CS 475
Operating Systems

UNIVERSITY of PUGET SOUND
Est. 1888

Department of Mathematics
and Computer Science

Lecture 4
Threads and
Parallel Computing

Motivation 1: Process Responsiveness

- Consider the following C program:
  - First ... I want to compute π
  - Then I need to print a funny email I got from America to show students

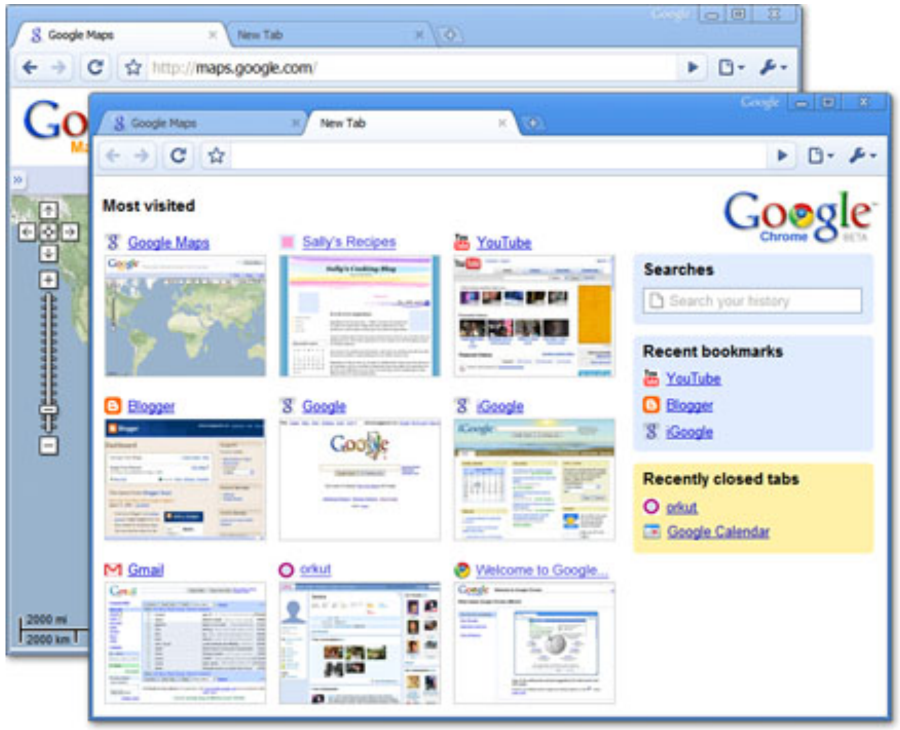
```
int main(int argc, char *argv[]) {
    computePi();
    printAmericasFunnyEmail();
    return 0;
}
```

- Two logically *independent* sub-tasks, but... second task is "stuck" behind the first.
  - `computePI()` will never finish.
  - Even on a timesharing OS, it'll never print America's email.

Motivation 1: Process Responsiveness (2)

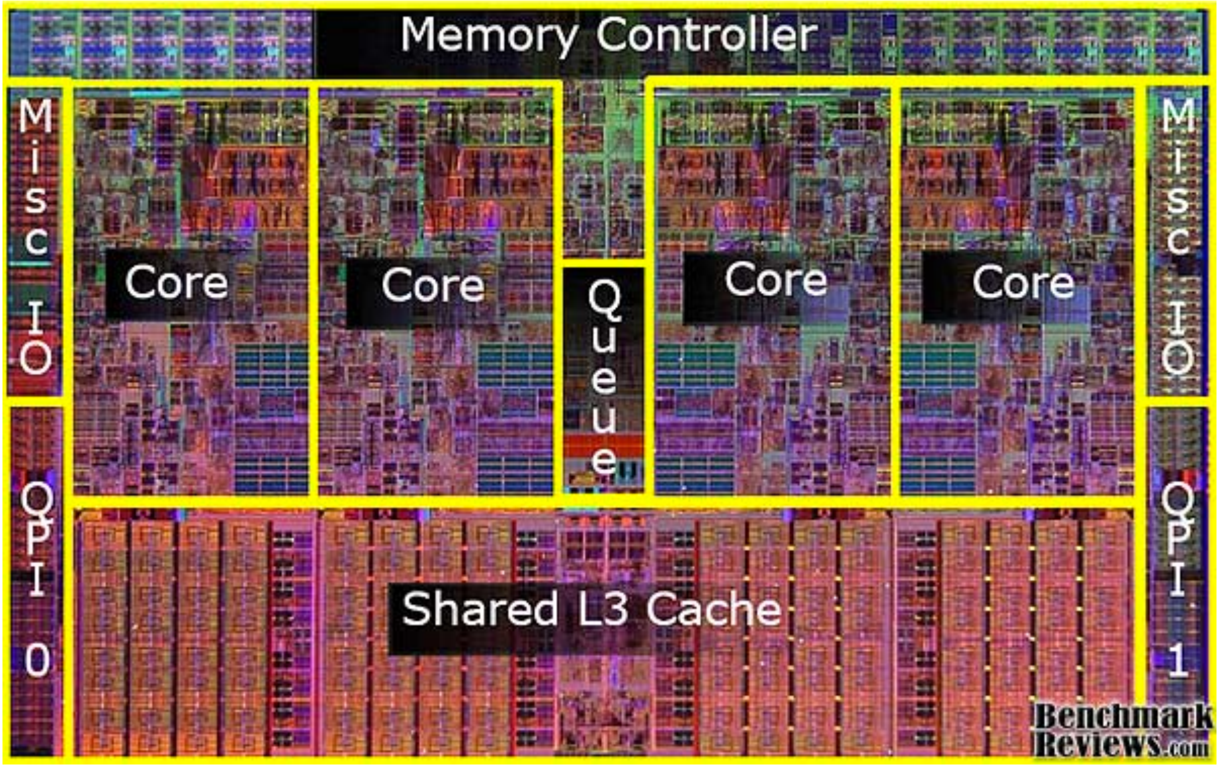
- **Responsiveness:** A single program may need to multitask!

