Operating Systems
Lecture 5

Thread

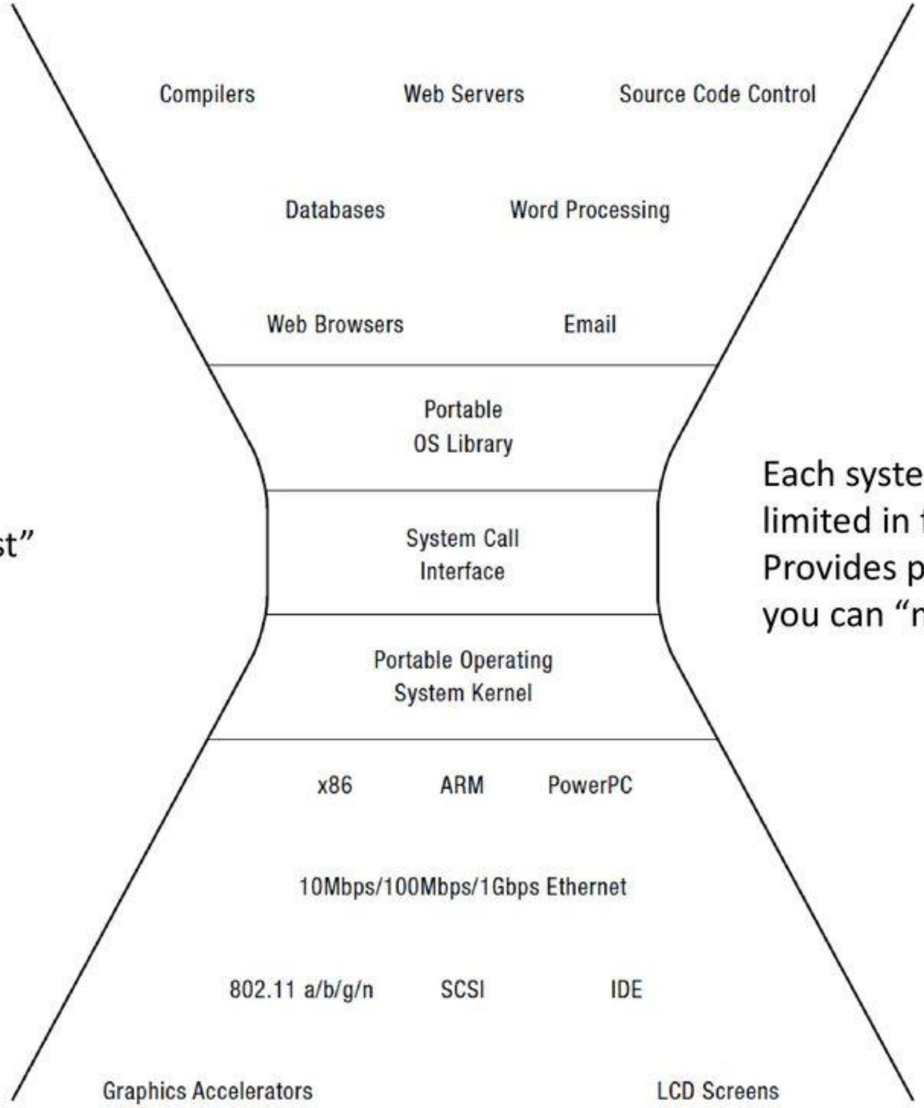
Prof. Mengwei Xu

Recap: OS Functions to Apps

- **Process management**
- **Input/output**
- Thread management
- Memory management
- File systems and storage
- Networking
- Graphics and window management
- Authentication and security

Recap: Syscall Design

- Flexibility
- Safety
- Reliability
- Performance



"thin waist"

Each system call can be limited in functionality. Provides primitives that you can "mix and match".

https://www.oilshell.org/blog/2022/03/backlog-arch.html

Recap: fork() in Unix

- A typical example of how fork() and exec() are used

```
int pid = fork();
if (pid == 0) {
    exec("foo");
} else {
    waitpid(pid, &status, options);
};
```

Child process

Parent process

Recap: File Descriptor in Unix

- File Descriptor (fd): a number (int) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

- Each process has its own file descriptor table
- A file can be opened multiple times and therefore associated with many file descriptors
- More in filesystem courses

