# Operating Systems
# Lecture 5

# Thread

Prof. Mengwei Xu

# https://buptos.github.io

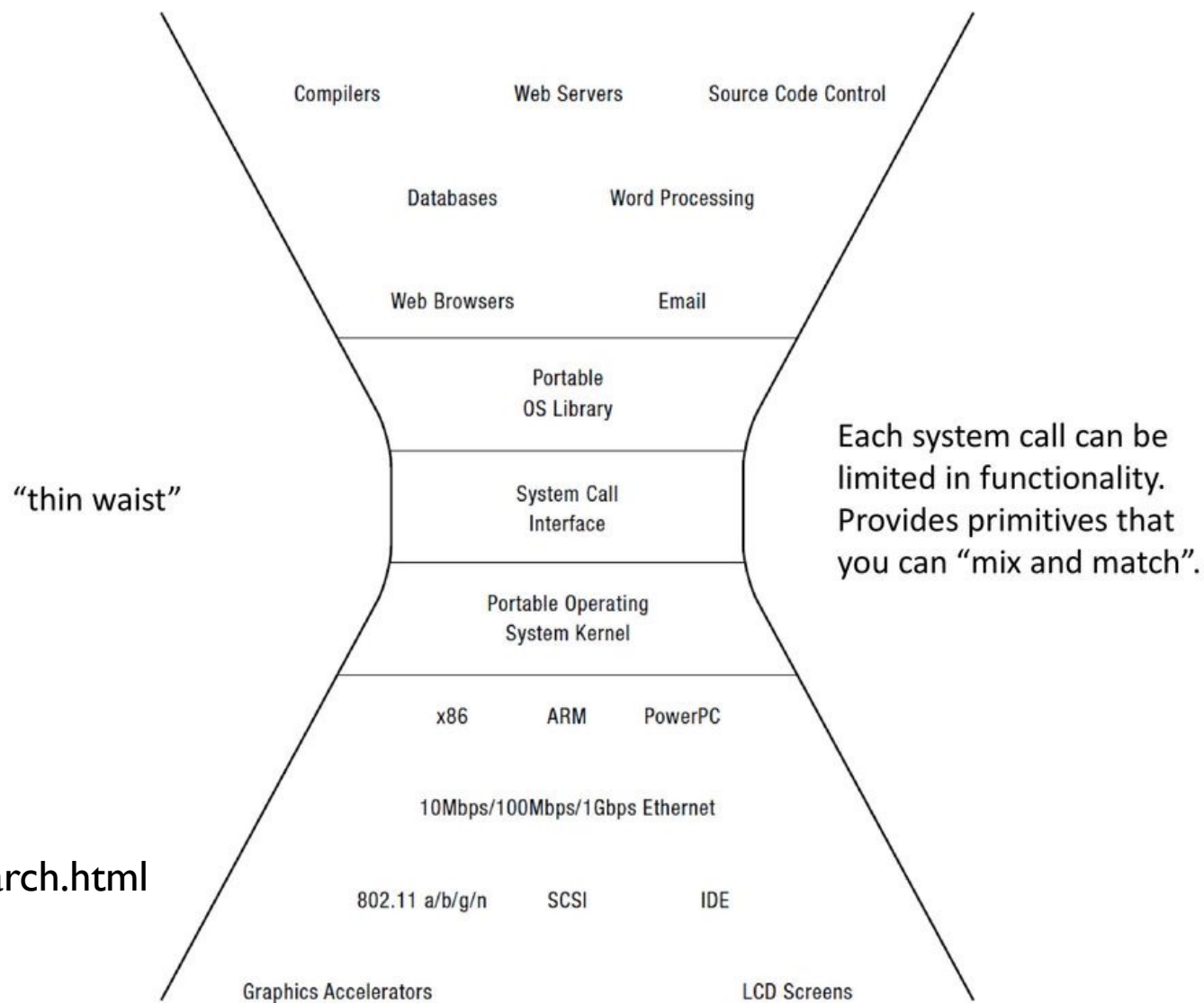# Recap: OS Functions to Apps

- <span style="color:red">Process management</span>

- <span style="color:red">Input/output</span>

- Thread management

- Memory management

- File systems and storage

- Networking

- Graphics and window management

- Authentication and security

- Flexibility
- Safety
- Reliability
- Performance

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

Compilers    Web Servers    Source Code Control

Databases    Word Processing

Web Browsers    Email

Portable
OS Library

"thin waist"

