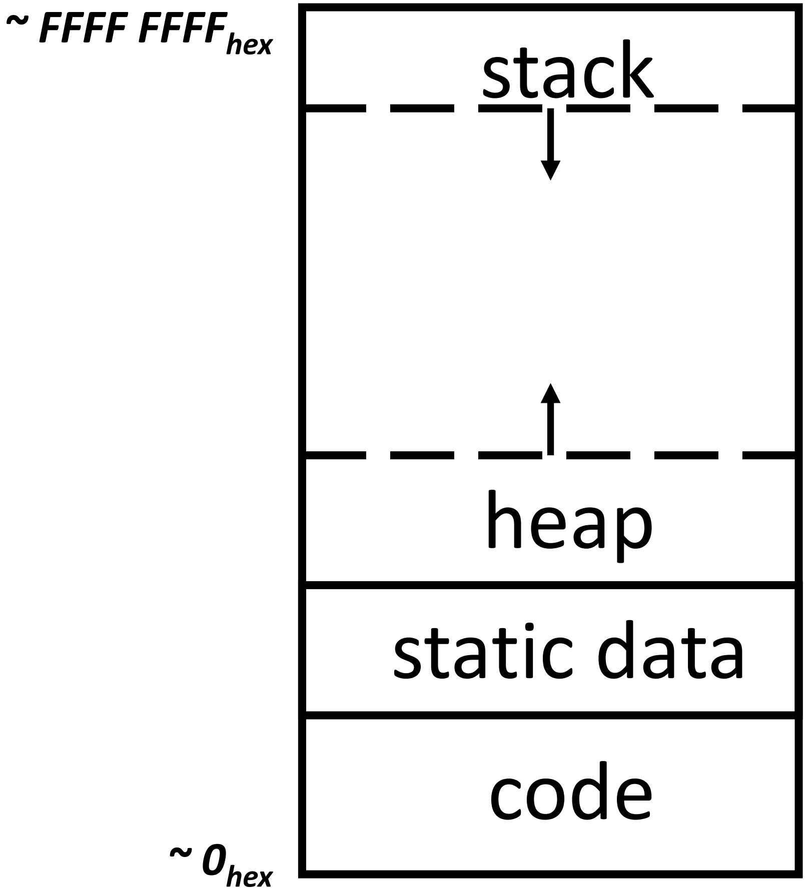
Definitions

- Program
  - Code (instructions + data)

- Process
  - A program in execution
  - Program code, execution context, one or more threads

Process Contents

- Address Space



- Registers (x86-64 pictured)

%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

- Program Counter PC

This is all an illusion of course…

POSIX processes have file descriptors

- Integers specifying a file the process is interacting with
  - Process contains a table linking integers to files (and permissions)

- Default file descriptors
  - 0 - Standard input (stdin)
  - 1 - Standard output (stdout)
  - 2 - Standard error (stderr)

- Function calls to interact with files
  - `int      open  (const char *path, int oflag, ... );`
  - `ssize_t read  (int fildes, void *buf, size_t nbyte);`
  - `ssize_t write (int fildes, const void *buf, size_t nbyte);`

Example file descriptors

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd    DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd    DIR    8,5     4096       2 /
vim     6447 brghena  txt    REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem    REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem    REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem    REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem    REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem    REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem    REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem    REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem    REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem    REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem    REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem    REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem    REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem    REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem    REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem    REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem    REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem    REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem    REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem    REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem    REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem    REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem    REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena   0u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   1u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   2u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   4u    REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Also all of the code in the address space

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd    DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd    DIR    8,5     4096       2 /
vim     6447 brghena  txt    REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem    REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem    REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem    REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem    REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem    REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem    REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem    REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem    REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem    REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem    REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem    REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem    REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem    REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem    REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem    REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem    REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem    REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem    REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem    REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem    REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem    REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem    REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena   0u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   1u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   2u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   4u    REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Additional Process Contents

- Whatever else the OS thinks is useful
  - Process ID
  - Priority
  - Time Used
  - Process State

Check your understanding

- Is it safe for two processes to have the same code section?