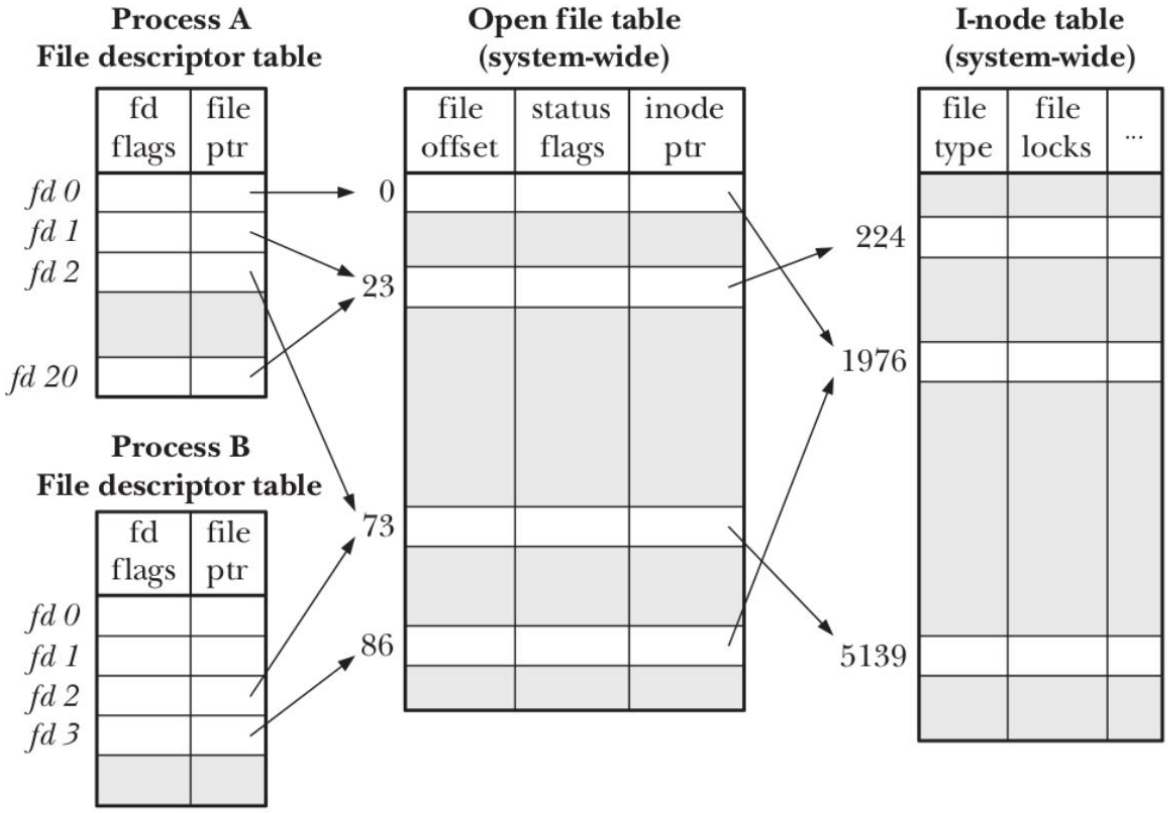
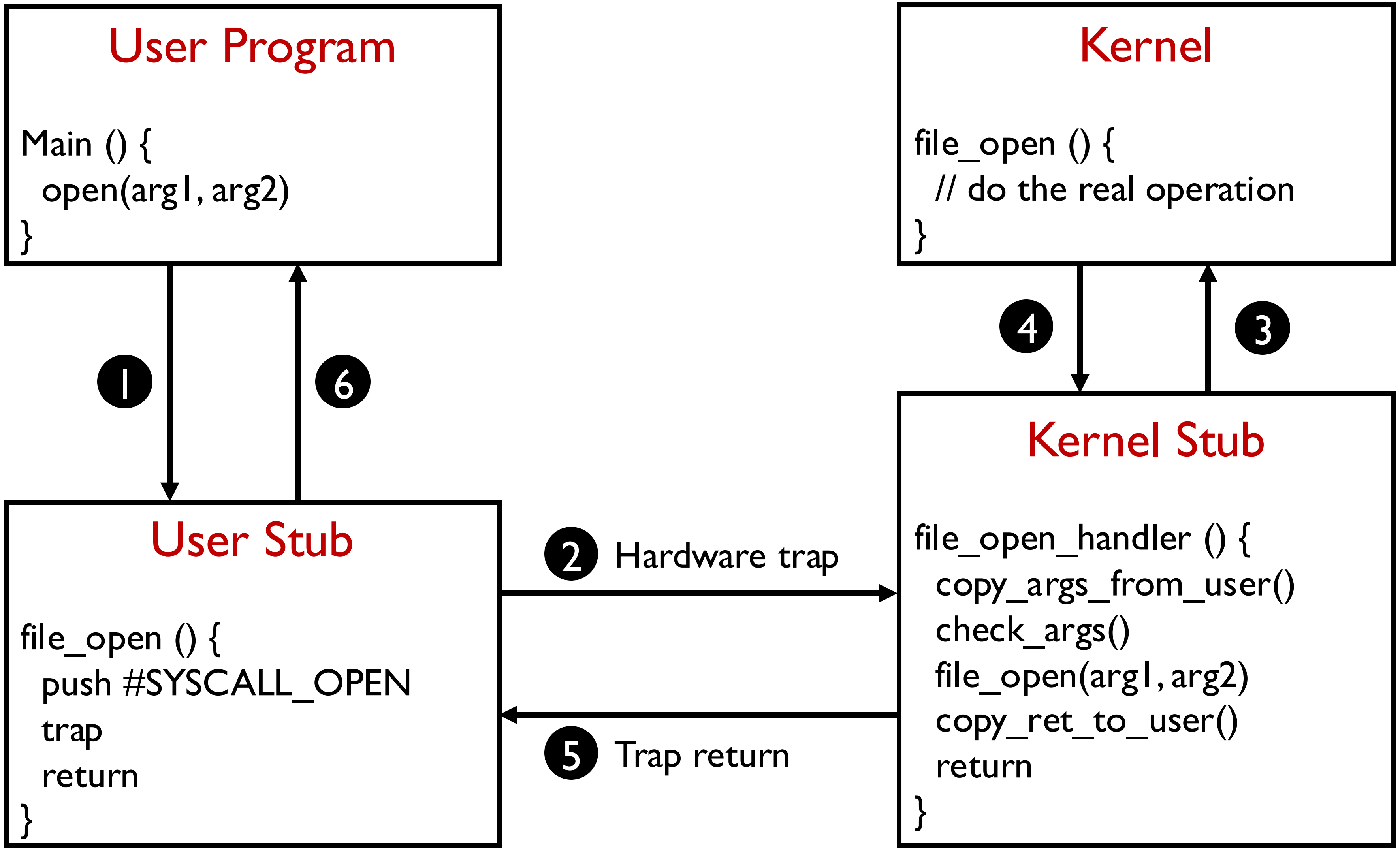


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Recap: System Calls Stubs



Recap: System Calls Stubs

https://developer.ibm.com/articles/l-kernel-memory-access/

- Can kernel directly access the parameters without copying?

- Why parameters must be copied from user memory to kernel memory?

- Can we check parameters before copying them to kernel memory?

Kernel Stub

```
file_open_handler () {
  copy_args_from_user()
  check_args()
  file_open(arg1, arg2)
  copy_ret_to_user()
  return
}
```

System Calls Stubs

https://developer.ibm.com/articles/l-kernel-memory-access/

- Can kernel directly access the parameters without copying?
  - Yes in most OSes, because kernel and user share memory space
- Why parameters must be copied from user memory to kernel memory?
  - Original parameters are stored in user memory stack
  - *copy_from_user* and *copy_to_usr*
- Can we check parameters before copying them to kernel memory?
  - time of check vs. time of use (TOCTOU) attack

Kernel Stub

```
file_open_handler () {
  copy_args_from_user()
  check_args()
  file_open(arg1, arg2)
  copy_ret_to_user()
  return
}
```

