Agenda

Threads

Application design with threads

Classical thread model

POSIX threads

Implementing threads

- User Space
- Kernal Space

Implementing a user space thread library using `ucontext`

Threads

A **thread** is a stream of execution within a process

Idea: Allow processes to have multiple streams of execution

- Can share address space, files, pipes, and other resources
- Easier to create and destroy than processes (10-100 times faster than creating processes)
- Performance gain, even on a single CPU, when threads have frequent IO that leads to blocking
  - Modern CPUs have multiple cores -> true concurrency!

Multi-process vs multi-threaded

Multi-process (fork())

- Each process has its own memory
- Sharing is harder and expensive
- Programmer must decide how to split work and coordinate processes
  - External mechanisms: shmem, pipes, sockets, wait, signals
- Concurrent code has no guarantees on the order it runs
- Slower creation/deletion (more overhead)
- Can split work across machines

Multi-threaded (pthread_create)

- Each thread shares memory in same process
- Sharing is easy and cheap
- Programmer must decide how to split work and coordinate threads
  - Built-in mechanisms: mutex, barrier, wait
- Concurrent code has no guarantees on the order it runs
- Fast creation/deletion (more lightweight)
- Must run on the same machine

Applications of threads: Parallel Processing

Speed up a program by dividing it up into parts, each part runs in parallel on multiple CPUs

++ *10 hours on 1 CPU ➔ 6 mins on 100 CPUs!!! (maybe)*

Examples:

- Split rendering of a large image into subsections, each subsection computed by its own thread
- Split searching a file system into subgroups of files, each subgroup search by its own thread

Applications of threads: Word Processor

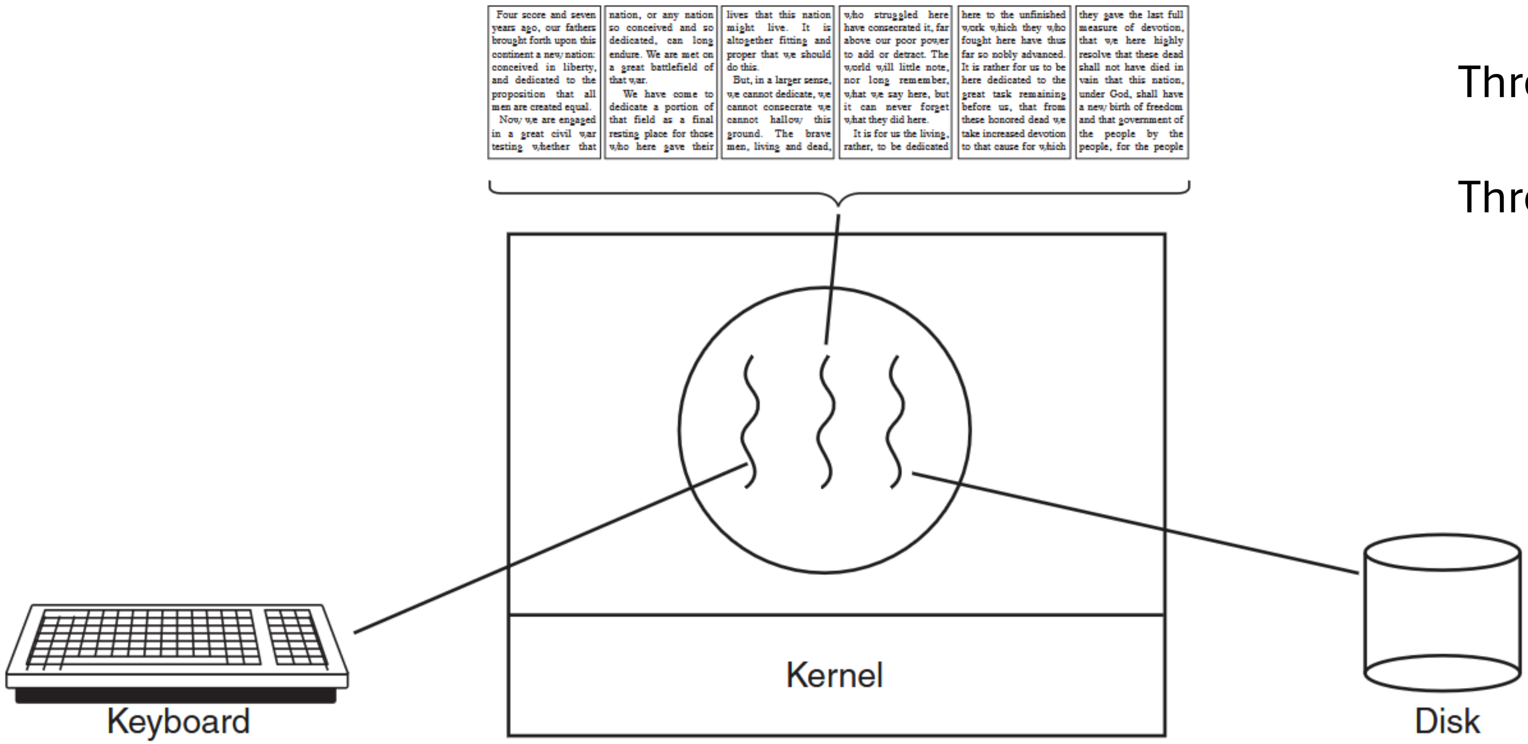


Figure 2-7. A word processor with three threads.