Check your understanding

- Is it safe for two processes to have the same code section?

Usually yes!

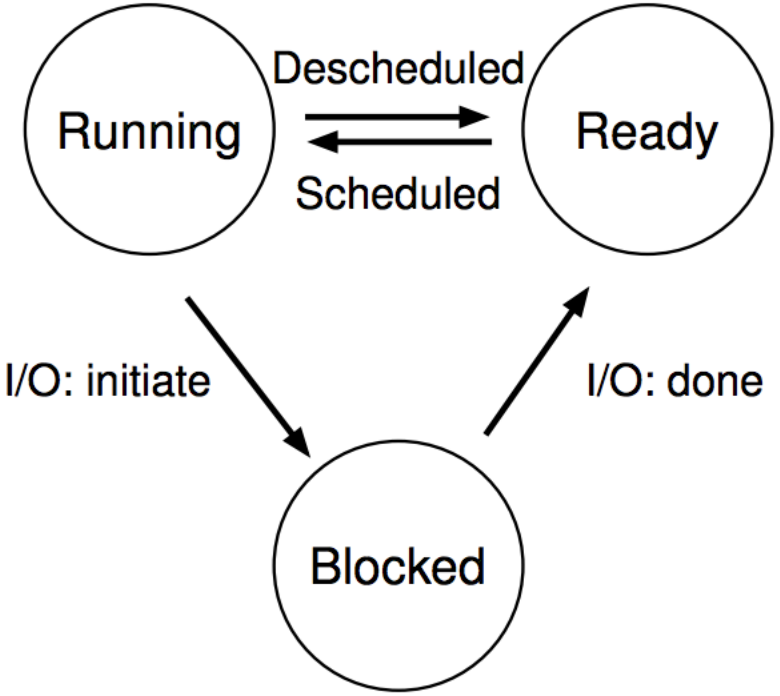
The code section is marked read-only.

Multiple instances of a terminal all share the same code.

Self-modifying code would be a problem…

Processes don't run all the time

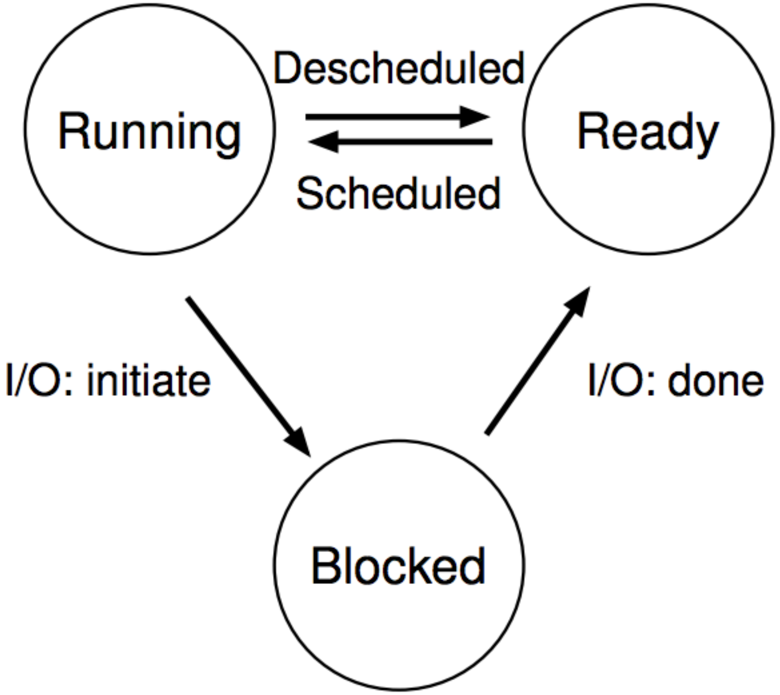
The three basic process states:



- 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**, telling the OS to let someone else 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)

- 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

Outline

- Processes
- **System Calls**
- Process Creation Calls
- Threads

Things a program cannot do itself

- Print "hello world"
  - *because the display is a shared resource.*
- Download a web page
  - *because the network card is a shared resource.*
- Save or read a file
  - *because the filesystem is a shared resource, and the OS wants to check file permissions first.*
- Launch another program
  - *because processes are managed by the OS*
- Send data to another program
  - *because each program runs in isolation, one at a time*

How does a process ask the OS to do something?