- For instance, web browsers:
  - Downloads an HTML page, it accepts user input, fetches from cache, renders page, ... all concurrently!
  - In a single-threaded execution, these actions would have to be done sequentially.
    - If browser gets stuck downloading large file, it has to block!



Motivation 2: Parallel Processing

- **Problem 1: Increase CPU utilization in the multicore era**
  - Programmers today need to extract parallelism for high performance
    - Can we fetch and load 4 TikTok videos simultaneously?



But can't processes utilize different cores?

Yes, generally if there's a free core, OS will schedule the process on it!

Then why do we need threads?

Process-Level Parallelism Is Possible, but...

- We'll just fork a new process for each task. (This *would* work!)

```
int main(int argc, char *argv[]) {
   if (0 != fork()) {
     computePi();   // Let the parent compute PI
   }
   else {
     printAmericasFunnyEmail();  // Let the child print email
   }
}
```

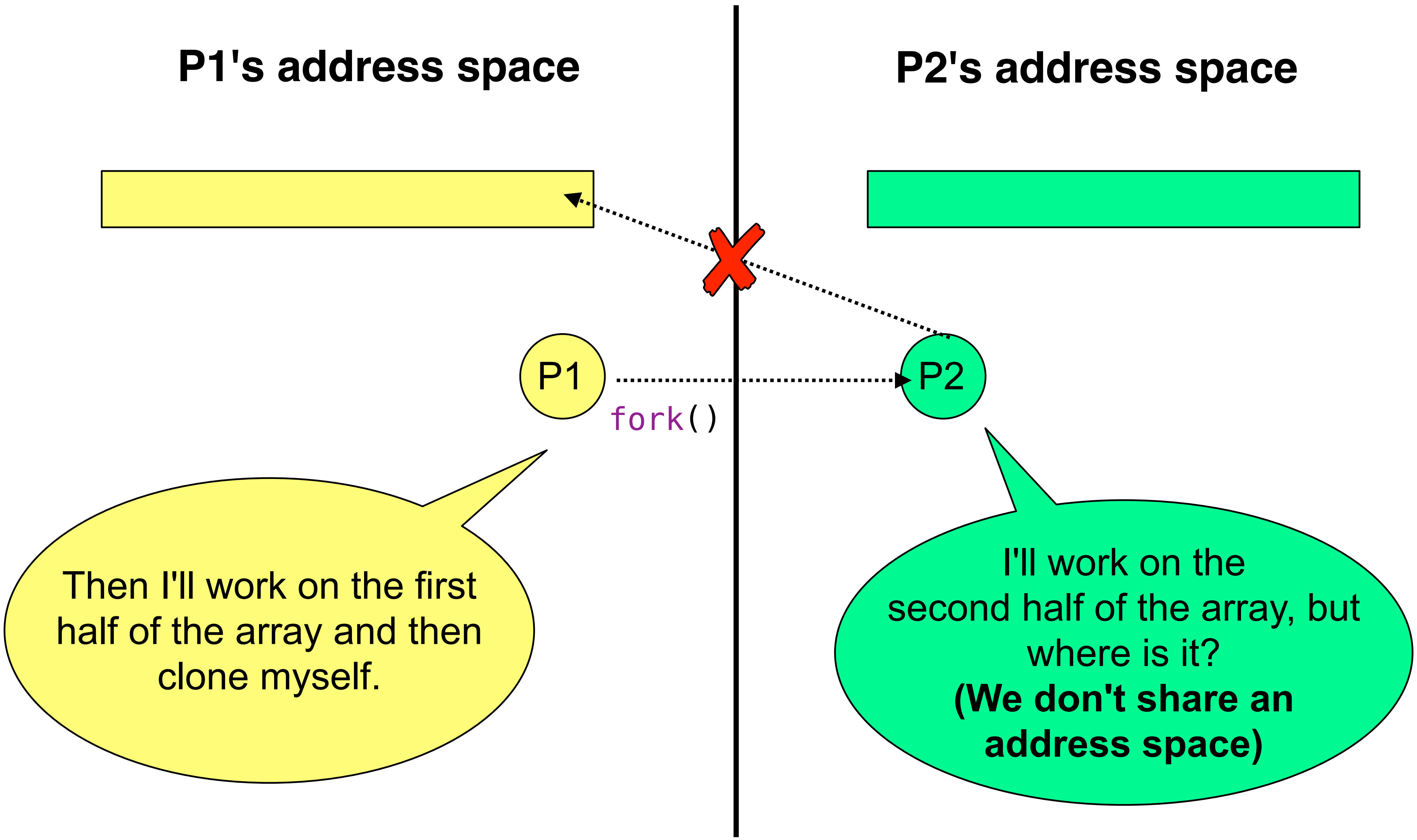
- Sure, but process creation is expensive 🤑 🤑 🤑!

```
Recall to create a new process using fork():

1) fork() is privileged, so you need trap a system call.
2) OS allocates a PCB for the new child
3) OS allocates child's address space.
4) OS copies address space from parent.
5) OS copies I/O state from parent.
6) Now the new child process can run
```