Thread 1: User Interface

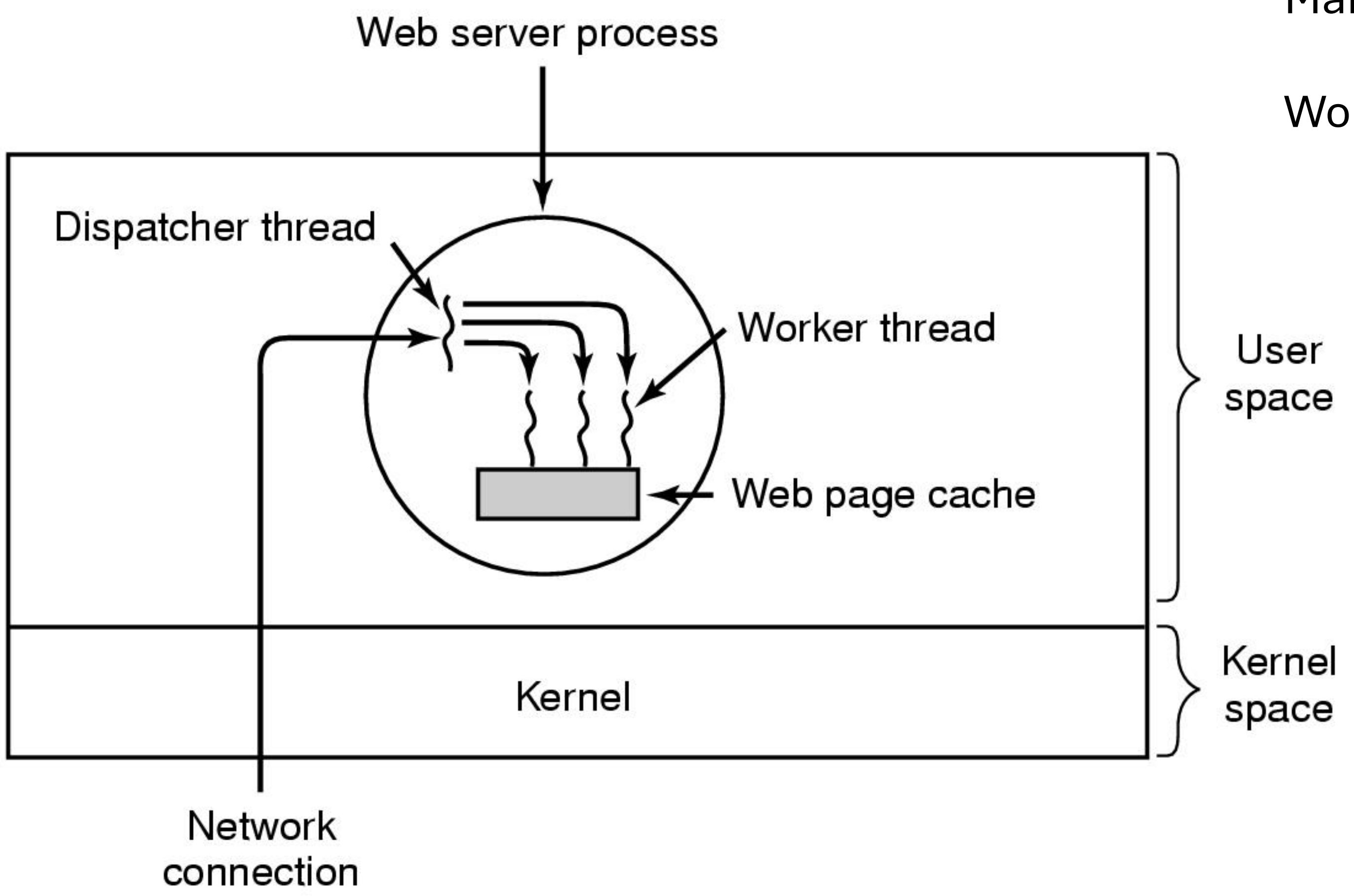
Thread 2: Formatting

Thread 3: Saving backups

Discuss: What if we only had one thread?

What if we tried to implement this with multiple processes?

Applications of threads: Web Server



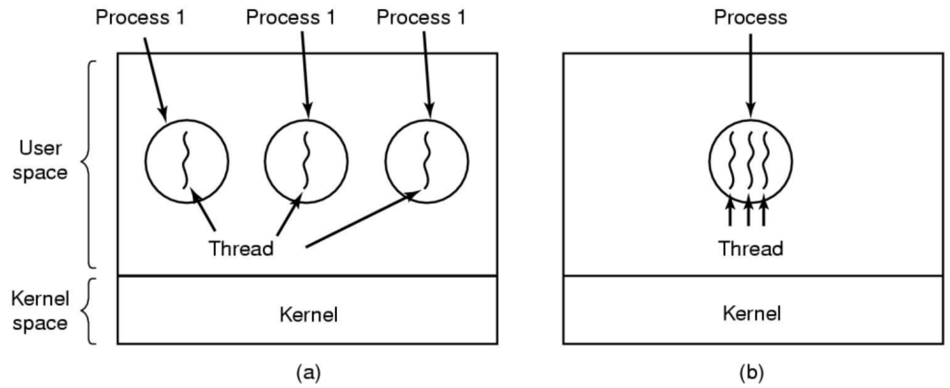
Main Thread: Dispatcher (waits for requests)

Worker Thread : Created for each request

Discuss: What if we only had one thread?

What if we tried to implement this with multiple processes?

Classical Thread Model



Multiprocess   Multithreaded

A process groups related resources together

open files, signal handlers, child processes, alarms, and more

A thread is a stream of execution. It has its own

- Program counter
- Registers (hold current working variables)
- Stack (currently executing functions)
- Errno
- Thread ID

Threads can be scheduled to run on the CPU

All processes have at least one thread, called the **main thread**

Like processes, threads can be in one of several states: **running, blocked, ready, or terminated**