Goals for Today

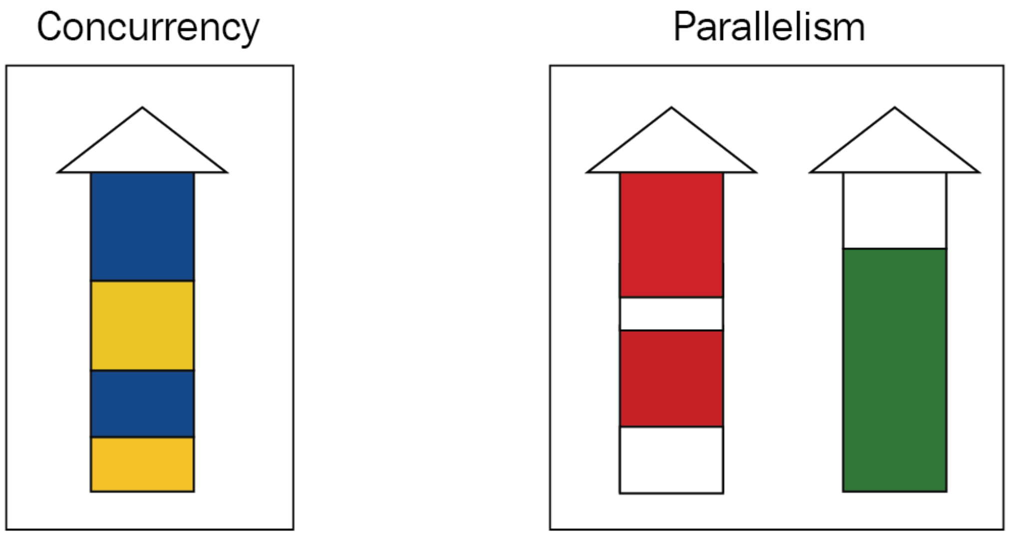
- Thread abstraction
- Thread implementation

Goals for Today

- **Thread abstraction**
- Thread implementation

Concurrency

- Concurrency (并发): multiple activities at the same time
  - Network service handles many client requests at the same time
  - User-interactive apps and background apps
- One of the most useful yet difficult concept in computer systems
- Concurrency vs. Multi-task vs. Parallel (并行)

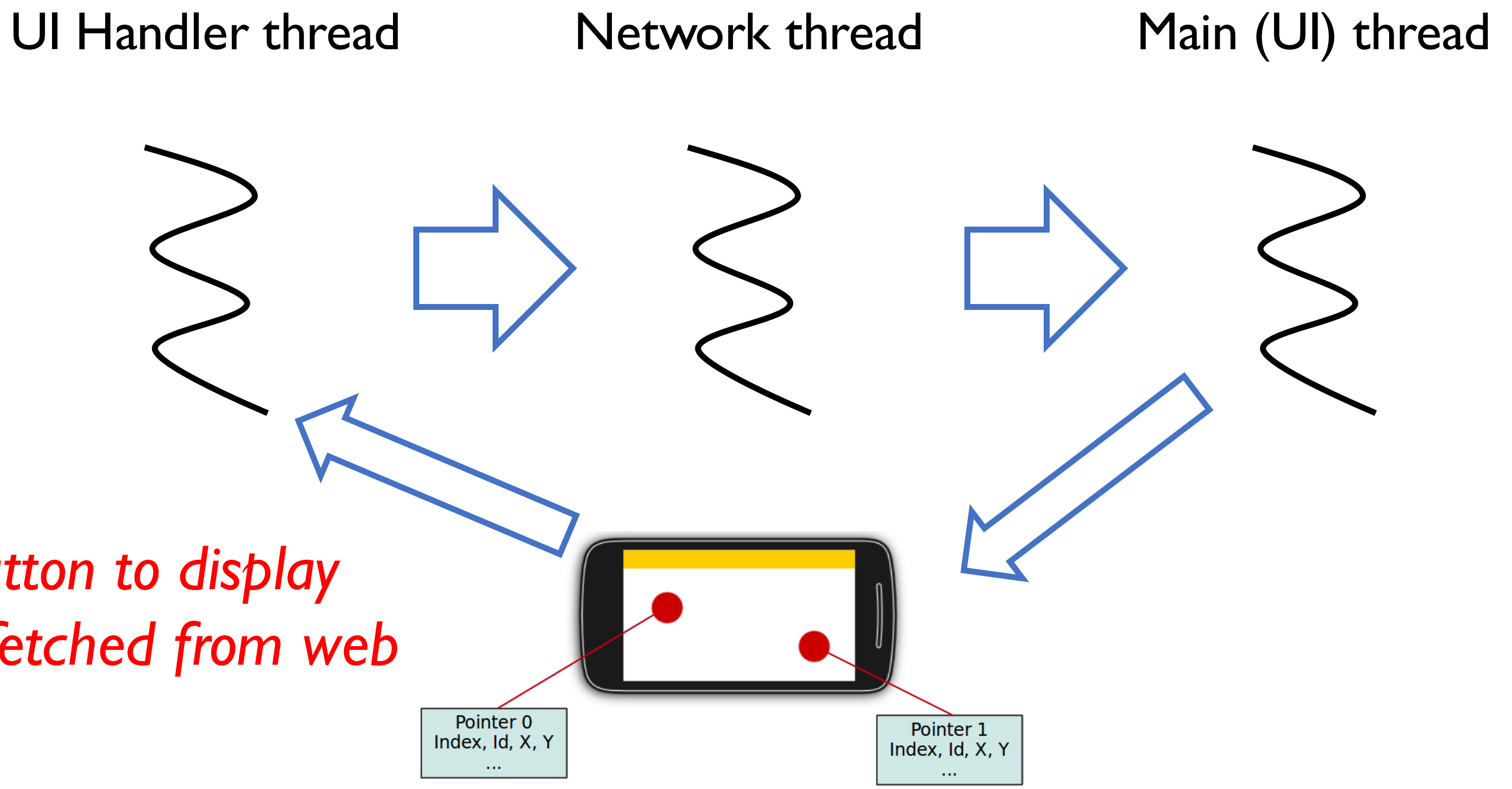


Concurrency is about *dealing with* lots of things at once

Parallelism is about *doing* lots of things at once

Thread Use Cases (1/4)