Process-Level Parallelism Is Possible, but...

- **Problem 2:** Often, processes need to *share* data, but how? (It can be done using the `mmap()` syscall, but it's slow and complicated to setup).

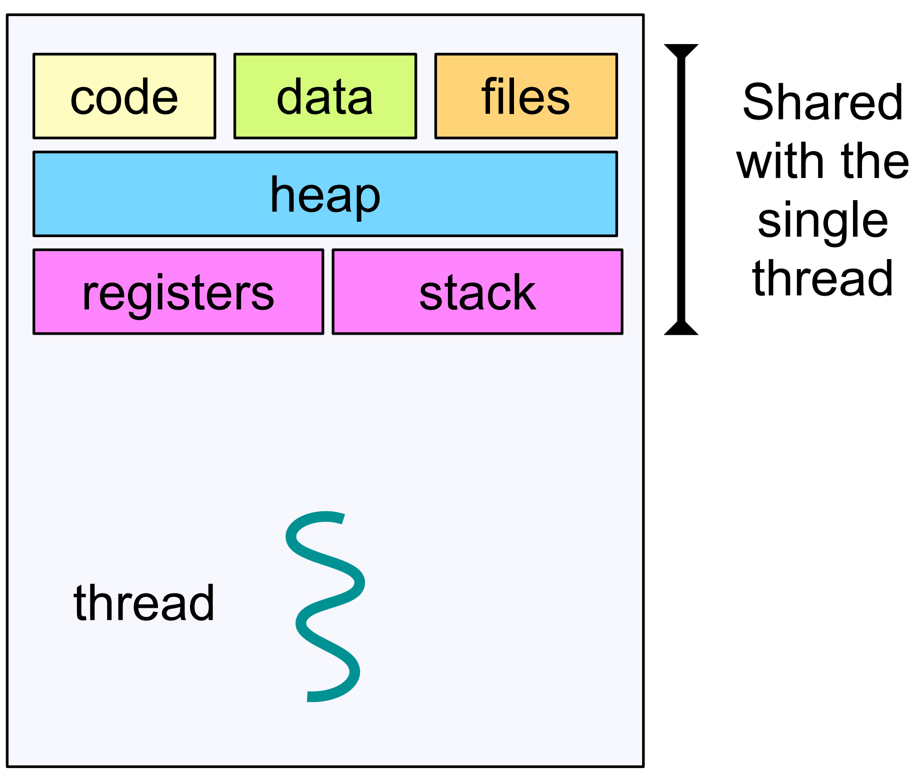


Goals for Today...