Single-threaded Process

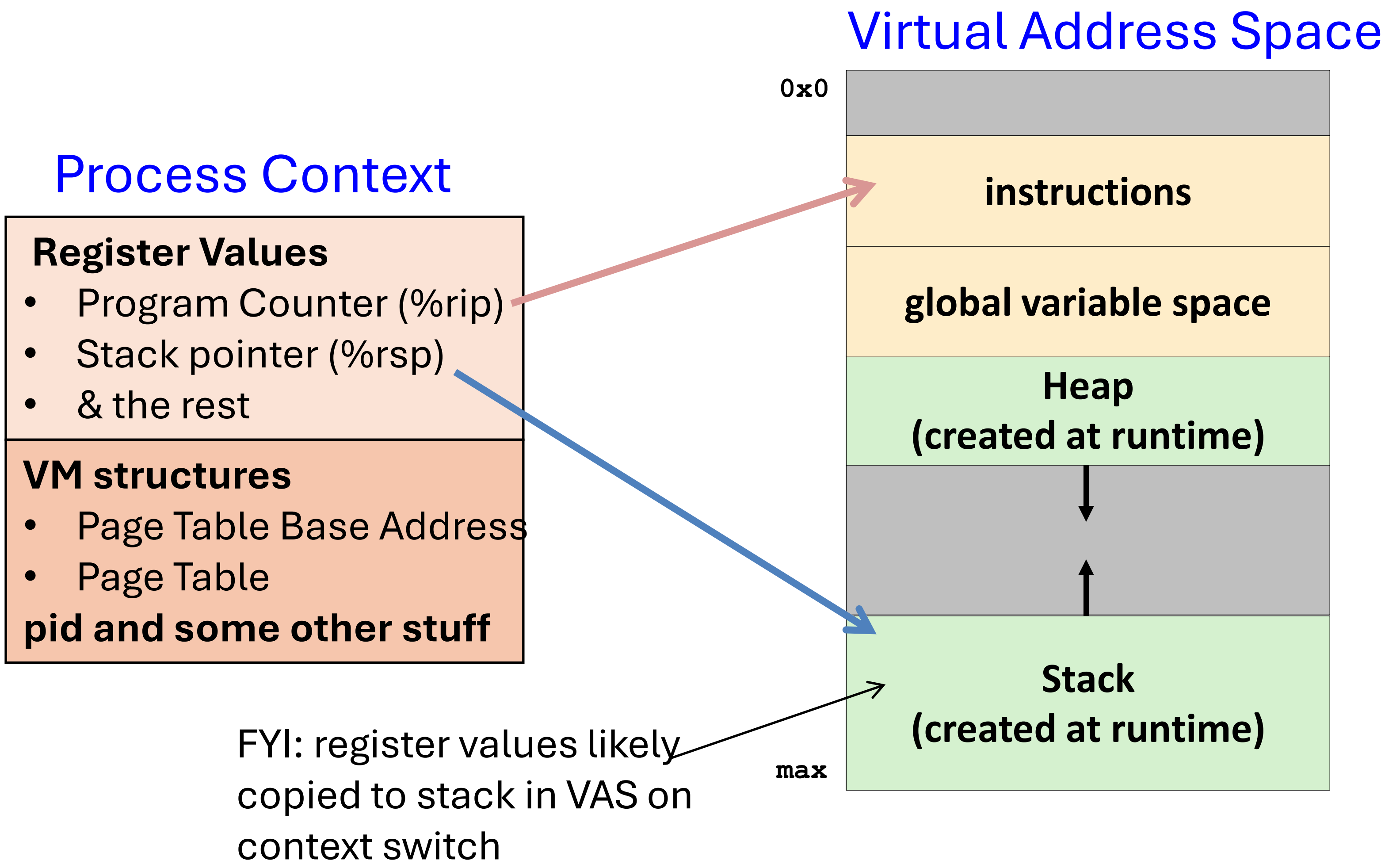


Diagram by Tia Newhall and Andy Danner

Multi-Threaded Process

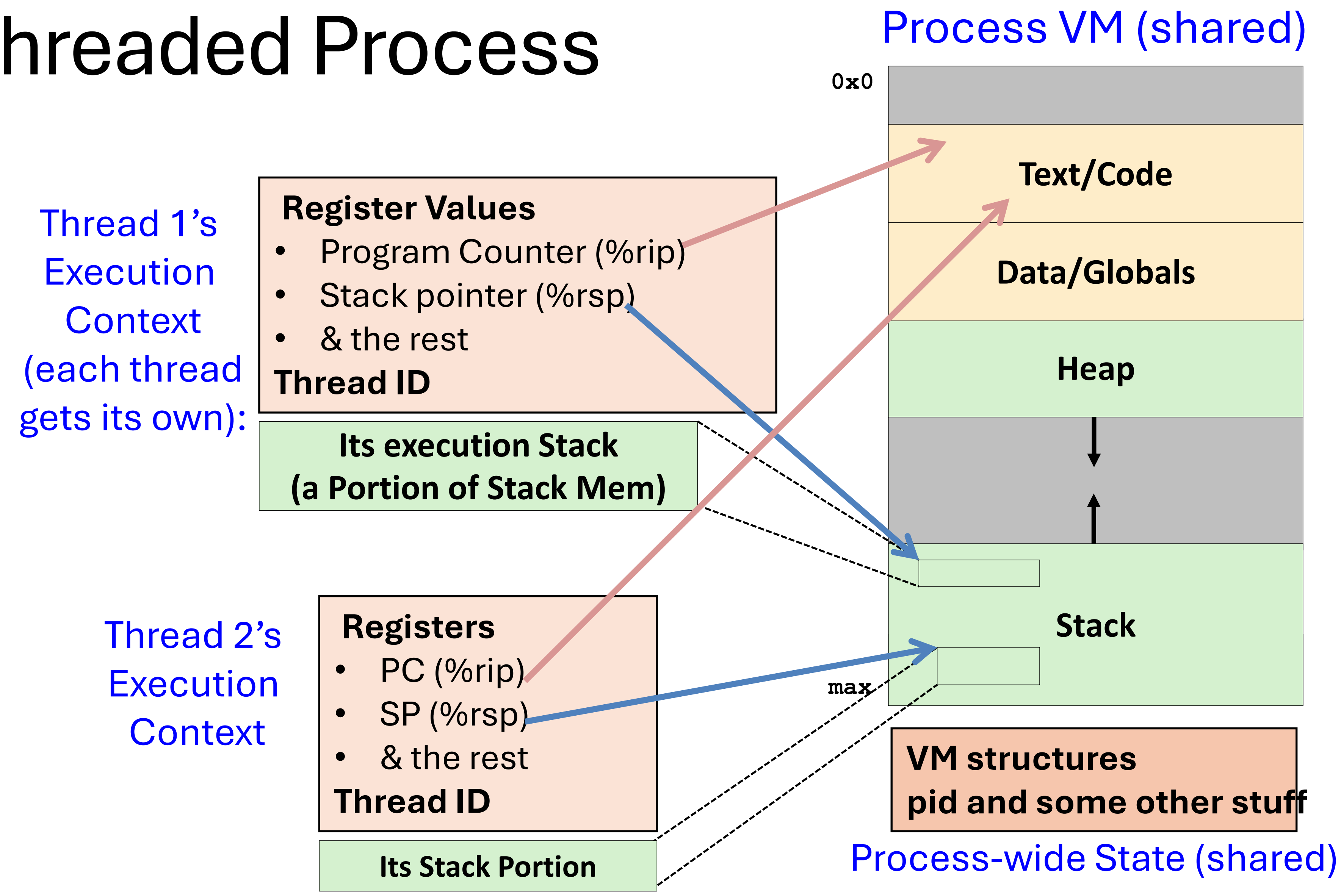


Diagram by Tia Newhall and Andy Danner

POSIX Threads (pthreads)

The **P**ortable **O**perating **S**ystem **I**nterface for UNI**X**

IEEE standard for working with threads (API)

To compile: `g++ myprog.c -lpthread`

Defines functions for

- Creating threads (**pthread_create**)
- Configuring thread priority (**pthread_attr_init**, **pthread_attr_destroy**)
- Wait for a thread to finish (**pthread_join**)
- Terminate the calling thread (**pthread_exit**)