- Certain things can only be accessed from kernel mode
  - All of memory, I/O devices, etc.

- **Bad Idea** to allow processes to switch into kernel mode
  - We do NOT trust processes

- Requirements
  1. Switch execution to the kernel
  2. Change into kernel mode
  3. Inform the kernel what you want it to do

Hardware can save us!

- Solution: hardware instruction – trap
  - Also known as exception or fault

- When instruction runs:
  1. PC is moved to a known location in the kernel
  2. Mode is changed to kernel mode

- Same mechanism is used for other exceptions
  - Division by zero, invalid memory access
  - Also very similar to hardware interrupts

System call steps (simplification)

1. Process loads parameters into registers (just like a function call)
2. Process executes trap instruction (`int, syscall, svc`, etc.)
3. Hardware changes PC to “handler” and switches to kernel mode
4. OS checks what the process wants to do from registers
5. OS decides *whether* the process is allowed to do so
6. OS sets process state to blocked

Returning from a system call (simplification)

- After OS finishes whatever operation it was asked to do
  - And when the process is scheduled to run again

1. OS places return result in a register (just like a function call)
2. OS sets process state to running
3. OS changes mode to user mode (and sets virtual memory stuff)
4. OS sets PC to instruction after the system call

Check your understanding

- After OS finishes whatever operation it was asked to do
  - And when the process is scheduled to run again

1. OS places return result in a register (just like a function call)
2. OS sets process state to running
3. OS changes mode to user mode (and sets virtual memory stuff)
4. OS sets PC to instruction after the system call

- Why doesn’t the OS need a special instruction to change mode and run the process?

Returning from a system call (simplification)

- After OS finishes whatever operation it was asked to do
  - And when the process is scheduled to run again

1. OS places return result in a register (just like a function call)
2. OS sets process state to running
3. OS changes mode to user mode (and sets virtual memory stuff)
4. OS sets PC to instruction after the system call

- Why doesn't the OS need a special instruction to change mode and run the process?
  - **It has privilege to change mode and is trusted to start the process**