System Call
Interface

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

Portable Operating
System Kernel

x86    ARM    PowerPC

10Mbps/100Mbps/1Gbps Ethernet

802.11 a/b/g/n    SCSI    IDE

Graphics Accelerators    LCD Screens

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

```
int pid = fork();
if (pid == 0) {
        exec("foo");              ⟵———— Child process
} else {
    waitpid(pid, &status, options);
};
                                  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
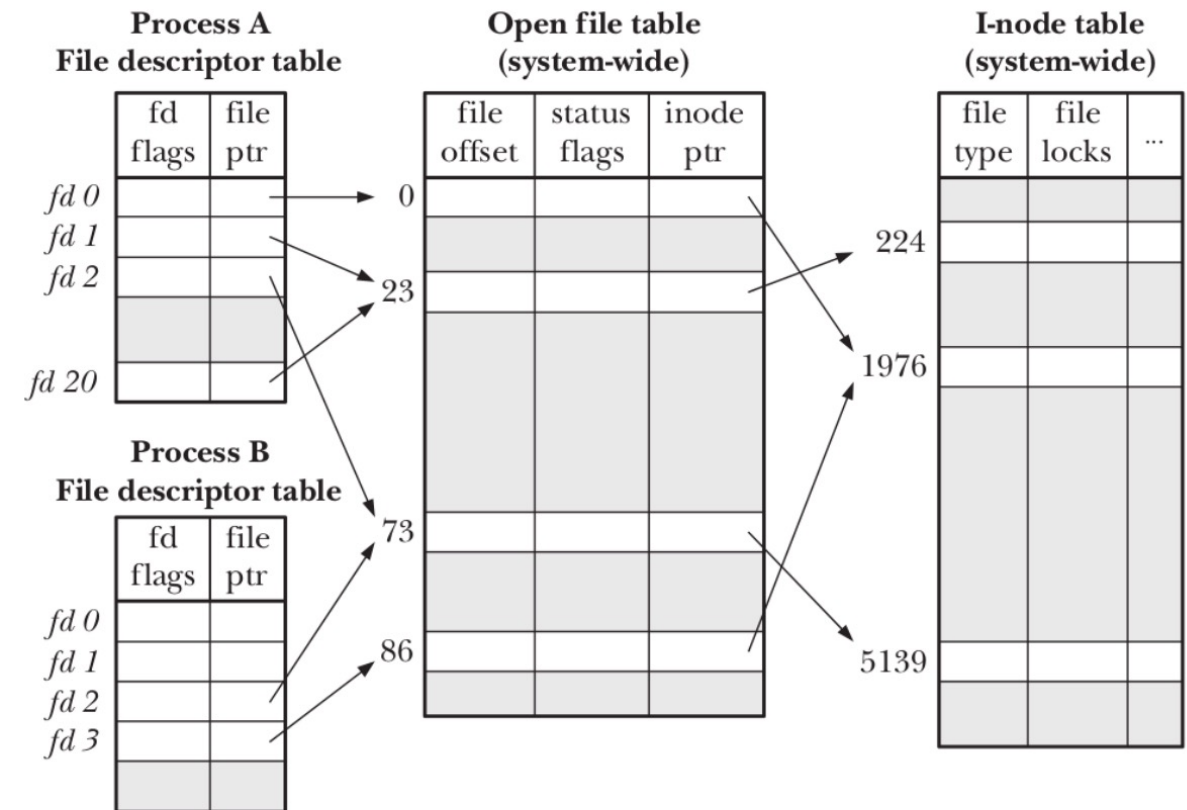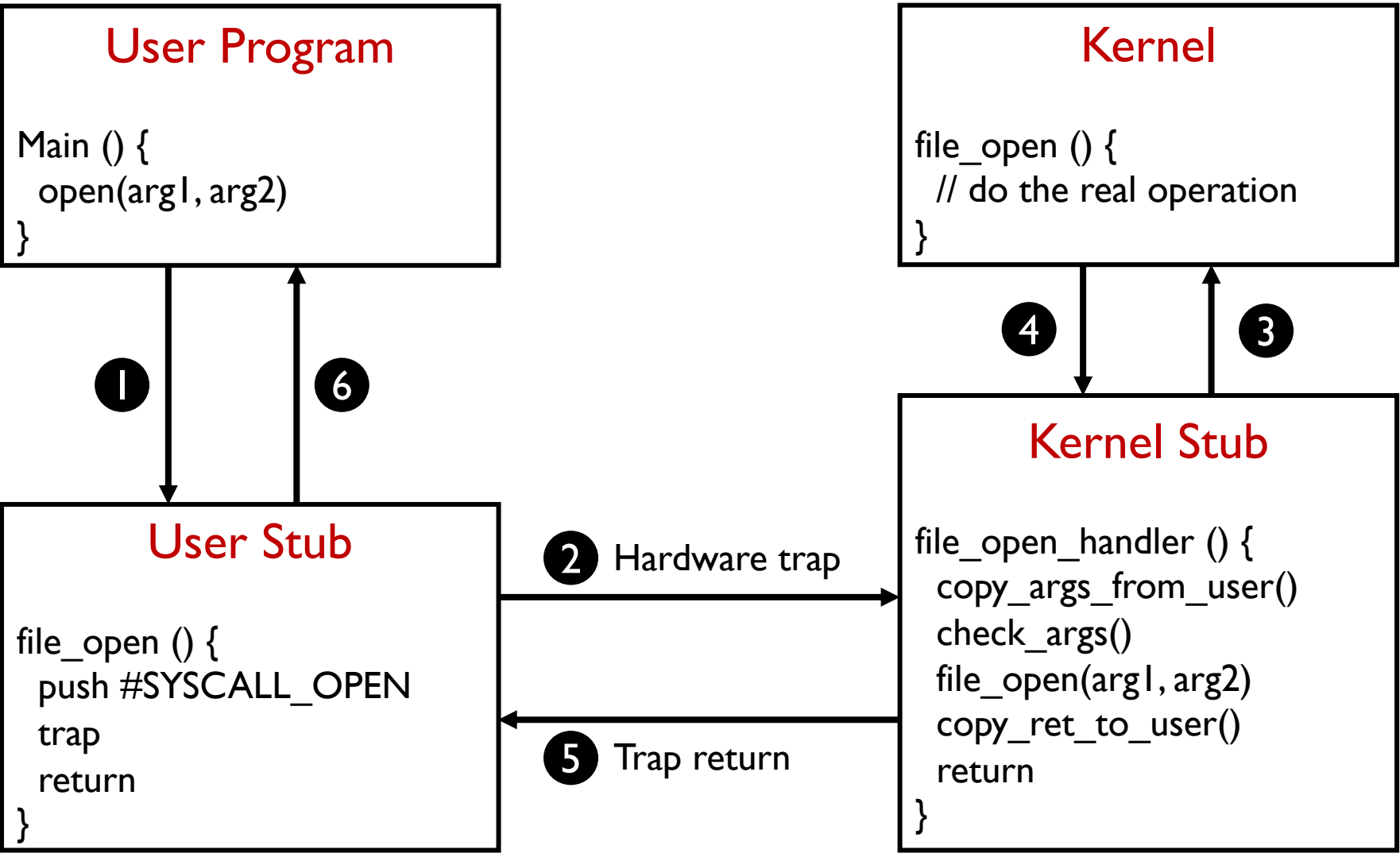- More in filesystem courses



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

# Recap: System Calls Stubs

**User Program**