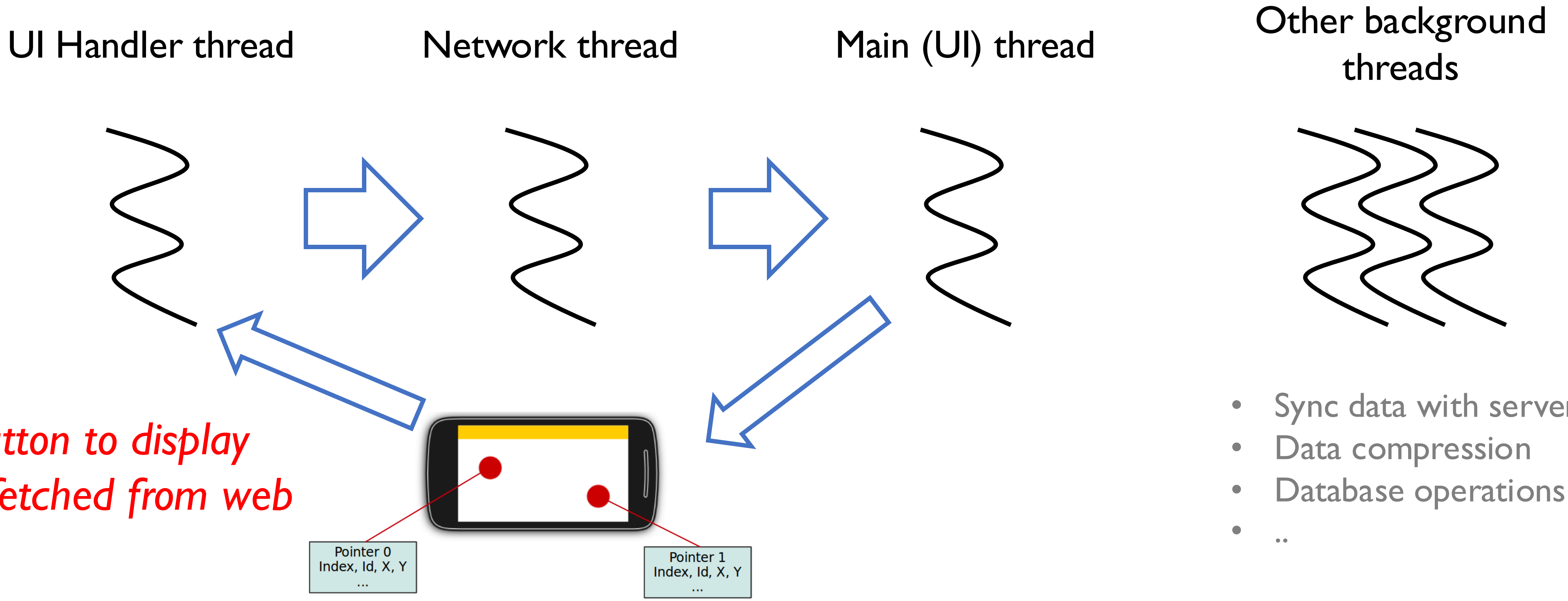
- Program structure: expressing logically concurrent tasks



Click a button to display contents fetched from web

Thread Use Cases (2/4)

- Responsiveness: shifting work to run in the background

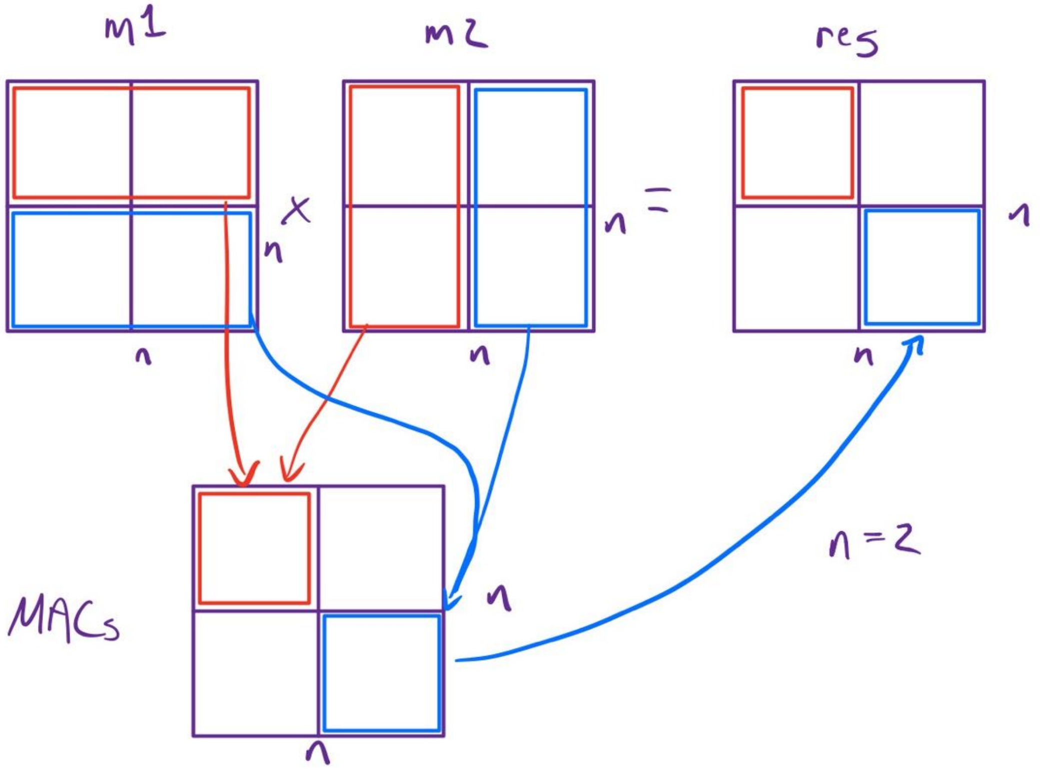


Click a button to display contents fetched from web

- Sync data with server
- Data compression
- Database operations
- ..