System calls trigger *context switches*

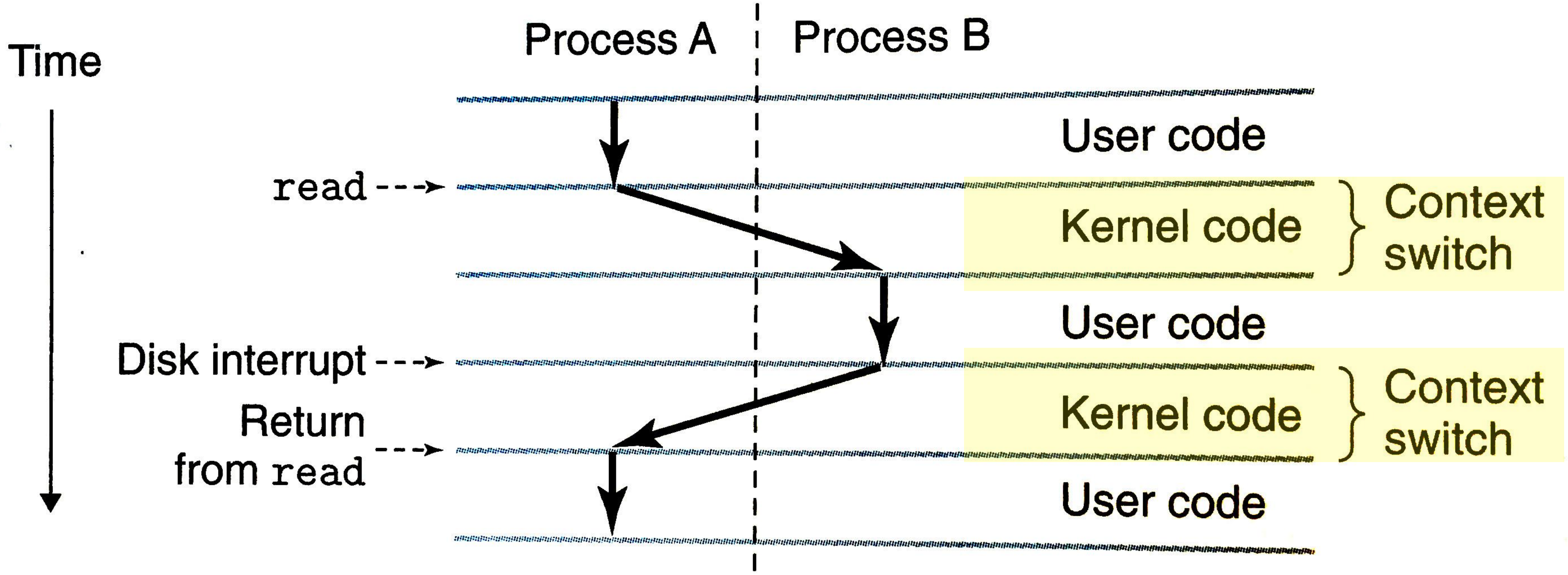


Diagram from Bryant & O'Hallaron book

Example Linux system calls

- https://man7.org/linux/man-pages/man2/syscalls.2.html

- Managing processes
  - Fork
  - Exec
  - Waitpid
  - Exit

- Managing files
  - Open
  - Close
  - Read
  - Write
  - Seek

Outline

- Processes
- System Calls
- **Process Creation Calls**
- Threads

Process system calls

```
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

```c
#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

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

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

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

← Existential crisis

Executing a new program

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

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

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

Creating your own shell

https://danishpraka.sh/2018/01/15/write-a-shell.html

```
void execute(char** args) {
  if (strcmp(args[0], "exit") == 0) {
    exit(); // exit the shell when requested
  }

  pid_t cpid = fork();
  if (cpid == 0) {
    if (execvp(args[0], args) < 0) { // child, execute new process
      printf("command not found: %s\n", args[0]);
    }
  } else {
    waitpid(cpid, & status, WUNTRACED); // parent, wait for process to be complete
}}

int main(){
  char** args;
  while(1){
    printf("> ");
    args = parse_incoming_text(); // complicated in C unfortunately
    execute(args);
}}
```

Creating your own shell

https://danishpraka.sh/2018/01/15/write-a-shell.html

```
void execute(char** args) {
  if (strcmp(args[0], "exit") == 0) {
    exit(); // exit the shell when requested
  }

  pid_t cpid = fork();
  if (cpid == 0) {
    if (execvp(args[0], args) < 0) { // child, execute new process
      printf("command not found: %s\n", args[0]);
    }
  } else {
    waitpid(cpid, & status, WUNTRACED); // parent, wait for process to be complete
}}

int main(){
  char** args;
  while(1){
    printf("> ");
    args = parse_incoming_text(); // complicated in C unfortunately
    execute(args);
}}
```

Many other system calls

- POSIX contains many others, for example time()
  - And especially lots of old ones
- Windows or other operating systems will have entirely different system call infrastructures