```
Main () {
  open(arg1, arg2)
}
```

**Kernel**

```
file_open () {
  // do the real operation
}
```

❶

❻

❹

❸

**User Stub**

```
file_open () {
  push #SYSCALL_OPEN
  trap
  return
}
```

❷ Hardware trap

❺ Trap return

**Kernel Stub**

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

# Recap: System Calls Stubs

- 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?

<div style="border: 1px solid black; padding: 10px;">

## Kernel Stub

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

</div>

# Goals for Today

- Thread abstraction
- Thread implementation

# Goals for Today

- <span style="color:red">Thread abstraction</span>
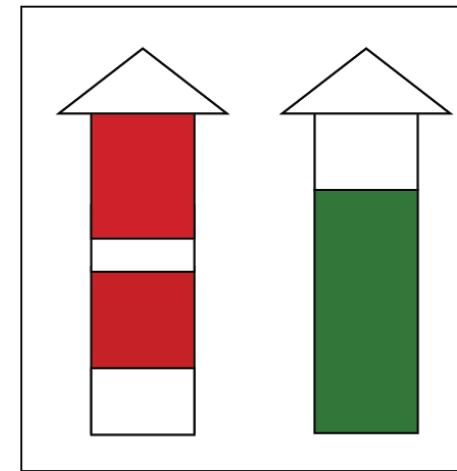
- Thread implementation