Thread Use Cases (3/4)

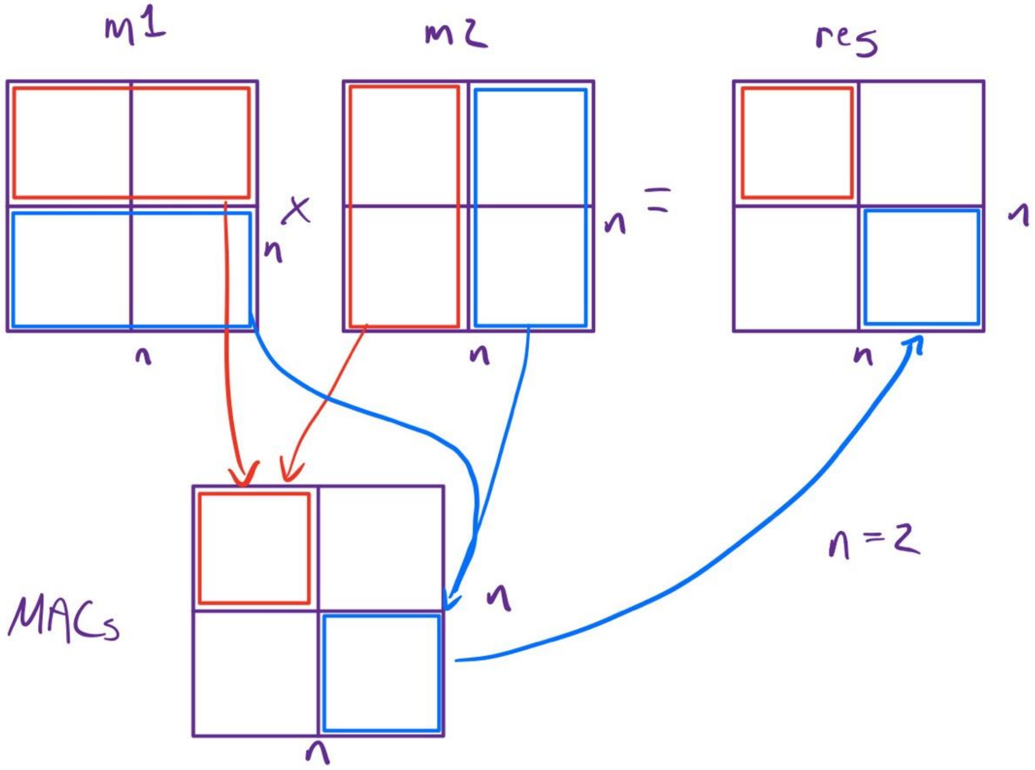
- Performance: exploiting multiple processors
  - Concurrency turns into parallelism



Extensively used in matrix operations and deep learning

Thread Use Cases (3/4)

- Performance: exploiting multiple processors
  - Concurrency turns into parallelism

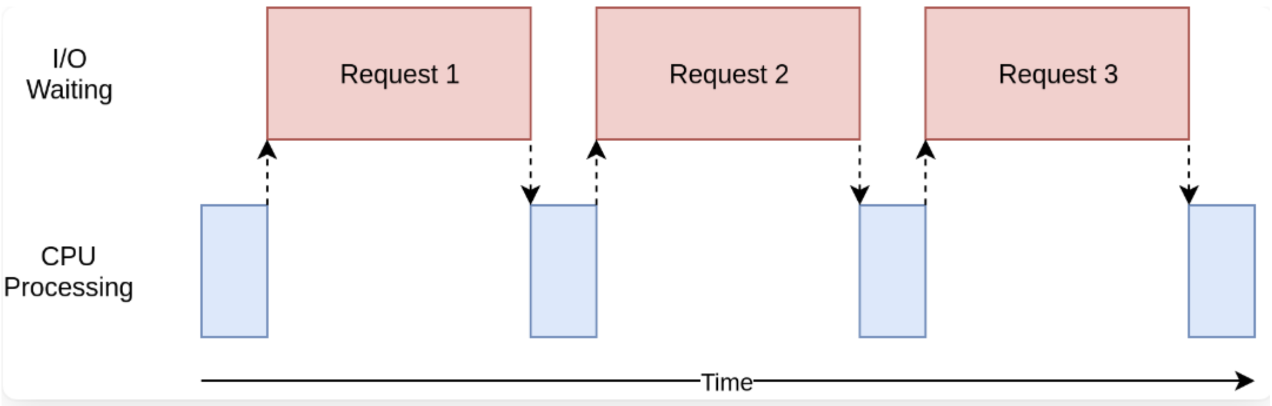


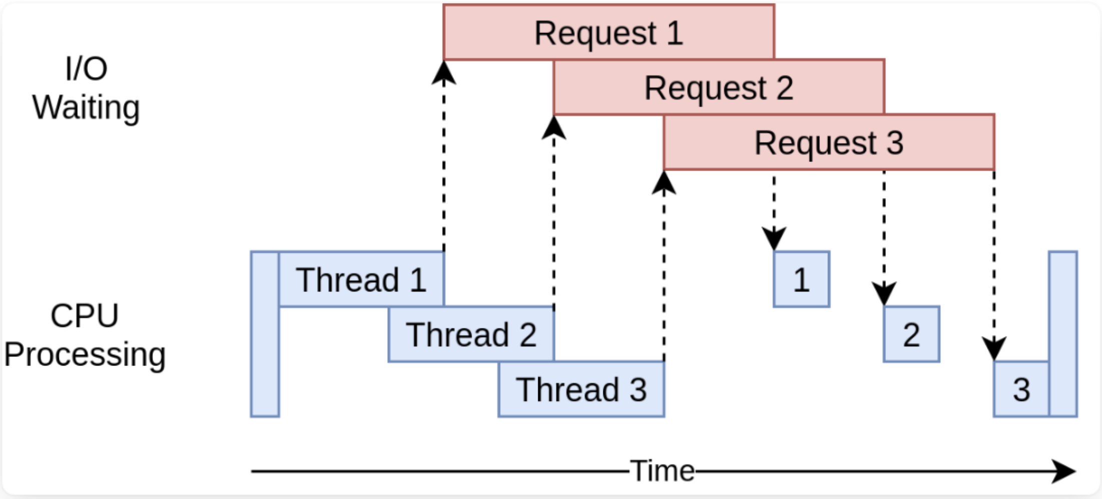
Extensively used in matrix operations and deep learning

- *Can more cores always bring speedup?*
- *How about asymmetric cores?*

Thread Use Cases (4/4)

- Performance: managing I/O devices
  - Processors are usually faster than I/O devices
  - Keep the processors busy!