- Let another thread run (**pthread_yield**)
  - Depending on implementation, threads may not yield automatically (like processes). On systems where threads are scheduled with time slices, yield is a suggestion to the scheduler

What are the analogous process functions?

Review: What is void*?

Demo: HelloThread

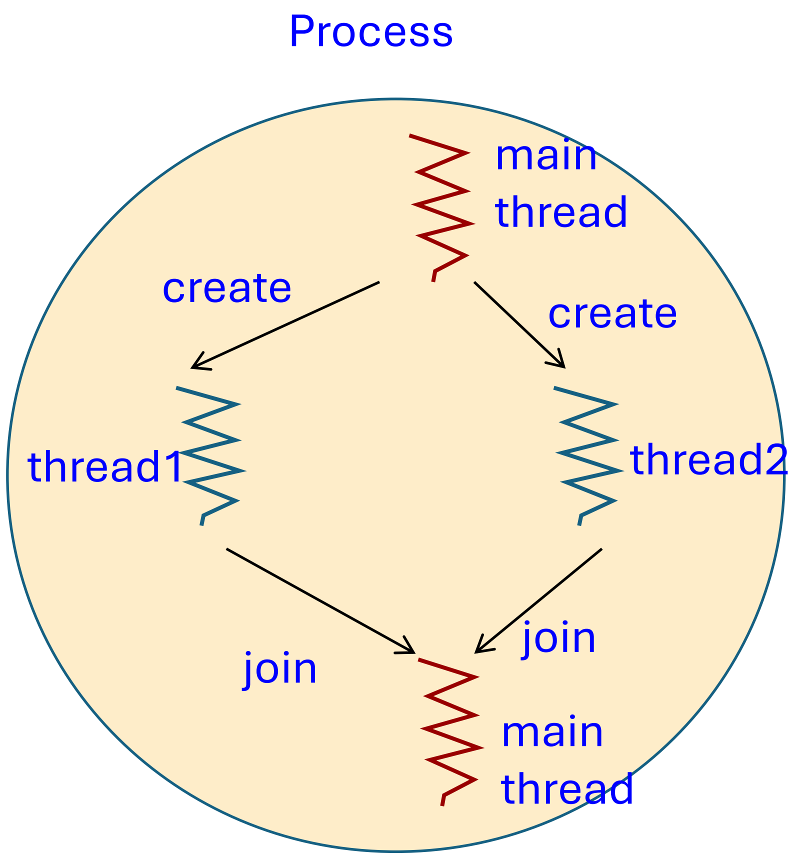
```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1, id2 = 2;
  long* retval1 = NULL, *retval2 = NULL;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, HelloWorld, &id1);
  pthread_create(&thread2, NULL, HelloWorld, &id2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