# 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)
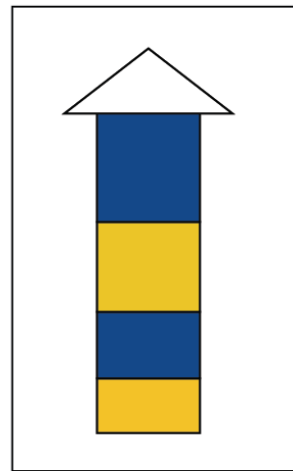


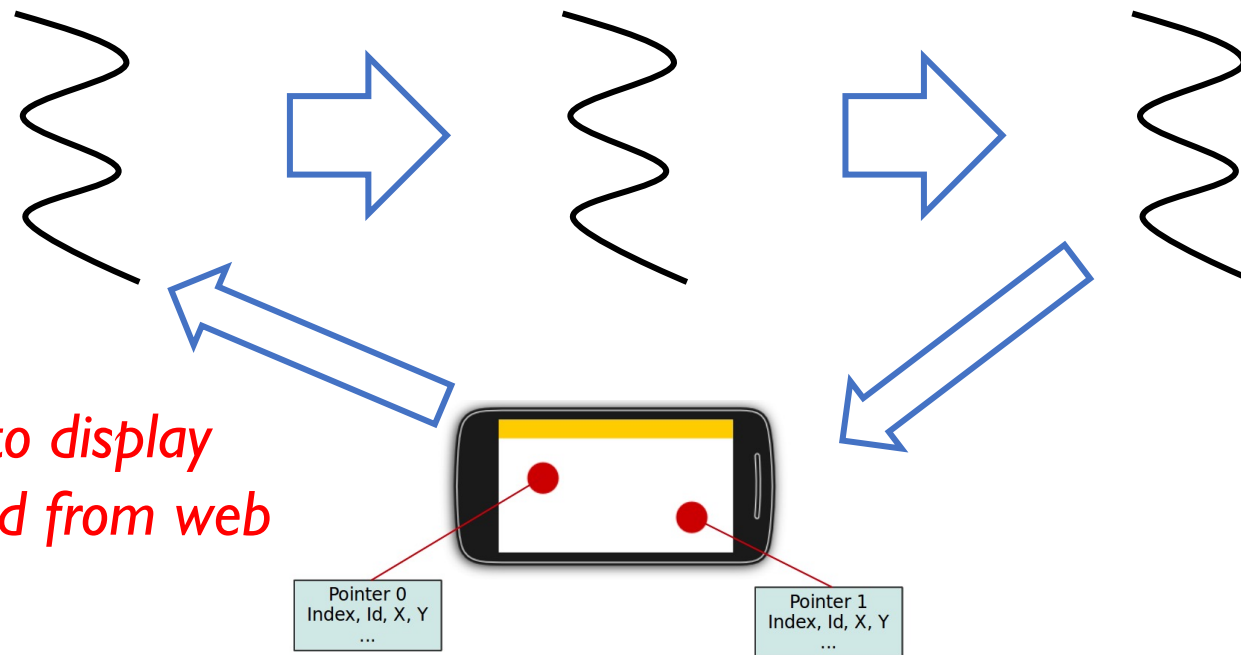| Concurrency | Parallelism |
|---|---|
| Concurrency is about *dealing with* lots of things at once | Parallelism is about *doing* lots of things at once |

# Thread Use Cases (1/4)

- Program structure: expressing logically concurrent tasks

UI Handler thread      Network thread      Main (UI) thread

*Click a button to display contents fetched from web*

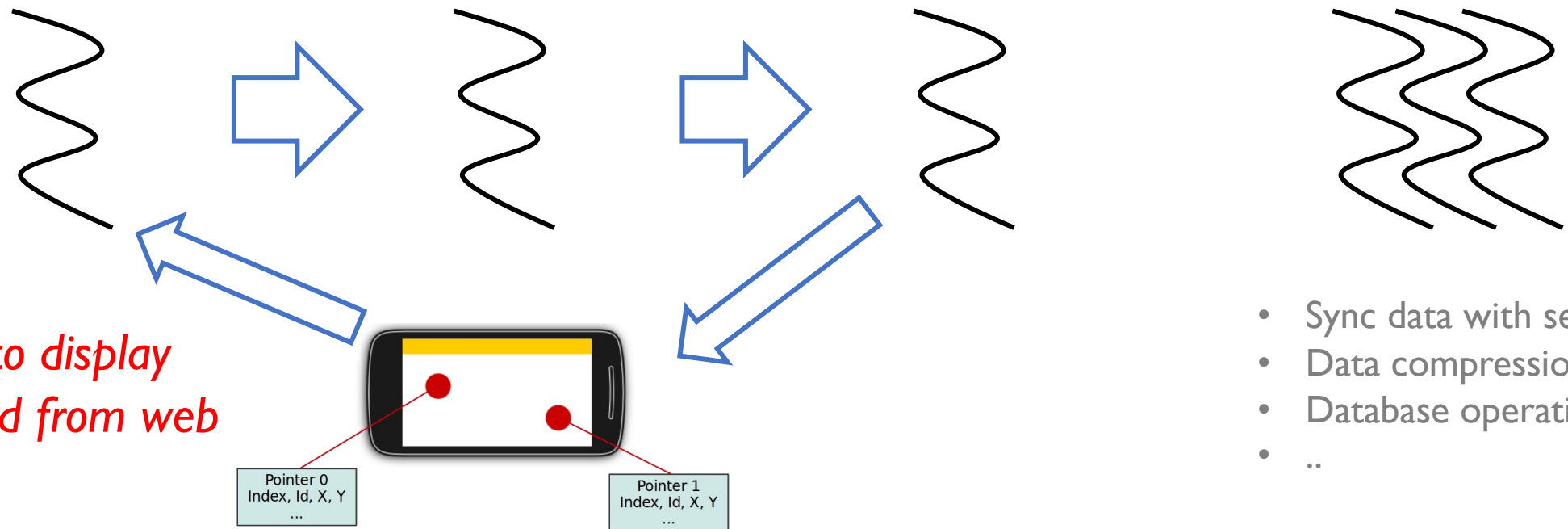Pointer 0
Index, Id, X, Y
...

Pointer 1
Index, Id, X, Y
...

- Responsiveness: shifting work to run in the background

UI Handler thread    Network thread    Main (UI) thread    Other background threads