Thread Abstraction

- Thread: a single execution sequence that represents a separately schedulable task

Each thread executes a sequence of instructions (assignments, conditionals, loops, procedures, etc) just as in the sequential programming model

The OS can run, suspend, or resume a thread at any time

Thread Abstraction

- Thread: a single execution sequence that represents a separately schedulable task

Each thread executes a sequence of instructions (assignments, conditionals, loops, procedures, etc) just as in the sequential programming model

The OS can run, suspend, or resume a thread at any time

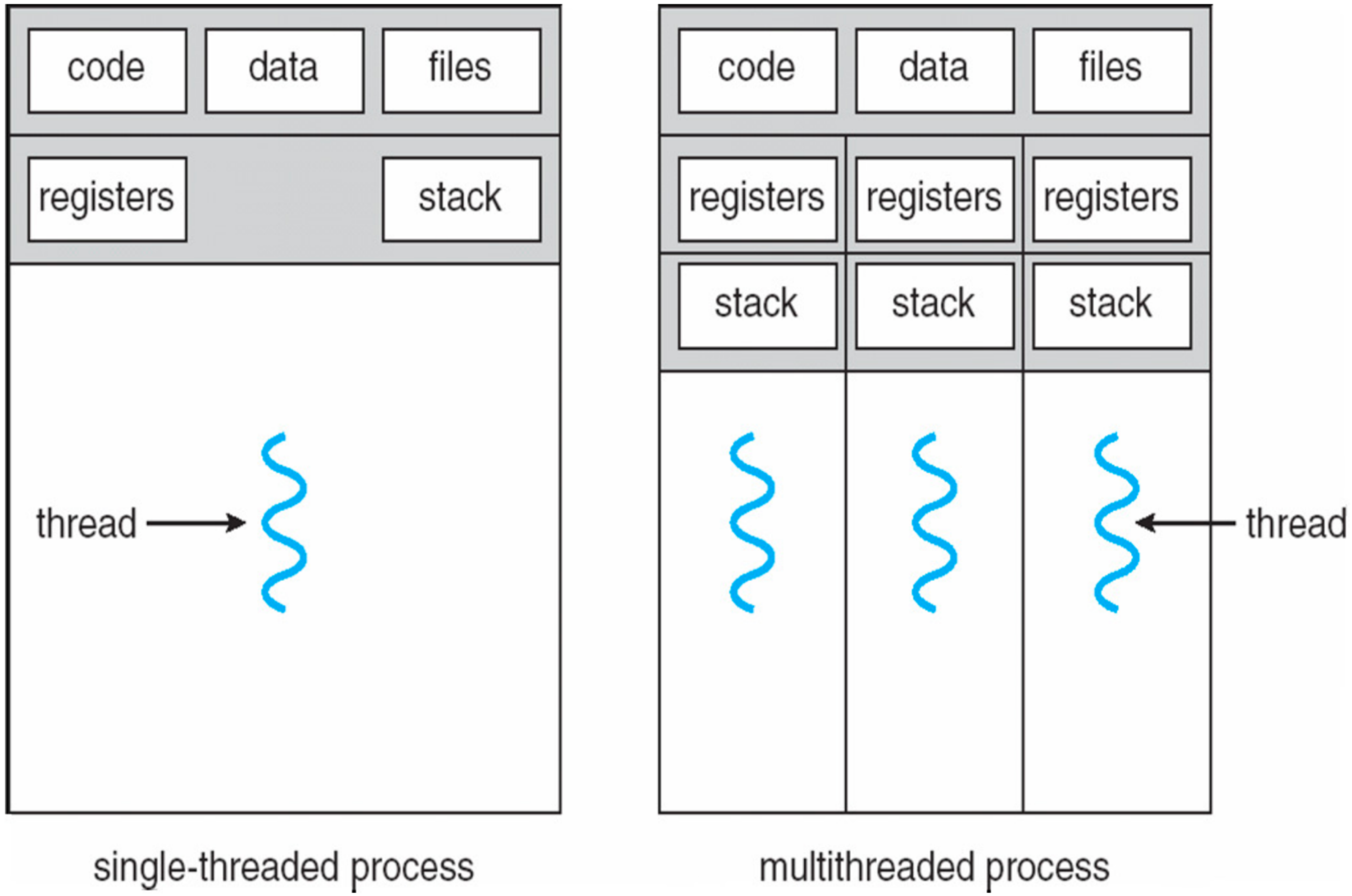
The minimal scheduling unit in OS!

Thread Abstraction

- Thread: a single execution sequence that represents a separately schedulable task

Threads in the same process share memory space, but not execution context

- There will be thread context switch



Thread Abstraction

- Thread execution speed is "unpredictable"
  - Thread switching is transparent to the code

Programmer's View

```
int main() {
   x = x + 1;
   y = y + 1;
   z = x + y;
}
```

Possible Execution #1

```
int main() {
   x = x + 1;
   y = y + 1;
   z = x + y;
}
```

Possible Execution #2

```
int main() {
   x = x + 1;
  ==========
  Thread suspended.
  Other thread running.
  Thread resumed
  ==========
   y = y + 1;
   z = x + y;
}
```

Possible Execution #3

```
int main() {
   x = x + 1;
   y = y + 1;
  ==========
  Thread suspended.
  Other thread running.
  Thread resumed
  ==========
   z = x + y;
}
```

Thread vs. Process

	Thread	Process
Currency	Both of them can be scheduled by OS.	
Context	Different threads/processes have their dedicated execution contexts (registers values and stacks). Scheduling them incurs context switching.	
Definition	A single execution sequence that represents a separately schedulable task	An execution of any program
	The minimal scheduling unit “a lightweight process”	The minimal dedicated memory space
Resources	Consume less resources	Consume more resources
Memory	Threads in the same process share memory space	Processors do not share memory space
Communications	Easier and faster for threads in the same process to communicate with each other	More complex and slow for different processes to communicate with each other

POSIX Thread APIs

#include <pthread.h>, Compile and link with -pthread.	
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);	Creates a new thread with attributes specified in attr, storing information about it in thread. Concurrently with the calling thread, thread executes the function start_routine with the argument arg.
int pthread_join(pthread_t thread, void **retval);	Waits for the thread specified by thread to terminate. If that thread has already terminated, it returns immediately. The thread specified by thread must be joinable. It copies the exit status of the target thread into the location pointed to by retval.
int pthread_yield();	The calling thread voluntarily gives up the processor to let some other threads run. The scheduler can resume running the calling thread whenever it chooses to do so.
void pthread_exit(void *retval);	Terminates the calling thread and returns a value via retval that. If another thread is already waiting in a call to **thread_join**, resume it.