```
g++ thread-hello.c -lpthread
```

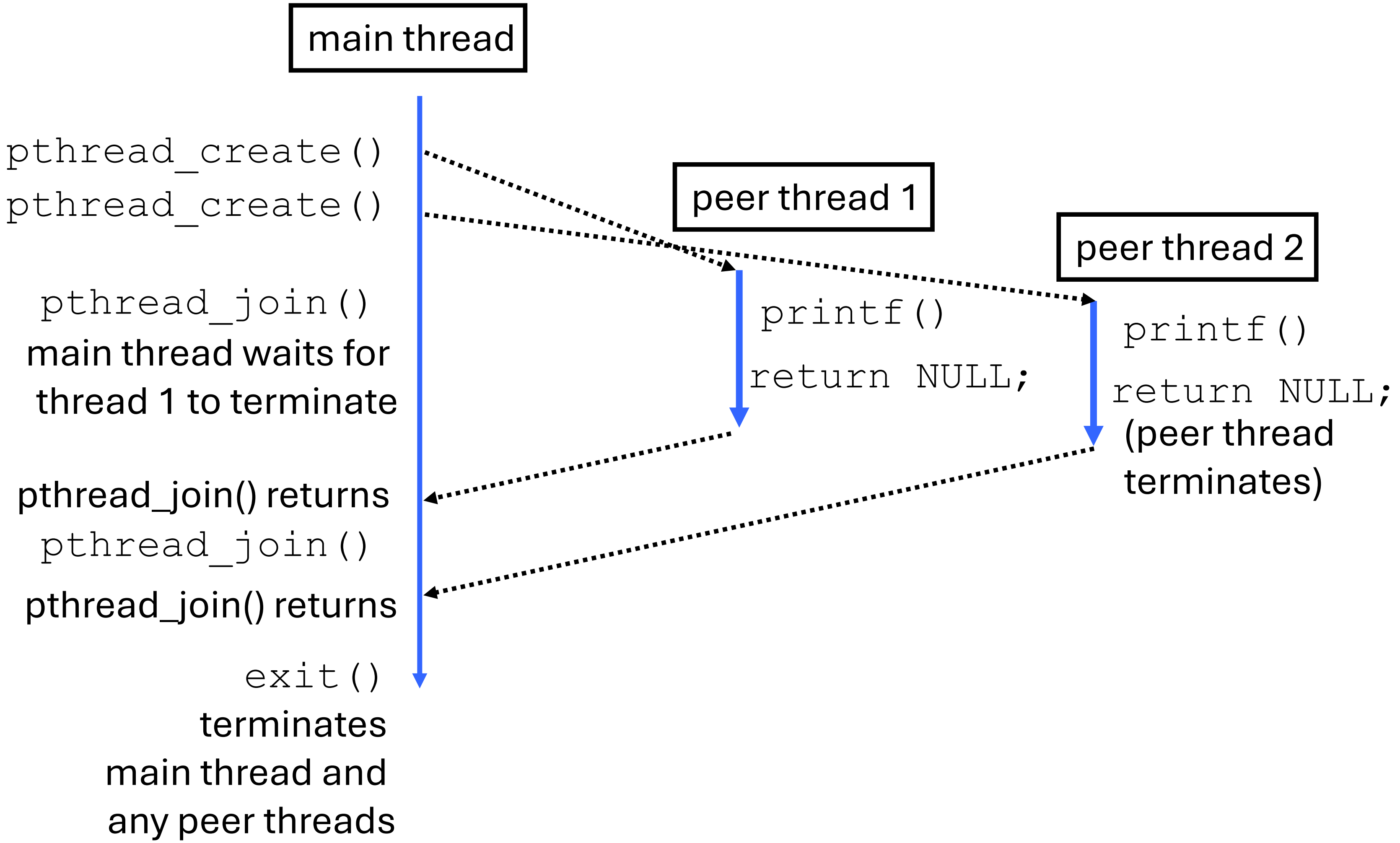
Visualizing HelloThread

Which lines of code are executed by which thread?

- main thread executes lines 12-16
- each thread executes lines 6-8
- main thread waits for thread 1 (line 17)
- main thread waits for thread 2 (line 18)
- main thread returns and exits (line 19)



Visualizing HelloThread: concurrent execution



Visualizing process execution: Hello

```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1;
  HelloWorld(&id1);
  return 0;
}
```

NOTE: A single-threaded application has one thread, called the *main thread*

Visualizing Thread execution: HelloThread

```c
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1, id2 = 2;
  long* retval1 = NULL, retval2 = NULL;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, HelloWorld, &id1);
  pthread_create(&thread2, NULL, HelloWorld, &id2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

Each thread gets its own stack but shares data/code

Visualizing process execution: HelloProcess

```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1;
  if (fork() == 0) {
    id1 = 1;
    HelloWorld(&id1);
  }
  else {
    id1 = 2;
    HellowWorld(&id1);
  }
  return 0;
}
```

Comparing threads and processes: memory

```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  pthread_t thread1, thread2;

  long* id = (long*) malloc(sizeof(long));
  *id = 12;
  pthread_create(&thread1, NULL, HelloWorld, id);
  pthread_create(&thread2, NULL, HelloWorld, id);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  free(id);
  return 0;
}
```

Comparing threads and processes: memory

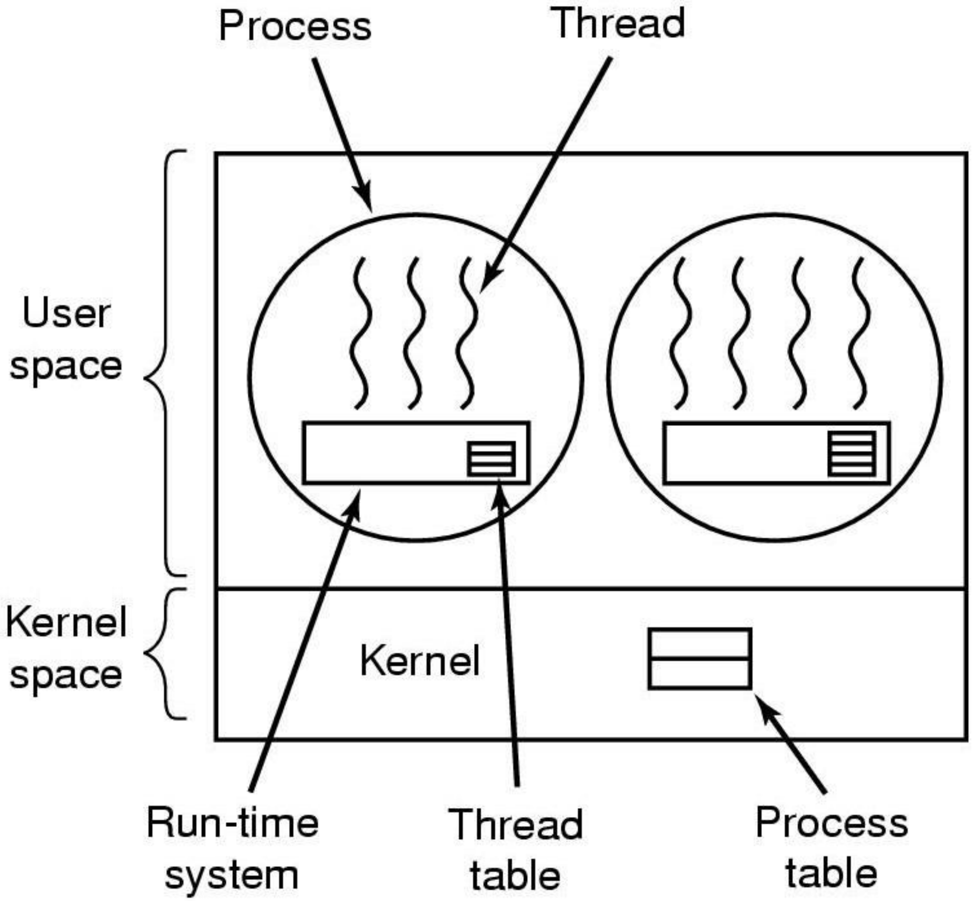
```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long* id = (long*) malloc(sizeof(long));
  if (fork() == 0) {
    *id = 2;
    HelloWorld(id);
  }
  else {
    *id = 1;
    HelloWorld(id);
  }
  free(id);
  return 0;
}
```