*Click a button to display contents fetched from web*

Pointer 0
Index, Id, X, Y
...

Pointer 1
Index, Id, X, Y
...

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

- Performance: exploiting multiple processors
  - Concurrency turns into parallelism

m1    m2    res

$n \times n = n$

MACs

$n = 2$
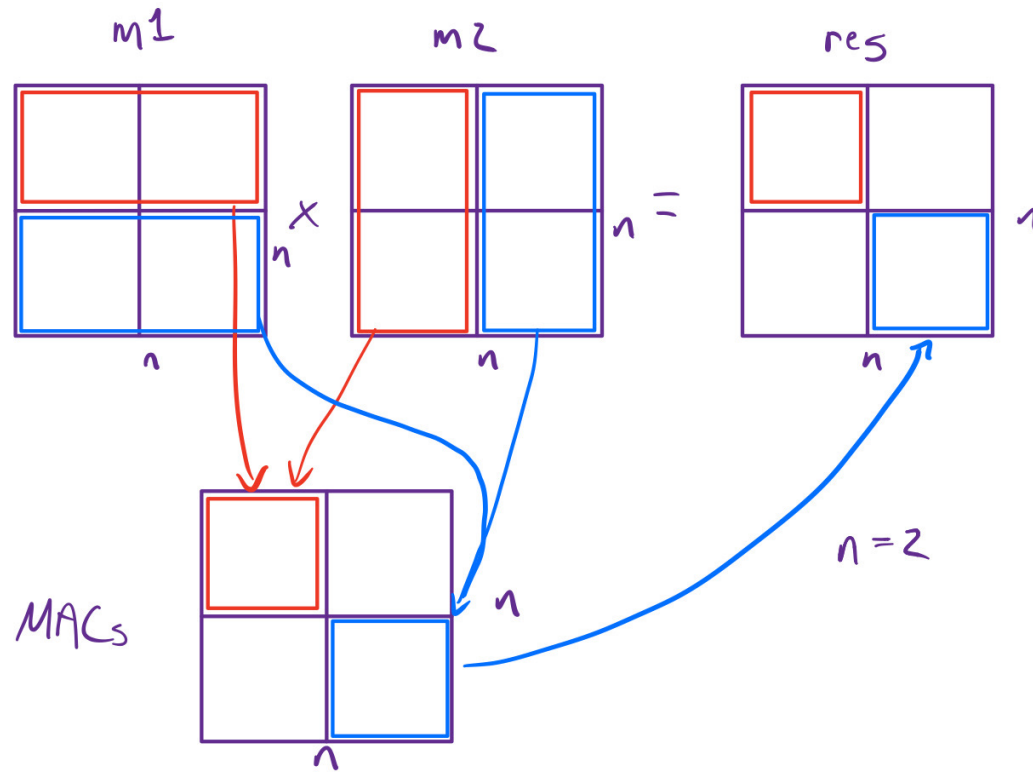
Extensively used in matrix operations and deep learning

- Performance: exploiting multiple processors
  - Concurrency turns into parallelism
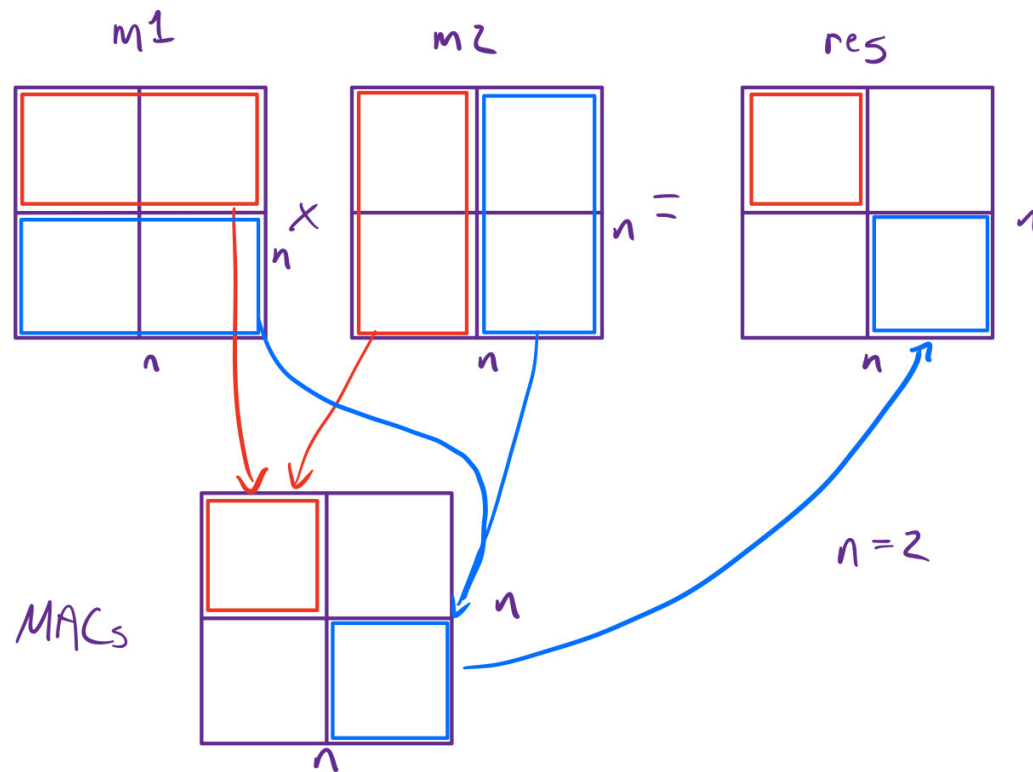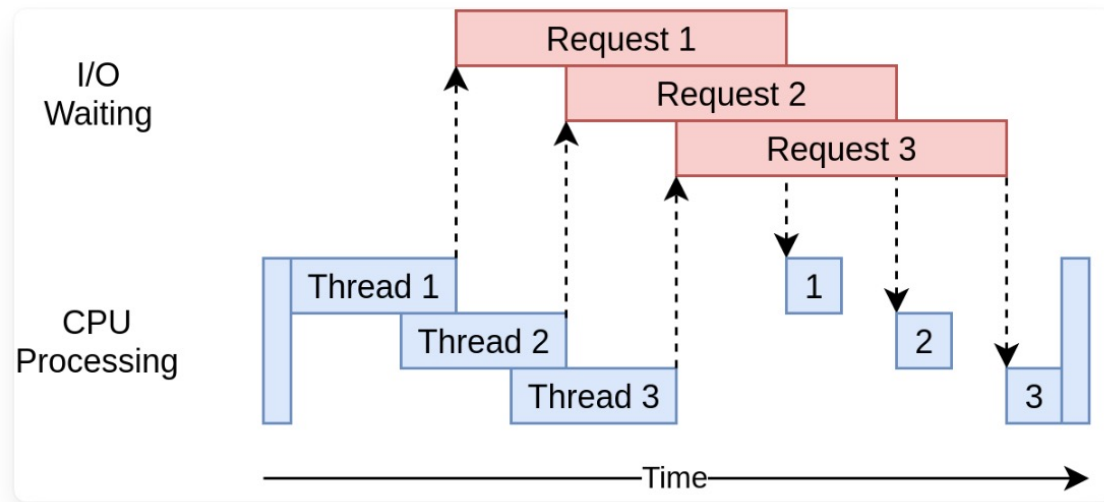


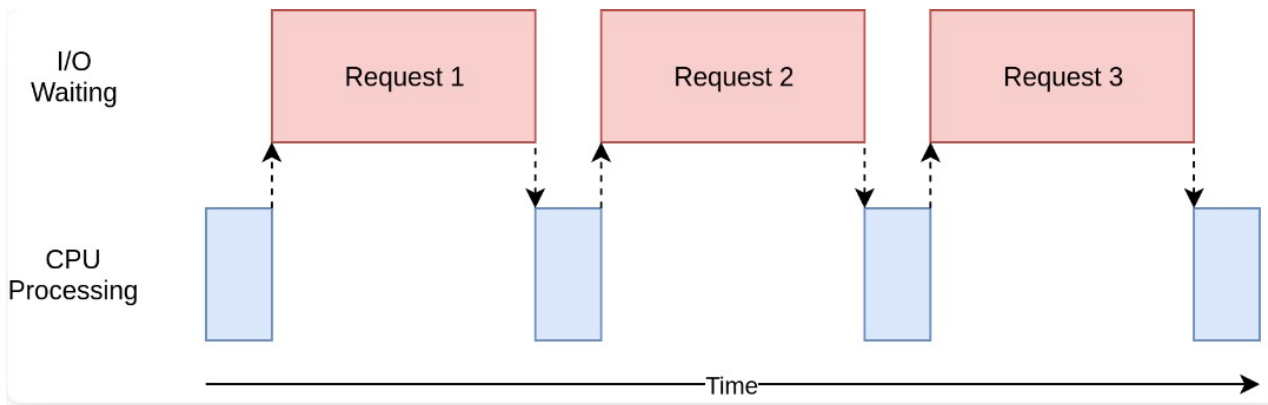- *Can more cores always bring speedup?*
- *How about asymmetric cores?*