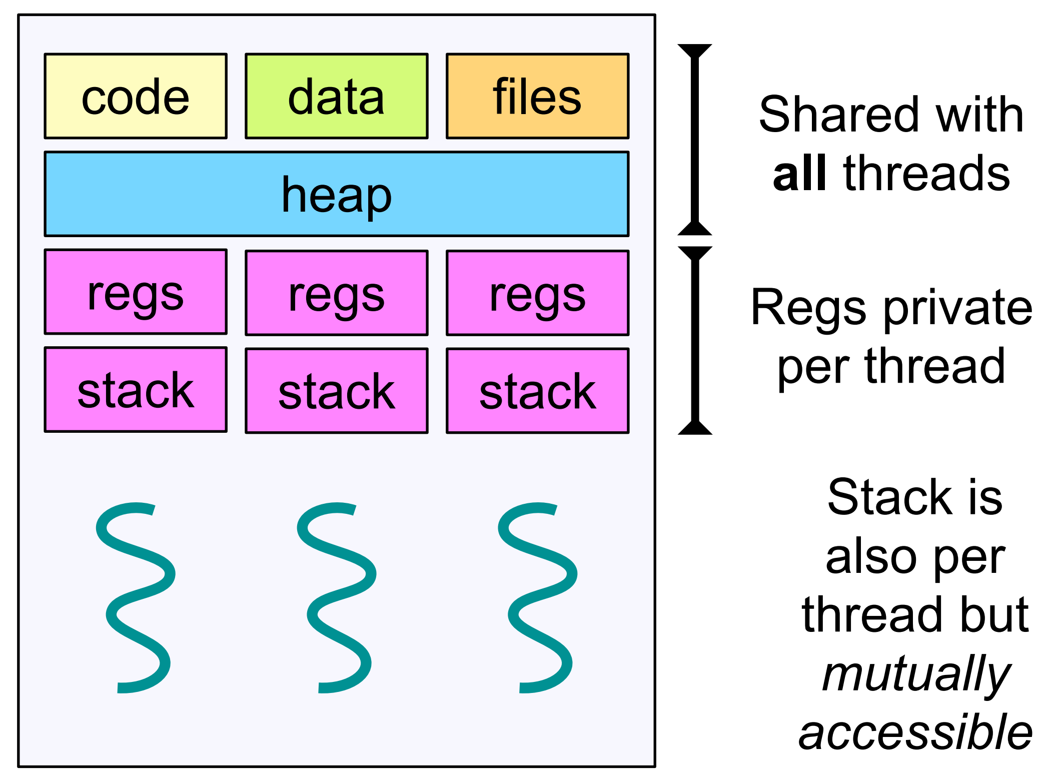
- Threads
- Implementing Threads
- Pthread Examples in C
- Parallel Processing Paradigms
- User Threads vs. Kernel Threads
- Conclusion

Definition: Thread of Execution

- A "*thread*" is an independent instruction stream *inside* a process
  - The process *still* contains a single address space
    - Multiple threads all share this address space
  - Every thread must be <u>bound</u> to an existing process
  - Threads, not processes, are the new units of execution



Single-Threaded Process (What we're used to)



Multi-Threaded Process

Thread Control Block (TCB)

- If threads are the new units of execution... PCBs are no longer sufficient!
- **Idea:** Split the PCB into two parts
  - Piece 1: We still have a PCB, but it's stripped down to hold only the shared process state.
    - Process ID (pid), pointer to address space, open files, sockets
  - Piece 2: *TCBs* hold the data private to each thread (1 TCB per thread)
    - A thread ID (tid)
    - Thread state: CPU regs, SP, BP, PC belonging to that thread
    - Scheduling info: running state, priority of that thread
    - Pointer back to its parent PCB

Context Switching between Threads

- Context switches now have two possibilities:
  - Switching two threads in **different** processes. **Slow as before.**
    - Same as switching processes (i.e., it's just like before)
    - Save and restore state of two PCBs (and by extension the two TCBs)
    - **Big deal: Address spaces of 2 processes must be swapped!**
  - Switching two threads in the **same** process. **Faster!**
    - Only have to switch TCBs within same process. No need to save/restore PCB state.
    - The TCB has less state that has to be saved and restored
    - **Big deal: Don't have to switch the address space!**

Context Switching Threads/Processes (2)

```
void contexSwitch(TCB *currentTCB, TCB *nextTCB) {

    /* save state of current TCB */
    currentTCB->r0 = CPU[r0];
    currentTCB->r1 = CPU[r1];
    currentTCB->r2 = CPU[r2];
    //...
    currentTCB->sp = CPU[sp];  // save stack pointer
    currentTCB->pc = CPU[pc];  // save program counter


    /* we skip exchanging addr space if threads are in the same process */
    if (currentTCB->parentPCB != newTCB->parentPCB) {
        copyToDisk(currentTCB->parentPCB->addrsp);
        copyFromDisk(nextTCB->parentPCB->addrsp);
    }


    /* restore state of next TCB */
    CPU[r0] = nextTCB->r0;
    CPU[r1] = nextTCB->r1;
    CPU[r2] = nextTCB->r2;
    //...
    CPU[sp] = nextTCB->sp;
    CPU[pc] = nextTCB->pc;
}
```

Advantages of Threads

- Advantages of threads
  - Much cheaper to use compared to processes!
    - Thread creation time is tiny compared to `fork()`
      - No need to allocate another address space!
      - Don't need an entire PCB either. Just create a new lighter-weight TCB.
    - Reduced time required to context-switch between threads within *same* process
  - Can improve the responsiveness of programs
  - *Automatic* shared memory (Heap, Data, and Code segments)!
  - *Parallelism:* Scales to multicore CPUs
    - Each thread of execution can be dispatched to an available CPU core

Disadvantages of Threads

- Disadvantages:
  - Synchronizing threads is hard.
    - We have a whole book chapter on it!
  - You can monopolize the CPU by just creating more threads.
    - More threads means that more "portions"of a process are in the OS queues!
  - Overuse of threads isn't always a good thing.
    - Diminishing returns of parallel programs. Would 1000 threads sort a list significantly faster than 4?

Goals for Today...