Implementing Threads

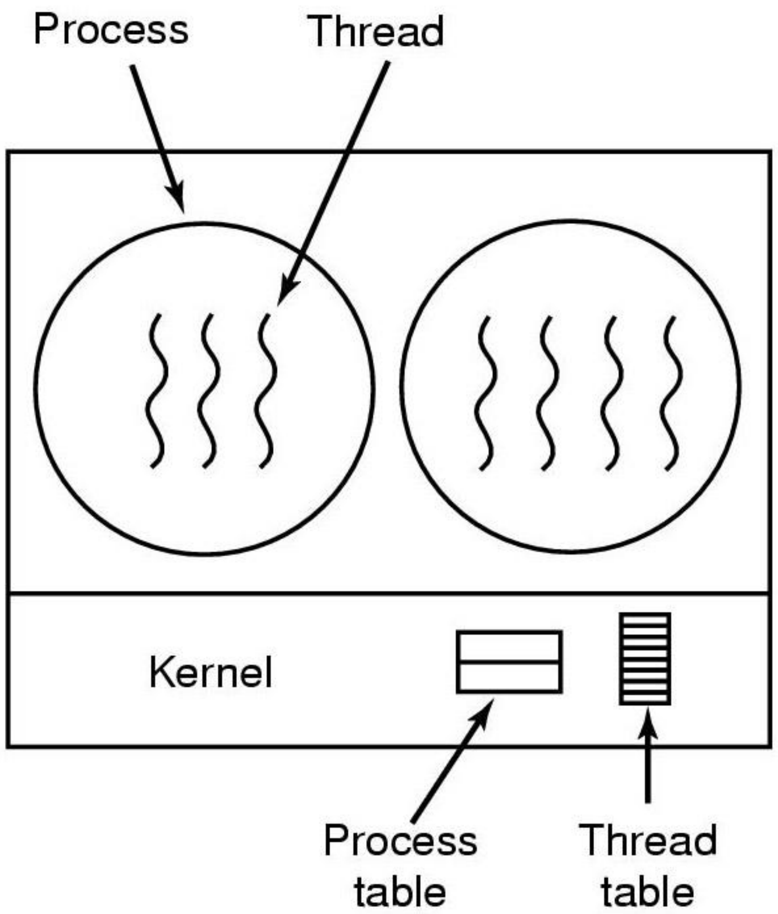
User space: threads are implemented in user space. The kernel is not aware of threads at all – the process handles all thread creation/deletion/scheduling

Kernel space: threads are implemented by the kernel

Implementing Threads in User Space



A user-level threads package



A threads package managed by the kernel

User-Level Threads

- The run-time support system for threads is entirely in user space.
- The threads run on top of a run-time system, which is a collection of procedures that manage threads.
  - e.g. a library with an API, like your next homework
- As far as the OS is concerned, it is a single (threaded) process.
- Threads can be implemented on an OS that does not support threads.
- Each process can have its own customized scheduling algorithm.

Kernel-supported Threads

- No run-time system is needed.
- For each process, the kernel has a table with one entry per thread, for thread's registers, state, priority, and other information.
- All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure.
- When a thread blocks, the kernel can run either another thread from the same process, or a thread from a different process.

User-level vs. Kernel-supported Threads

- If OS does not support threads, a library package in user space can do threads management

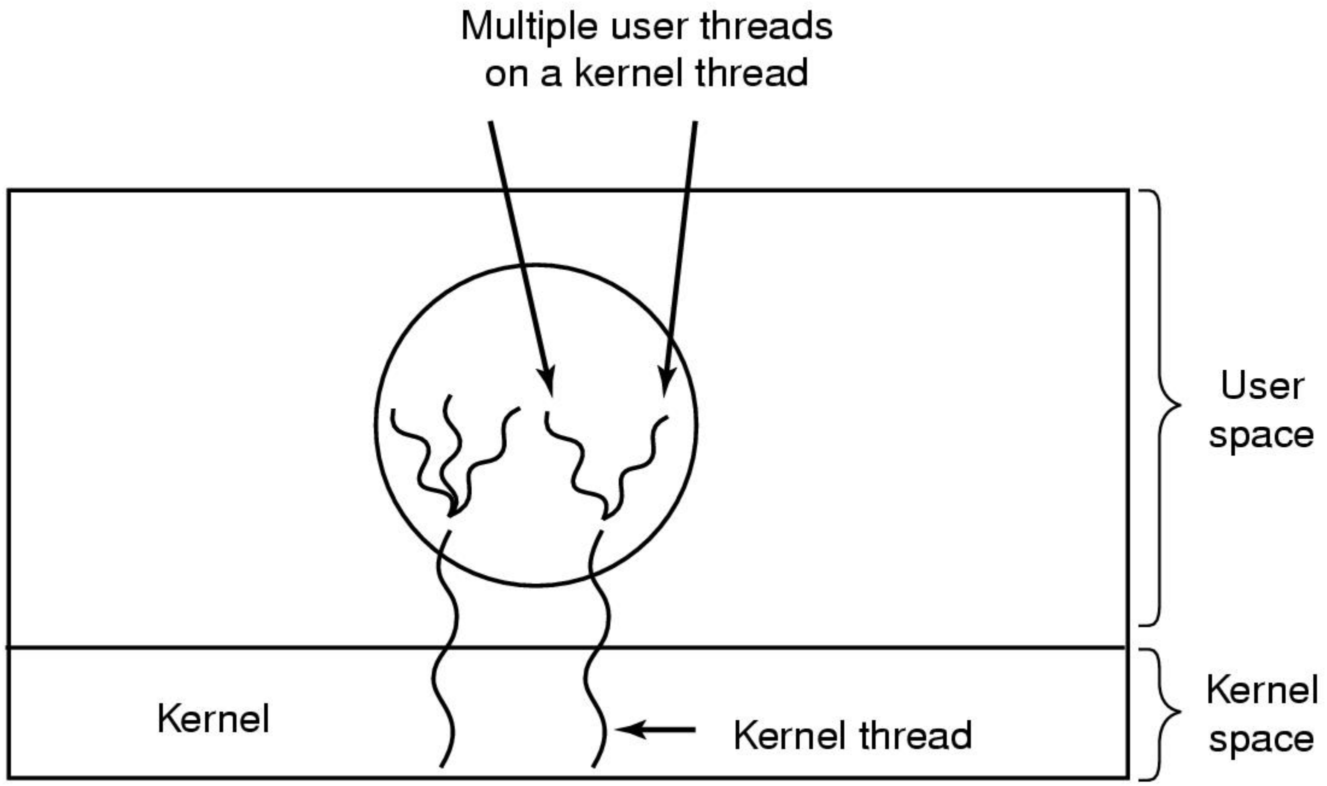
- What are the trade-offs for user-level vs kernel-level threads?
- Suppose we have the following:
  - Process A has one thread and Process B has 100 threads.
  - Scheduler allocates the time slices equally