Extensively used in matrix
operations and deep learning

- 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 <u>execution sequence</u> that represents a <u>separately schedulable task</u>

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
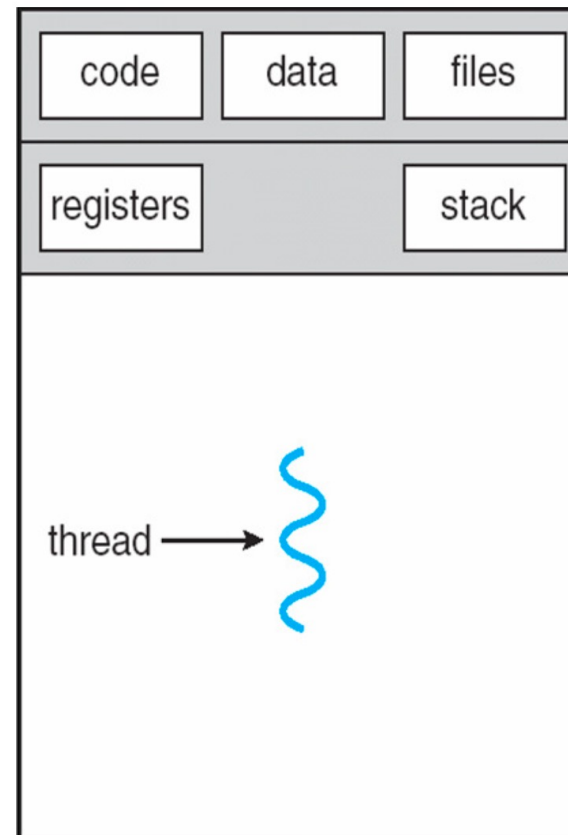
**The minimal scheduling unit in OS!**

# Thread Abstraction

- Thread: a single <u>execution sequence</u> that represents a <u>separately schedulable task</u>

Threads in the same process share memory space, but not execution context
- There will be thread context switch



single-threaded process
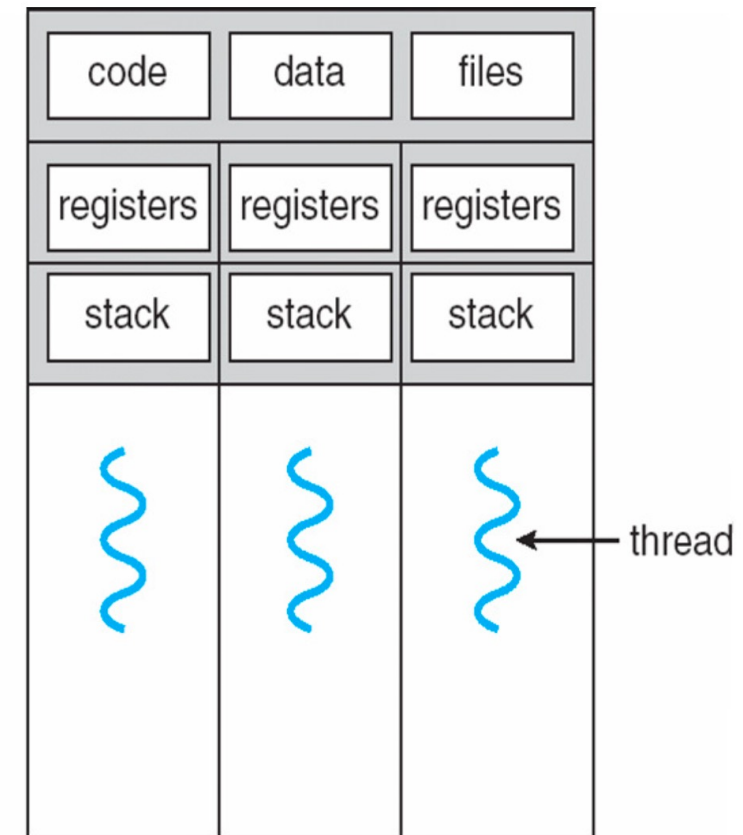
multithreaded process

# 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;
}
```