- Threads
- Implementing Threads
- Pthread Examples in C
- Parallel Processing Paradigms
- User Threads vs. Kernel Threads
- Conclusion

First, Define the Thread's Work Function

- What does each thread do? Define its routine using this template:
  - Note that `void*` is a pointer to a generic
    - (You have to *cast* it later to make sense of it)

```c
/**
 * Defines a single thread's routine.
 *
 * @param *arg Generic pointer to some data or structure (can be NULL)
 * @return a generic pointer to some data or structure (can be NULL)
 */
void* routine(void* arg) {

    // 1. if arg != NULL, cast *arg in order to understand its input
    // 2. Do some work.
    // 3. return a pointer to the data (or NULL)
}
```

pthread Library: Creating a Thread

- To use threads in C, `#include <pthread.h>` in your program.

```
int pthread_create(
  pthread_t      *tid,
  pthread_attr_t *attr,
  void*          *routine(void *), // what the? It's a function ptr
  void           *arg);
```

`@param *tid`: A pointer to the new thread's unique ID

`@param *attr`: Set of attributes for the new thread, use `NULL` for default attributes

`@param *routine`: The new thread starts by invoking `routine()`

`@param *arg`: Pointer to argument passed to `routine()`

`@return` 0 if successful, otherwise an error number

pthread Library: Terminating a Thread

UNIVERSITY of PUGET SOUND Est. 1888

- Sometimes we may want to force a thread to terminate.

```
int pthread_cancel(pthread_t tid)
```

`@param tid`: The target thread ID

`@return` 0 if successful, otherwise an error number

- Two cancellation protocols:
  - *Asynchronous:* target thread is immediately terminated
    - What if it was in the middle of updating a shared structure?
  - *Deferred:* target thread periodically checks whether there is a termination request
    - If so, wrap up any critical work and self-terminate (**pthread** does this)

pthread Library: Terminating a Thread (2)

UNIVERSITY of PUGET SOUND

- Deferred Cancellation
  - Threads check for cancellation request at various cancellation points.

- *Cancellation Points* are predefined locations where it is deemed safe to terminate a thread, such as:
  - Blocking on an I/O
  - During sleep
    - A list of cancellation points are found here: https://man7.org/linux/man-pages/man7/pthreads.7.html
  - or at an explicitly defined cancellation points in a thread's code
    - https://man7.org/linux/man-pages/man3/pthread_testcancel.3.html

pthread Library: Thread ID

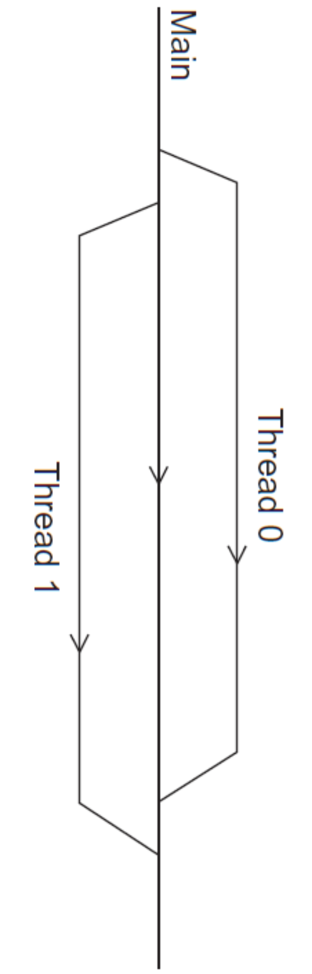
- To get the calling thread's ID
  - Usually some random non-negative value
  - Don't just assume it's 0, 1, 2, ...

```
pthread_t pthread_self()
```

`@return` thread's identifier. This function always succeeds.

pthread Library: Reaping a Thread

- The calling thread waits for thread `tid` to terminate
  - When a dead thread is reaped, its resources are reclaimed
    - *e.g.*, TCB, thread's stack, and register contents




```
int pthread_join(
    pthread_t tid,
    void **thread_return
)
```

`@param tid` The thread ID

`@param **thread_return` Pointer to the thread's return pointer.

`@return` 0 if successful, otherwise an error number

Passing a **Single Argument** to a Thread

```
void* print_msg(void *arg) {
    char *msg = (char*) arg;  // Important: cast the input argument!
    printf("%s\n", msg);
    return NULL;
}

int main(int argc, char* argv[]) {
    // Important: put it on the heap so it's shared!
    char *message = (char*) malloc(10 * sizeof(char));
    strcpy(message, "Hello!!!");

    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, print_msg, message);
    pthread_create(&thread2, NULL, print_msg, message);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

Output: (Note that we're not sure *which* thread printed first)

```
Hello!!!
Hello!!!
```

Passing **Multiple Arguments** to a Thread

```c
/** Multiple args? Declare a struct to encapsulate input arguments a thread */
typedef struct arg_t {
    int  num;
    char *msg;
} arg_t;