User-level vs. Kernel-supported Threads

- User-level Thread:
  - A thread in process A runs 100 times as fast as a thread in process B.
  - One blocking system call blocks all threads in process B.
- Kernel-supported Threads:
  - Process B receives 100 times the CPU time than process A.
  - Switching among the thread is more time-consuming because the kernel must do the switch.
  - Process B could have 100 system calls in operation concurrently.

```
std::random_device dev; std::mt19937 rng(dev()); std::uniform_int_distribution<std::mt19937::result_type> dist6(1,6); // distribution in range [1, 6] std::cout << dist6(rng) << std::endl;
```

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

Leverage advantages of both user and kernel threads

User threads are mapped to kernel threads (1x1 on Linux). The kernel threads are special lightweight processes (LWP) that share resources but can be scheduled by the OS

Kernel can switch between threads in response to IO blocks detected by the kernel , even if they are in different processes

Exercise: multi-threaded max

Sketch a parallel algorithm that implements max on a very large vector

1. What data structures do we need?
2. What can be done in parallel?
3. What must be done serially?

Implementing a user space thread library

`ucontext` is a low-level mechanism for switching between functions

- Can be used to implement coroutines or lightweight threads (LWT)

Idea: Using ucontext we can interrupt the execution of a function at *anytime*, save its state, and then resume execution

A context corresponds to the “box” we draw in a stack diagram

- When we swapcontext, we take the state of one box and replace it with another
- When we setcontext, we replace the current box with a new one

For more information: *man 3 makecontext*

Example1: ucontext

```
void f(){
  printf("Hello World\n");
}
```

```
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = NULL;

  makecontext(&uc, f, 0); // associate context with f()
  setcontext(&uc); // “invokes" f(); doesn’t return on success
  perror("setcontext"); //setcontext() doesn’t return on success
  return 0;
}
```

Example 1: Visualizing its execution

```
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = NULL;

  makecontext(&uc, f, 0);
  setcontext(&uc);
  perror("setcontext");
  printf("boo\n"); // never prints
  return 0;
}
```

setcontext

```
void f(){
  printf("Hello World\n");
}
```

process ends when this function returns

main() and f() are on different stacks

Contexts

Every function context, represented by a ucontext structure, has the following components:

- its own stack
- its own set of signal flags
- registers

Contexts can be created, suspended, copied and destroyed

Calling functions on the same stack (e.g. the normal way)

```
void f(){
  printf("Hello World\n");
}
```

```
int main(int argc, char * argv[]){
  f();
  printf("Back in main\n");
  return 0;
}
```

Example 2: Calling functions on different stacks

```
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc, mainc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = &mainc;

  makecontext(&uc, f, 0); // associate context with f()
  int r = swapcontext(&mainc, &uc); // “invokes" f(); doesn’t return on success
  if (r) perror("setcontext");
  return 0;
}
```

```
void f(){
  printf("Hello World\n");
}
```

What’s different from example 1?

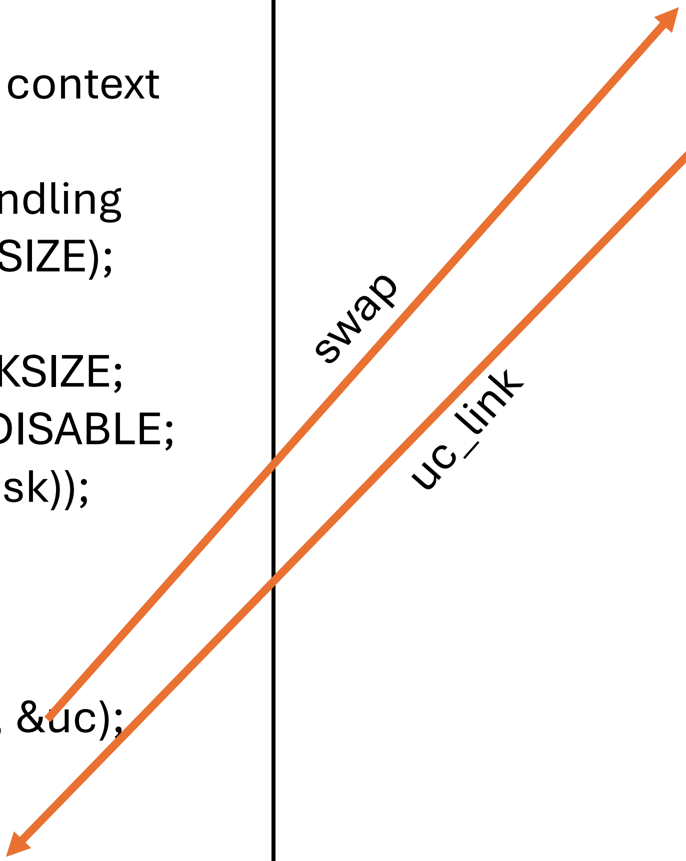
Example 2

```
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc, mainc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = &mainc;

  makecontext(&uc, f, 0);
  int r = swapcontext(&mainc, &uc);
  if (r) perror("setcontext");
  free(stack);
  return 0;
}
```

```
void f(){
  printf("Hello World\n");
}
```