Mengwei Xu @ BUPT Fall 2022

# 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

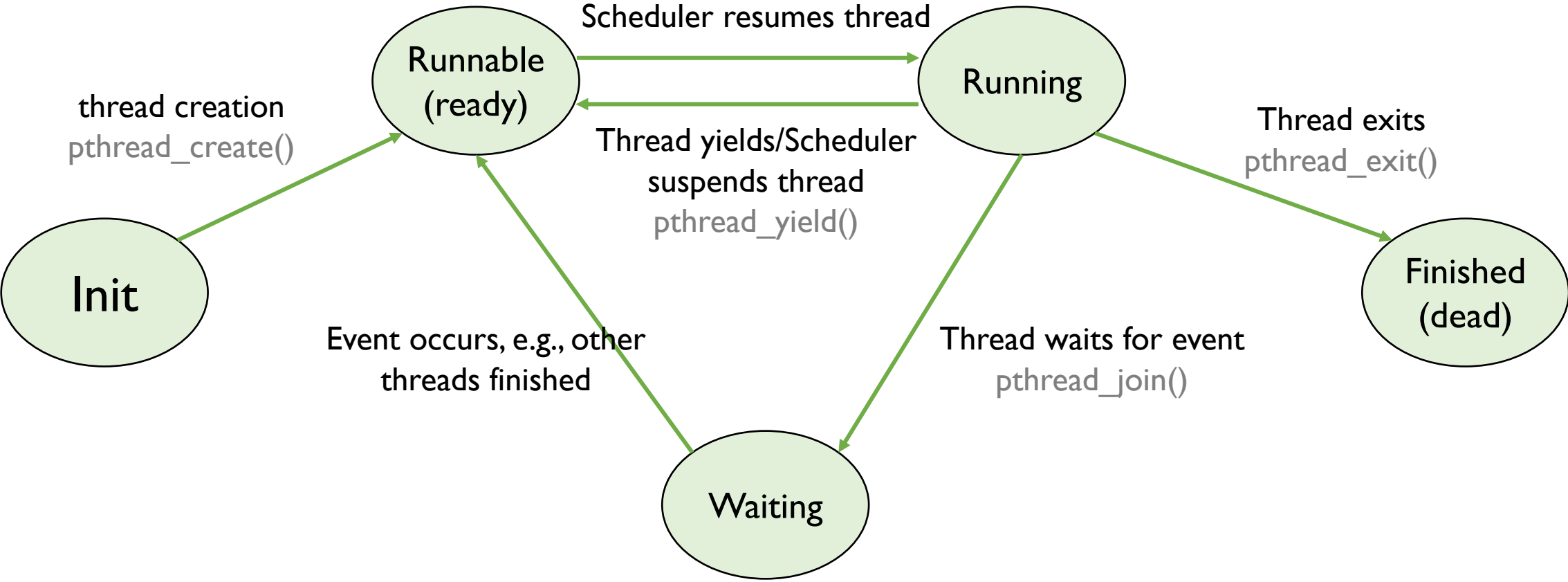| #include <pthread.h>, Compile and link with -pthread. | |
|---|---|
| int pthread_create(<br>  pthread_t *thread,<br>  const pthread_attr_t *attr,<br>  void *(*start_routine)(void *),<br>  void *arg<br>); | Creates a new thread with attributes specified in attr, storing information about it in thread. Concurrently with the calling thread, thread executes the function start_routine with the argument arg. |
| int pthread_join(<br>  pthread_t thread,<br>  void **retval<br>); | 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