/** thread routine */
void* print_msg(void *arg) {
    // cast the input argument!
    arg_t* input = (arg_t*) arg;
    for (int i = 0; i < input->num; i++) {
        printf("%s\n", input->msg);
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    // allocate and populate the input struct on the heap!
    arg_t *params = (arg_t*) malloc(sizeof(arg_t));
    params->num = 500;
    params->msg = (char*) malloc(sizeof(char) * 10)
    strcpy(params->msg, "Hello!!!");

    pthread_t thread;
    pthread_create(&thread, NULL, print_msg, params);
    pthread_join(thread, NULL);
}
```

Reading Thread's `return` Value

```c
/** thread routine */
void* work(void *arg) {
    int *val = (int*) malloc(sizeof(int));
    *val = 100;
    return val;
}

int main(int argc, char* argv[]) {
    pthread_t thread;
    pthread_create(&thread, NULL, work, NULL);

    // thread return value held in a void* pointer
    void *returned;
    pthread_join(thread, &returned);

    //cast void* to an int*, then de-reference
    int actual = *((int*) returned);
    printf("Here was what the thread returned: %d\n", actual);
}
```

Output:

```
Here was what the thread returned: 100
```

Why Is This `work()` Code Broken?

```
/** thread routine */
void* work(void *arg) {
    int val = 100;
    return &val;
}
```

val is on the thread's stack, and stacks are private per thread. **val**'s address is inaccessible from another thread (**main**).

```
int main(int argc, char* argv[]) {
    pthread_t thread;
    pthread_create(&thread, NULL, work, NULL);

    // thread return value held in a void* pointer
    void *returned;
    pthread_join(thread, &returned);

    //cast void* to an int*, then dereference
    int actual = *((int*) returned);
    printf("Here was what the thread returned %d\n", actual);
}
```

*Dereferencing **val** leads to segfault!*

Output:

Segmentation fault

What's Going On?

```c
int x = 1; // global (shared)

void* print_msg(void *arg) {
    x++;      // increment x!
    if (x == 2) {
        printf("Thread %lu: I win!\n", pthread_self());
    }
    else {
        printf("Thread %lu: I lost!\n", pthread_self());
    }
    return NULL;
}
```

Code that threads will run

```c
int main(int argc, char* argv[]) {
    /** fork phase **/
    printf("Begin threads\n");
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, print_msg, NULL);
    pthread_create(&thread2, NULL, print_msg, NULL);

    /** join phase **/
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

printMsg.c

```
Begin threads
Thread 1: I Win
Thread 2: I lost
```

(Blue won the race!)

```
Begin threads
Thread 2: I Win
Thread 1: I lost
```

(Pink won the race!)

```
Begin threads
Thread 1: I lost
Thread 2: I lost
```

(What the? Yes it can happen)

Goals for Today...

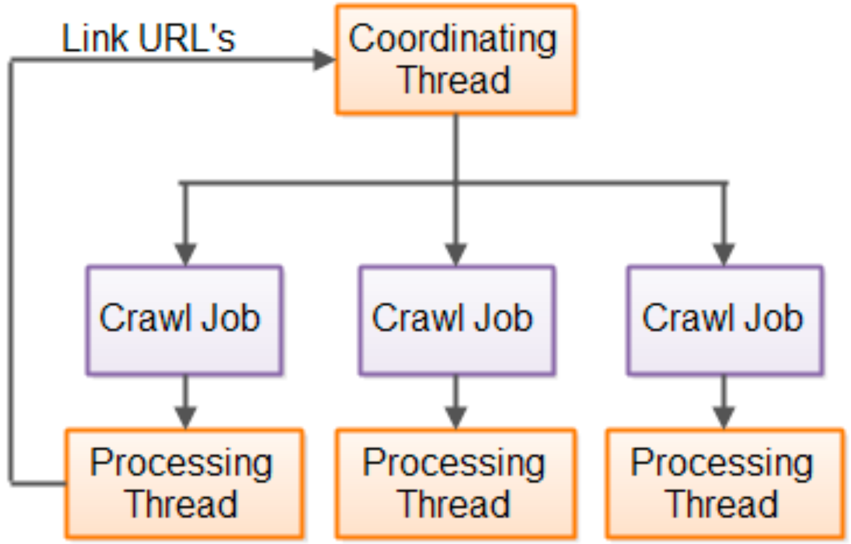
- Threads
- Implementing Threads
- Pthread Examples in C
- Parallel Processing Paradigms
- User Threads vs. Kernel Threads
- Conclusion

Two Approaches to Parallel Computing

- *Task Parallelism* when the problem can be decomposed into subtasks. Each thread works on an independent subtask.
  - Git
    - To push all the files in your git repository onto Github
    - Create a thread to upload each file.

Two Approaches to Parallel Computing

- *Task Parallelism* when the problem can be decomposed into subtasks. Each thread works on an independent subtask.
  - Git
    - To push all the files in your git repository onto Github
    - Create a thread to upload each file.
  - Web Crawler
    - To discover all the webpages out there, start with a random page
    - Create a new thread to retrieve each link found on the page. Repeat.