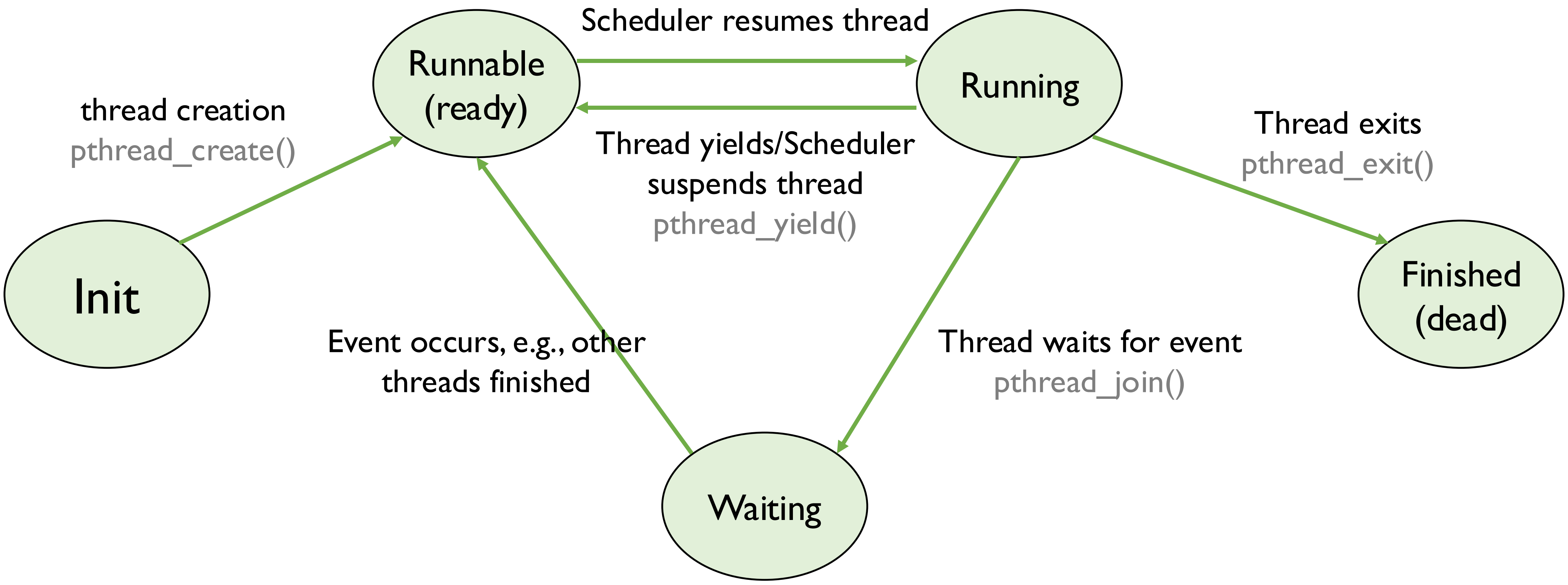
It looks like an *asynchronous procedure call*

POSIX Thread Example

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

What's the possible output?

Thread Lifecycle



Goals for Today

- Thread abstraction
- **Thread implementation**

Thread Data Structures

- Thread Control Block (TCB)
  - Stack pointer: each thread needs their own stack
  - Copy of processor registers
    - ❑ General-purpose registers for storing intermediate values
    - ❑ Special-purpose registers for storing instruction pointer and stack pointer
  - Metadata
    - ❑ Thread ID
    - ❑ Scheduling priority
    - ❑ Status
  - What's different from PCB??

Thread Data Structures

- Thread Control Block (TCB)
  - Stack pointer: each thread needs the
  - Copy of processor registers
    - ❑ General-purpose registers for storing int
    - ❑ Special-purpose registers for storing ins
  - Metadata
    - ❑ Thread ID
    - ❑ Scheduling priority
    - ❑ Status

```c
struct thread {
	union {
		struct rb_node	rb_node;
		struct list_head node;
	};
	struct maps		*maps;
	pid_t			pid_; /* Not all tools update this */
	pid_t			tid;
	pid_t			ppid;
	int			cpu;
	int			guest_cpu; /* For QEMU thread */
	refcount_t		refcnt;
	bool			comm_set;
	int			comm_len;
	bool			dead; /* if set thread has exited */
	struct list_head	namespaces_list;
	struct rw_semaphore	namespaces_lock;
	struct list_head	comm_list;
	struct rw_semaphore	comm_lock;
	u64			db_id;

	void			*priv;
	struct thread_stack	*ts;
	struct nsinfo		*nsinfo;
	struct srccode_state	srccode_state;
	bool			filter;
	int			filter_entry_depth;

	/* LBR call stack stitch */
	bool			lbr_stitch_enable;
	struct lbr_stitch	*lbr_stitch;
};
```

https://github.com/torvalds/linux/blob/master/tools/perf/util/thread.h

Thread Data Structures

- Thread Control Block (TCB)
  - Stack pointer: each thread needs their own stack
  - Copy of processor registers
    - ❑ General-purpose registers for storing intermediate values
    - ❑ Special-purpose registers for storing instruction pointer and stack pointer
  - Metadata
    - ❑ Thread ID
    - ❑ Scheduling priority
    - ❑ Status
- How large is the stack?
  - In kernel, it's usually small: 8KB in Linux on Intel x86
  - In user space, it's library-dependent
    - ❑ Most libraries check if there is a stackoverflow
    - ❑ Few PL/libs such as Google Go will automatically extend the stack when needed

Thread Data Structures

- Thread Control Block (TCB)
- Shared state
  - Code
  - Global variables
  - Heap variables

Thread Data Structures

- Thread Control Block (TCB)
- Shared state
- OS does not enforce physical division on threads' own separated states
  - If thread A has a pointer to the stack location of thread B, can A access/modify the variables on the stack of thread B?

Thread Implementation

- Kernel threads
  - What are the use cases?

- User-level threads
  - Can be implemented with or without kernel help

Calling Conventions

```
void func_A() {
    int a = 1 + 2;
    int c = func_B(a);
    print(c);
}
```

```
int func_B(int x) {
    int y = x + 3;
    return y;
}
```

- How does func_A goes to func_B?
- How does func_B return correctly back to func_A?