Outline

- Processes
- System Calls
- Process Creation Calls
- **Threads**

Software Tasks: Threads

Unit of execution *within* a process

Processes discussed so far have a single thread

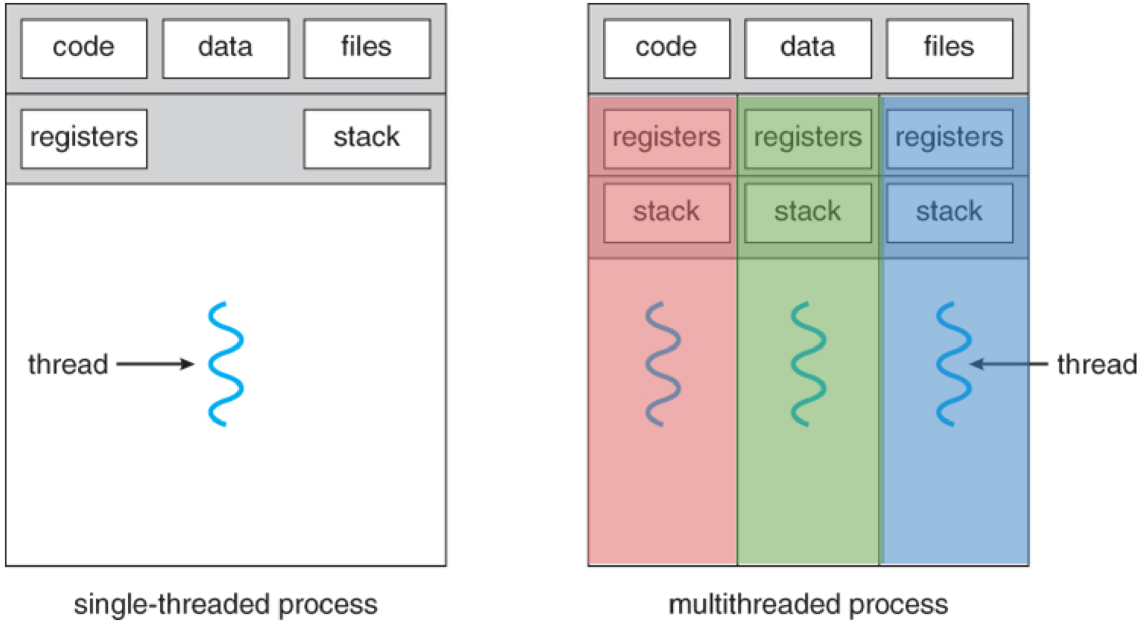
- They "have a single thread of execution"
- They "are single-threaded"

But a single process could have multiple threads

Thread Memory

Threads have separate:

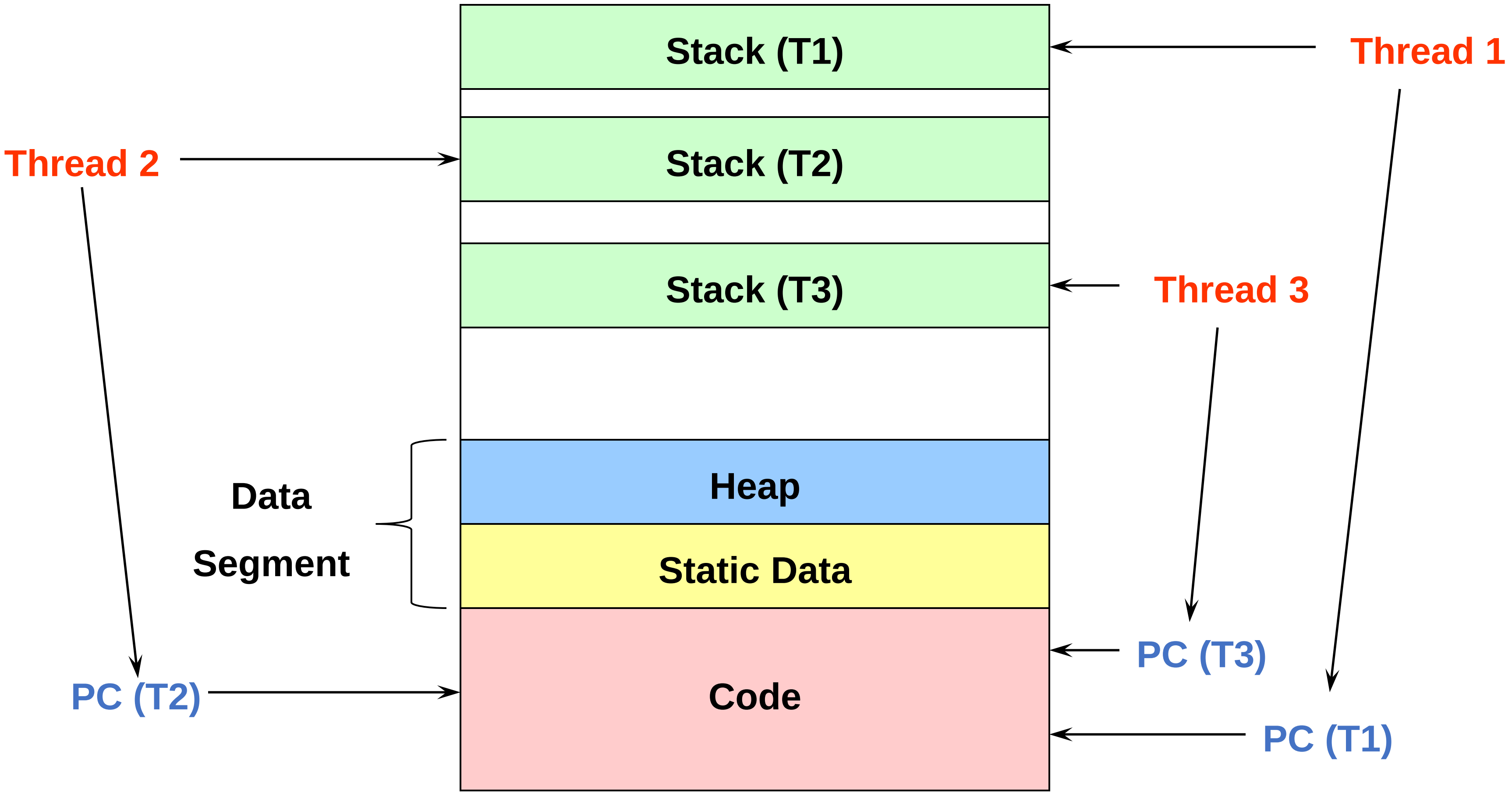
- PC
- Registers
- Stack memory



Threads share:

- Code memory
- Global variables (static memory)
- File descriptors

Process address space with threads



Thread use case: web browser

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

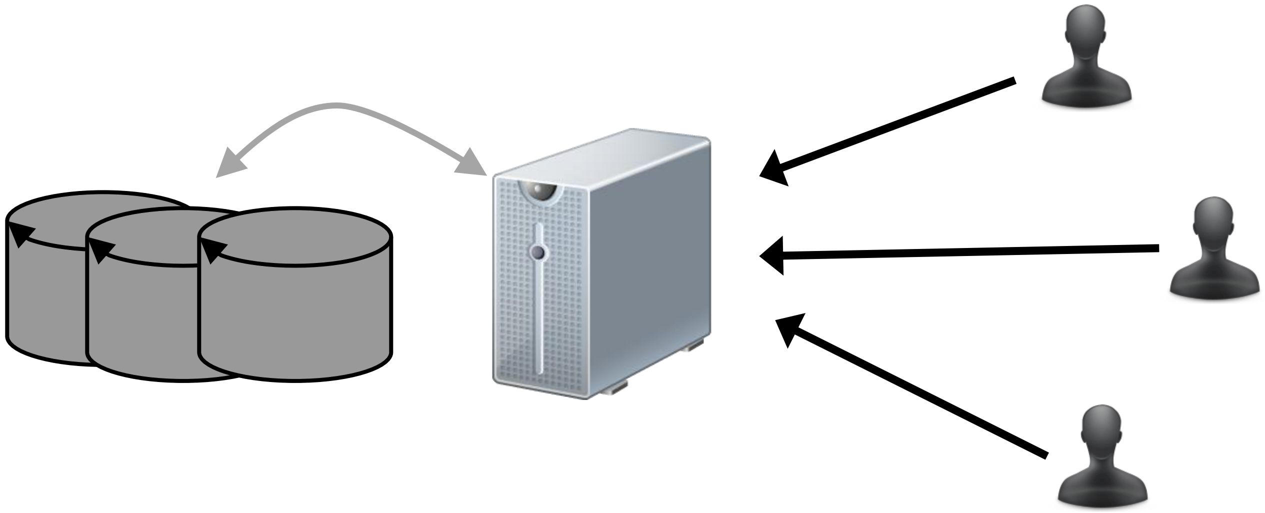
You want a tab for each web page you open:

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

Disclaimer: Actually, browsers use separate processes for each tab for a variety of reasons including performance and security

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

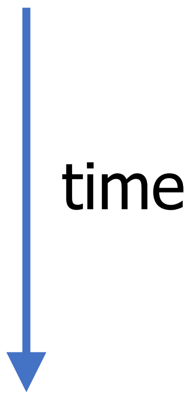
time

- 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



- 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.

Main Thread	**Thread 1**	**Thread 2**
Request 1 arrives		
Create thread →		
	Read in request 1	
	Start disk I/O	
Request 2 arrives		
Create thread →		
		Read in request 2
		Start disk I/O
	Disk I/O finishes	
	Respond to request 1	
	Exit	

time ↓

- 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

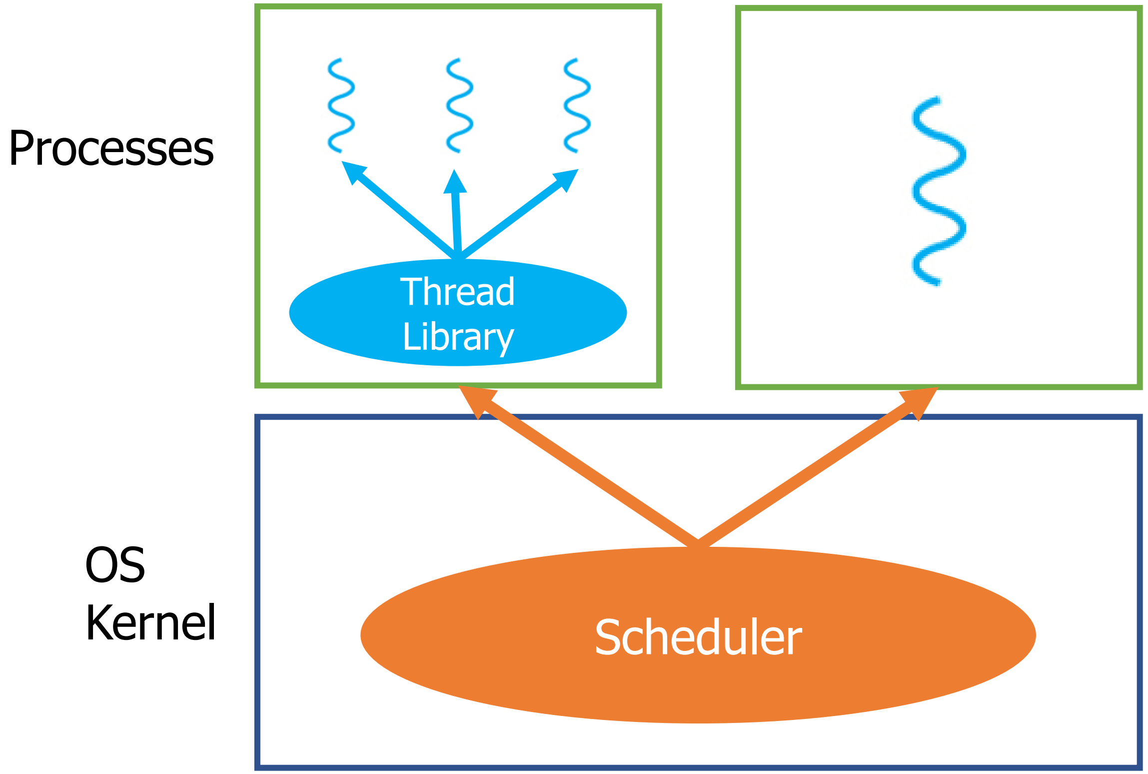
Handle I/O in separate thread, avoid blocking other progress