Two Approaches to Parallel Computing

- *Data Parallelism:* when the data set can be decomposed into smaller chunks. Each thread works on a subset of data.
  - Sort a massive list:
    - View the list as multiple sublists.
    - Threads sort each sublist.
    - Main thread merges all sublists afterwards.
  - Sudoku game validation:
    - View the game board as multiple columns, rows, and 3x3 boxes
    - Threads validate each column, row, box
    - If none of the threads return false, the game is valid!

7	3	5	6	1	4	8	9	2
8	4	2	9	7	3	5	6	1
9	6	1	2	8	5	3	7	4
2	8	6	3	4	9	1	5	7
4	1	3	8	5	7	9	2	6
5	7	9	1	2	6	4	3	8
1	5	7	4	9	2	6	8	3
6	9	4	7	3	8	2	1	5
3	2	8	5	6	1	7	4	9

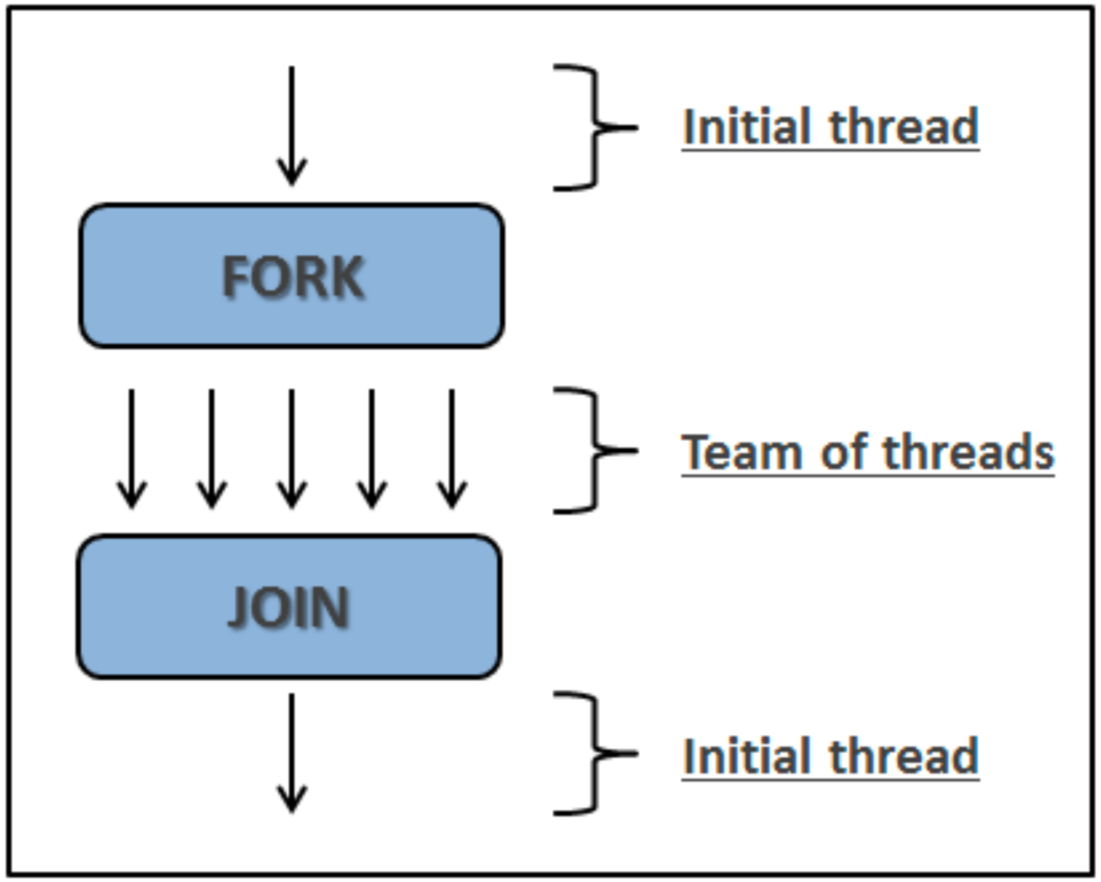
Two Approaches to Parallel Computing

- *Another Data Parallel example:*
  - Matrix multiplication
    - Two input matrices A, B, and a result matrix C are in shared space.
    - Threads calculate the product(s) defined for their "region."
      - (Maybe each thread calculates a chunk of rows in C? Up to programmer)
    - Main thread simply waits for all worker threads to finish, and exits.

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

Fork-Join Paradigm of Parallel Programming

- *In both approaches, the Fork-Join approach can be applied.*
  - Given a problem of size n
  - Main thread creates m worker threads
    - Each worker does some fraction of work of the same nature
    - Each worker might derive some partial result
  - Main thread waits for <u>all</u> threads to finish and reduces partial results to a final one.



Example: Parallel Count Odds

- Let's start with a small ***Data Parallel*** example:
  - Count up all the odd numbers in a given array of 100,000 numbers

- Fork-Join setup:
  - Malloc the array (so it's shared).
  - Assign the array with random numbers.
  - **Fork:** Create 2 threads to count the first and second halves of the array.
  - **Join:** Add up both counts and print.