} stack

Search for "calling conventions" and try to understand what happens at assembly/instruction level

Implementing Kernel Threads

- Create a thread
  - Allocate per-thread state: the TCB and stack
  - Initialize per-thread state: registers (args)
  - Put TCB on ready list

```
// explained later
void thread_dummySwitch(TCB tcb) {
  *(tcb->sp) = stub;
  tcb->sp--;
  tcb->sp -= SizeOfPopad;
}
```

```
void thread_create(thread_t *thread, void
(*func)(int), int arg) {
  TCB *tcb = new TCB();
  thread->tcb = tcb;
  tcb->stack_size = INITIAL_STACK_SIZE;
  tcb->stack = new Stack(tcb->stack_size);
  tcb->sp = tcb->stack + tcb->stack_size;
  tcb->pc = stub;

  *(tcb->sp) = arg;
  tcb->sp --;
  *(tcb->sp) = func;
  tcb->sp --;

  thread_dummySwitch(tcb);
  tcb->state = READY;
  readyList.add(tcb);
}

void stub(void (*func)(int), int arg) {
  (*func)(arg);
  thread_exit(0);
}
```

Implementing Kernel Threads

- (Voluntary) kernel thread context switch
  - thread_yield()

- (Involuntary) kernel thread context switch
  - Interrupts, exceptions

Implementing Kernel Threads

- (Voluntary) kernel thread context switch
  - Turn off interrupts (why?)
  - Get a next ready thread
  - Mark the old thread as ready
  - Add the old thread to readyList
  - Save all registers and stack point
  - Set stack point to the new thread
  - Restores all the register values

- How to ensure the correct return location?

```
void thread_yield() {
  TCB *chosenTCB;
  disableInterrupts(); // why??
  chosenTCB = readyList.getNextThread();
  if (chosenTCB == NULL) {
    // Nothing to do here
  } else {
    runningThread->state = READY;
    readyList.add(runningThread);
    thread_switch(runningThread, chosenTCB);
    runningThread->state = RUNNING;
  }
  enableInterrupts();
}

void thread_switch(oldTCB, newTCB) {
  pushad;
  oldTCB->sp = %esp;
  %esp = newTCB->sp;
  popad;
  return;
}
```

Implementing Kernel Threads

Logical View

Thread 1	**Thread 2**

go(){
  while(1){
   thread_yield();
  }
}
	go(){
	  while(1){
	   thread_yield();
	  }
	}

Physical Reality

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB

	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	return from thread_switch	return from thread_switch
	return from thread_yield	return from thread_yield
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
...	...	...

A Few Questions

- When does switch (change of pc) actually happen?
- What's the goal of thread_dummpySwitch?
- What's the purpose of stub function and how it is correctly called (with correct args)?
- Why we need to disable interrupts during thread_switch?

Implementing Kernel Threads

- (Involuntary) kernel thread context switch
  - Save the states
  - Run the kernel’s handler
  - Restore the states

- Almost identical to user-mode transfer (3rd course), except:
  - There’s no need to switch modes (or stacks)
  - The handler can resume any thread on the ready list rather than always resuming the thread/process that was just suspended

Implementing Kernel Threads

- Delete a thread
  - Remove the thread from the ready list so it will never run again
  - Free the per-thread state allocated for the thread

- Can a thread delete its own state?
  - A bad case: a thread removes itself from the ready list, and an interrupt occurs..
  - A worse case: a thread frees its own state (stack), and..

Implementing Kernel Threads

- Delete a thread
  - Remove the thread from the ready list so it will never run again
  - Free the per-thread state allocated for the thread

- Can a thread delete its own state?
  - A bad case: a thread removes itself from the ready list, and an interrupt occurs..
  - A worse case: a thread frees its own state (stack), and..
- Solution
  - The thread moves its TCB from the ready list to a list of *finished* threads
  - Let *other* threads free those finished threads

Implementing Multi-threaded Processes

- Implementing user-level multi-threaded processes through
  1. Kernel threads (each thread op traps into kernel)
  2. User-level libraries (no kernel support)
  3. Hybrid mode

Implementing Multi-threaded Processes

- Implementing multi-threaded processes through kernel threads
  - Each thread operation invokes the corresponding kernel thread syscall

Create a kernel thread

- Allocate per-thread state in kernel: the TCB and stack
- Initialize per-thread state: registers (args)
- Put TCB on ready list

Create a user-level thread

- User lib allocates a user-level stack
- Invokes thread_create() syscall
- Stores a pointer to the TCB in the PCB (why?)

How about join, yield, exit?

Implementing Multi-threaded Processes

- Implementing multi-threaded processes in user libraries
  - The library maintains everything in user space
    - TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call

Implementing Multi-threaded Processes

- Implementing multi-threaded processes in user libraries
  - The library maintains everything in user space
    - TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call
- How can we make user-level threads run currently, as kernel is not aware of their existence?
- How can program change the PC and stack pointer?

Implementing Multi-threaded Processes

- Implementing multi-threaded processes in user libraries
  - The library maintains everything in user space
    - TCBs, stacks, ready list, finished list
  - The library determines which thread to run
  - A thread op is just a procedure call
- How can we make user-level threads run currently, as kernel is not aware of their existence?
  - The preemptive way: timer interrupts (upcall) from kernel
  - The cooperative way: threads yield voluntarily
- How can program change the PC and stack pointer?
  - jmp and esp

Threads in Kernel vs. User

	User-level Threads	Kernel Threads
Currency	Both of them run currently	
Context	Share heap/code, but have separated stack/registers	
Role of kernel	No kernel assistance at all	Each thread operation invokes kernel syscall
Speed (context switch, creating, etc)	Fast	Slow
Memory cost	Small	Large
I/O waiting time	Cannot avoid the I/O waiting time (though there are certain optimizations to do so)	Kernel can schedule another thread when I/O blocks
Multi-core processor	No parallel on multi-core processors	Can schedule many threads in the same process at the same time on multi-core processors

Implementing Multi-threaded Processes

- Implementing multi-threaded processes in hybrid way: optimizations based on kernel threads
  - Hybrid thread join: for example, no need for syscall if the thread to be joined is already finished (with exit value saved in memory)
  - Per-processor kernel thread with user-level thread implementation
  - Scheduler activations: in recent Windows, the user-level scheduler can be notified when a thread blocks in a syscall, so it can schedule another thread to fully utilize the processor.

Homework

- Easy Lab 1: implementing a user-level threading library
  - Check it out on our website