Models for thread libraries: user threads

- Thread scheduling is implemented within the process
  - OS only knows about the process, not the threads

- Upsides
  - Works on any hardware or OS
  - Performance is better when creating and switching

- Downsides
  - A system call in any thread **blocks all threads**



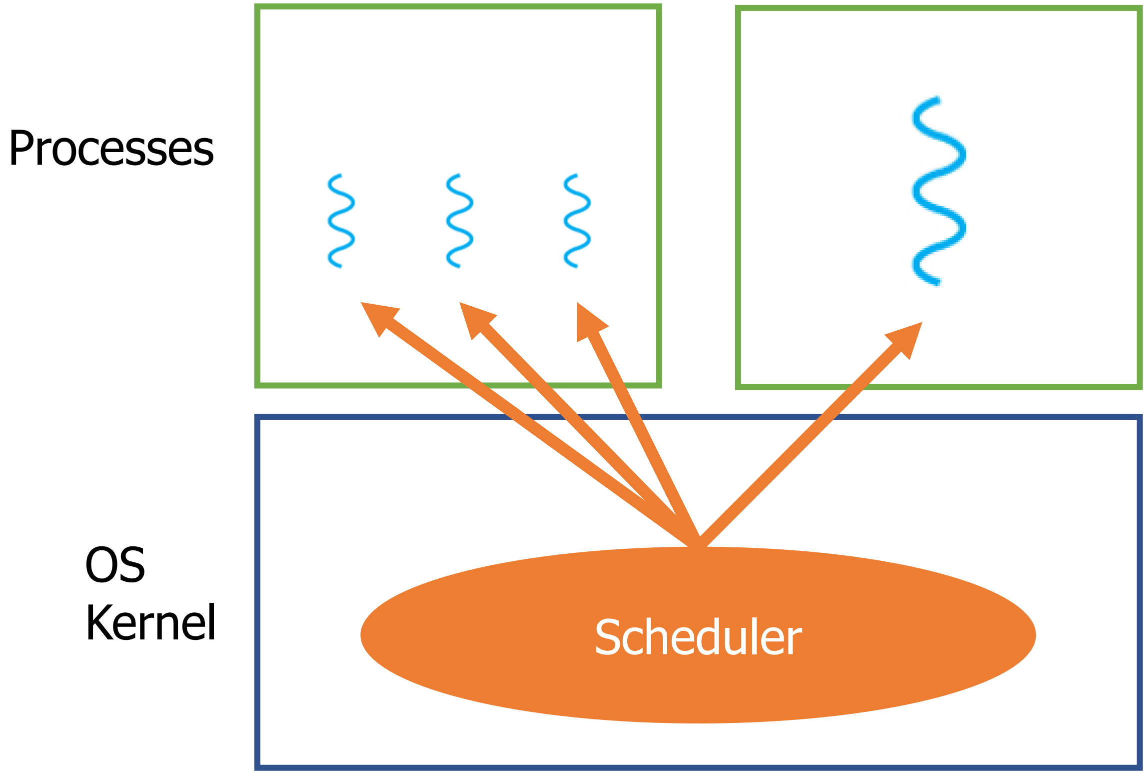
Models for thread libraries: kernel threads

- 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

- Hybrid models are possible



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*.

Pthread system call example

- What happens when pthread_create() is called in a process?

Library:

```
int pthread_create(…) {
   Do some work like a normal function

   asm code … syscall # into %eax
   put args into registers %ebx, …
   special trap instruction
```

Linux uses the clone() syscall to do this

Kernel:

```
   get args from regs
   do the work to spawn the new thread
   store return value in %eax
```

```
   get return values from regs
   Do some more work like a normal function
};
```

Threads Example

```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

- 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  */
}
```

Outline

- Processes
- System Calls
- Process Creation Calls
- Threads