What's the possible output?

```c
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <pthread.h>
4
5    void *print_message_function( void *ptr );
6
7    main()
8    {
9         pthread_t thread1, thread2;
10        char *message1 = "Thread 1";
11        char *message2 = "Thread 2";
12        int  iret1, iret2;
13
14        iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
15        iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
16
17        pthread_join( thread1, NULL);
18        pthread_join( thread2, NULL);
19
20        printf("Thread 1 returns: %d\n",iret1);
21        printf("Thread 2 returns: %d\n",iret2);
22        exit(0);
23   }
24
25   void *print_message_function( void *ptr )
26   {
27        char *message;
28        message = (char *) ptr;
29        printf("%s \n", message);
30   }
```

# 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 their ow
  - Copy of processor registers
    - ❑ General-purpose registers for storing interme
    - ❑ Special-purpose registers for storing instructi
  - Metadata
    - ❑ Thread ID
    - ❑ Scheduling priority
    - ❑ Status

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

```
32    struct thread {
33        union {
34                struct rb_node    rb_node;
35                struct list_head node;
36        };
37        struct maps            *maps;
38        pid_t                  pid_; /* Not all tools update this */
39        pid_t                  tid;
40        pid_t                  ppid;
41        int                    cpu;
42        int                    guest_cpu; /* For QEMU thread */
43        refcount_t             refcnt;
44        bool                   comm_set;
45        int                    comm_len;
46        bool                   dead; /* if set thread has exited */
47        struct list_head       namespaces_list;
48        struct rw_semaphore    namespaces_lock;
49        struct list_head       comm_list;
50        struct rw_semaphore    comm_lock;
51        u64                    db_id;
52
53        void                   *priv;
54        struct thread_stack    *ts;
55        struct nsinfo          *nsinfo;
56        struct srccode_state   srccode_state;
57        bool                   filter;
58        int                    filter_entry_depth;
59
60        /* LBR call stack stitch */
61        bool                   lbr_stitch_enable;
62        struct lbr_stitch      *lbr_stitch;
63    };
```

# 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

- <span style="color:red">OS does not enforce physical division on threads' own separated states</span>
  - 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

# 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
1.  void thread_dummySwitch(newThread) {
2.    *(tcb->sp) = stub;
3.    tcb->sp--;
4.    tcb->sp -= SizeOfPopad;
5.  }
```

```
1.   void thread_create(thread_t *thread, void
     (*func)(int), int arg) {
2.     TCB *tcb = new TCB();
3.     thread->tcb = tcb;
4.     tcb->stack_size = INITIAL_STACK_SIZE;
5.     tcb->stack = new Stack(tcb->stack_size);
6.     tcb->sp = tcb->stack + tcb_stack_size;
7.     tcb->pc = stub;

8.     *(tcb->sp) = arg;
9.     tcb->sp --;
10.    *(tcb->sp) = func;
11.    tcb->sp --;

12.    thread_dummySwitch(tcb);
13.    tcb->state = READY;
14.    readyList.add(tcb);
15.  }

16.  void stub(void (*func)(int), int arg) {
17.    (*func)(arg);
18.    thread_exit(0);
19.  }
```

# 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

Mengwei Xu @ BUPT Fall 2022

# 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?

```
1.   void thread_yield() {
2.     TCB *chosenTCB;
3.     disableInterrupts(); // why??
4.     chosenTCB = readyList.getNextThread();
5.     if (chosenTCB == NULL) {
6.       // Nothing to do here
7.     } else {
8.       runningThread->state = READY;
9.       readyList.add(runningThread);
10.      thread_switch(runningThread, chosenTCB);
11.      runningThread->state = RUNNING;
12.    }
13.    enableInterrupts();
14.  }

15.  void thread_switch(oldTCB, newTCB) {
16.    pushad;
17.    oldTCB->sp = %esp;
18.    %esp = newTCB->sp;
19.    popad;
20.    return;
21.  }
```

Mengwei Xu @ BUPT Fall 2022

# 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 (3$^{rd}$ 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 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

- Write a program to accelerate matrix multiplication through threads on your PC.
  - Summarize the performance improvement over one thread
  - Test with different sizes: 100×100, 1000×1000, 10000×10000, etc.

- Submit a pdf with your design, main code, and experiment results.
  - Up to 2 pages

$$
\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}
$$

# Jos Lab & Bos Lab

- Jos Lab
  - Jos的前4个Lab，可参考外部资料
  - 验收形式：实验报告、代码+面试
  - 提供Docker开发环境

- Bos Lab
  - Lab 1: 用户态线程创建
  - Lab 2: 多种调度策略
  - Lab 3: 同步机制
  - Lab 4: 内存管理
  - 验收形式：实验报告、含代码
  - 提供Docker开发环境

| Lab | Time |
| --- | --- |
| Lab 1 | 2022.09.23-2022.10.14 |
| Lab 2 | 2022.10.07-2022.11.04 |
| Lab 3 | 2022.10.28-2022.11.25 |
| Lab 4 | 2022.11.18-2022.12.16 |

- Lab 1: 用户态线程创建
  - 实验要求
    - 提供创建用户态线程的机制
    - 上下文切换
    - 线程状态的管理
    - random调度策略
    - 测试程序
  - 验收形式：
    - 代码和说明文档
    - 提交格式在课程主页上具体说明