Example 2: Visualizing its execution

```c
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc, mainc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = &mainc;
  makecontext(&uc, f, 0);

  int r = swapcontext(&mainc, &uc);
  if (r) perror("setcontext");
  free(stack);
  return 0;
}
```




```c
void f(){
  printf("Hello World\n");
}
```

main() and f() are on different stacks

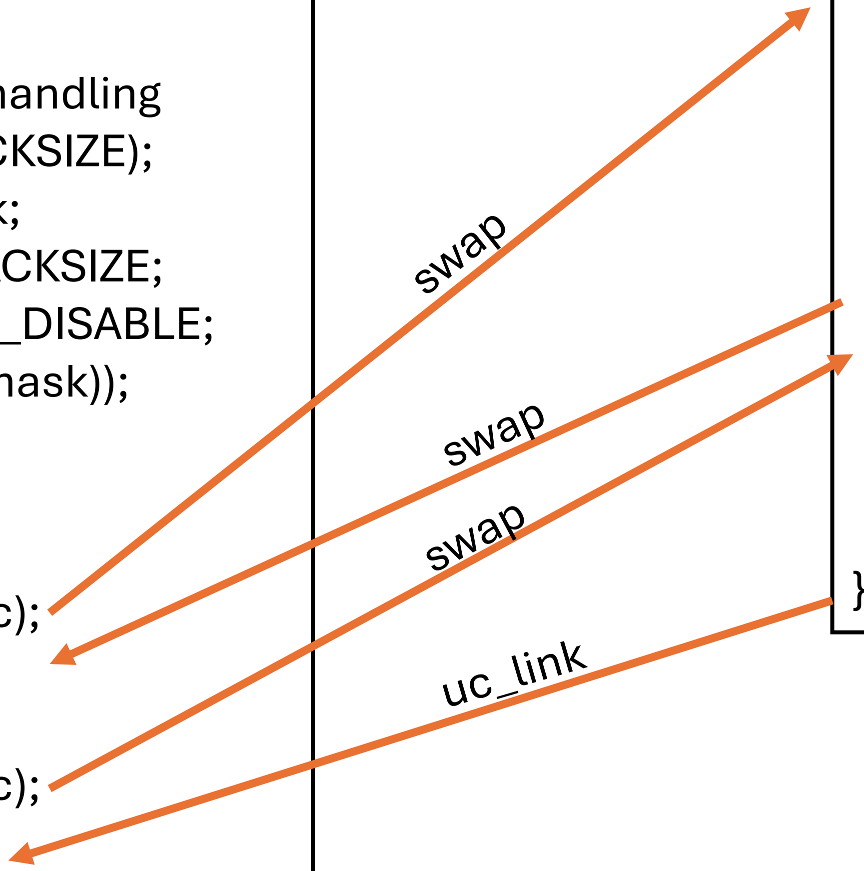
Example 3: Alternating functions

```
int main(int argc, char * argv[]){

  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = &mainc;
  makecontext(&uc, f, 0);

  swapcontext(&mainc, &uc);
  printf("Back in main 1\n");
  swapcontext(&mainc, &uc);
  printf("Back in main 2\n");
  free(stack);
  return 0;
}
```

```
void f(){
  int i = 0;
  while (i < 3) {
    printf("i = %d\n", i);
    i++;
  }
  swapcontext(&uc, &mainc);
  while (i < 6) {
    printf("i = %d\n", i);
    i++;
  }
}
```

Example 3: Alternating functions

```c
int main(int argc, char * argv[]){

  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = &mainc;
  makecontext(&uc, f, 0);

  swapcontext(&mainc, &uc);
  printf("Back in main 1\n");

  swapcontext(&mainc, &uc);
  printf("Back in main 2\n");
  free(stack);
  return 0;
}
```




```c
void f(){
  int i = 0;
  while (i < 3) {
    printf("i = %d\n", i);
    i++;
  }
  swapcontext(&uc, &mainc);
  while (i < 6) {
    printf("i = %d\n", i);
    i++;
  }
}
```

main() and f() are on different stacks

Example 3: Alternating functions

ucontext and valgrind

```
#include "valgrind.h"
#define STACKSIZE 4096

int main(int argc, char * argv[]){
  ucontext_t uc;
  getcontext(&uc); // initialize context

  // setup stack and signal handling
  void* stack = malloc(STACKSIZE);
  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = SS_DISABLE;
  sigemptyset(&(uc.uc_sigmask));
  uc.uc_link = NULL;

int valgrind_id = VALGRIND_STACK_REGISTER(
    stack, (char*) stack + STACKSIZE);

  makecontext(&uc, f, 0);
  setcontext(&uc);
  perror("setcontext");
  printf("boo\n"); // never prints

  free(stack);
  VALGRIND_STACK_DEREGISTER(valgrind_id);

  return 0;
}
```

User-space thread library with ucontext

Idea: Each thread runs its' function(s) in its own stack

When we switch between threads, we save the current context and swap to a new one

In your homework, you will implement different schemes for swapping between threads (next class)