Parallel Count Odds (Preamble)

- Preamble code area

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

const int N = 1000000;
int *array; // points to the array of ints on the heap
int *results;
```

Parallel Count Odds (Two Threads)

```
void* worker(void *param) {
    int *whoAmI = (int*) param; // either thread 0 or 1

    int begin = (*whoAmI % 2 == 0) ? 0 : N/2;
    int end =   (*whoAmI % 2 == 0) ? N/2 : N;
    printf("thread %d start at [%d] and end at [%d]\n", *whoAmI, begin, end);

    int count = 0;
    for (int i = begin; i < end; i++) {
        if (array[i] % 2 == 1) {
            count++;
        }
    }
    results[*whoAmI] = count;
    free(param);
    return NULL;
}
```

Cast the parameter to make sense of it! (It's just a thread ID)

Parallel Count Odds (Two Threads)

```c
void* worker(void *param) {
    int *whoAmI = (int*) param; // either thread 0 or 1

    int begin = (*whoAmI % 2 == 0) ? 0 : N/2;
    int end =   (*whoAmI % 2 == 0) ? N/2 : N;
    printf("thread %d start at [%d] and end at [%d]\n", *whoAmI, begin, end);

    int count = 0;
    for (int i = begin; i < end; i++) {
        if (array[i] % 2 == 1) {
            count++;
        }
    }
    results[*whoAmI] = count;
    free(param);
    return NULL;
}
```

Depending on which thread I am (0 or 1), count the 1st or 2nd half of the array.

Parallel Count Odds (Two Threads)

```c
void* worker(void *param) {
    int *whoAmI = (int*) param; // either thread 0 or 1

    int begin = (*whoAmI % 2 == 0) ? 0 : N/2;
    int end =   (*whoAmI % 2 == 0) ? N/2 : N;
    printf("thread %d start at [%d] and end at [%d]\n", *whoAmI, begin, end);

    int count = 0;
    for (int i = begin; i < end; i++) {
        if (array[i] % 2 == 1) {
            count++;
        }
    }
    results[*whoAmI] = count;
    free(param);
    return NULL;
}
```

Work on my just half!

Parallel Count Odds (Two Threads)

```
void* worker(void *param) {
    int *whoAmI = (int*) param; // either thread 0 or 1

    int begin = (*whoAmI % 2 == 0) ? 0 : N/2;
    int end =   (*whoAmI % 2 == 0) ? N/2 : N;
    printf("thread %d start at [%d] and end at [%d]\n", *whoAmI, begin, end);

    int count = 0;
    for (int i = begin; i < end; i++) {
        if (array[i] % 2 == 1) {
            count++;
        }
    }
    results[*whoAmI] = count;
    free(param);
    return NULL;
}
```

Each thread deposits their counts in the corresponding spot.

Parallel Count Odds (main function)

```
int main(int argc, char *argv[]) {
    srand(0);    // "seed" the random number generator
    array = (int*) malloc(sizeof(int)*N);
    for (int i = 0; i < N; i++) {
        array[i] = rand();
    }

    // malloc an array for threads to deposit their results
    results = (int*) malloc(sizeof(int) * 2);

    // spin up the two worker threads!
    pthread_t tid[2];
    for (int i = 0; i < 2; i++) {
        int *threadID = (int*) malloc(sizeof(int));
        *threadID = i;
        pthread_create(&tid[i], NULL, worker, threadID);
    }

    // wait for all the threads to finish
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        pthread_join(tid[i], NULL);
        sum += results[i];
    }
    printf("Total count: %d\n", sum);
    free(results);
    free(array);
    return 0;
}
```

Allocate the array on the heap. Then populate it with random nums

Parallel Count Odds (main function)

```c
int main(int argc, char *argv[]) {
    srand(0);    // "seed" the random number generator
    array = (int*) malloc(sizeof(int)*N);
    for (int i = 0; i < N; i++) {
        array[i] = rand();
    }

    // malloc an array for threads to deposit their results
    results = (int*) malloc(sizeof(int) * 2);

    // spin up the two worker threads!
    pthread_t tid[2];
    for (int i = 0; i < 2; i++) {
        int *threadID = (int*) malloc(sizeof(int));
        *threadID = i;
        pthread_create(&tid[i], NULL, worker, threadID);
    }

    // wait for all the threads to finish
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        pthread_join(tid[i], NULL);
        sum += results[i];
    }
    printf("Total count: %d\n", sum);
    free(results);
    free(array);
    return 0;
}
```

Allocate the array on the heap. Then populate it.

`results[..]` ***will store each threads' partial counts.***

Prepare thread inputs: Just their IDs. Create both threads.

Parallel Count Odds (main function)

```c
int main(int argc, char *argv[]) {
    srand(0);    // "seed" the random number generator
    array = (int*) malloc(sizeof(int)*N);
    for (int i = 0; i < N; i++) {
        array[i] = rand();
    }

    // malloc an array for threads to deposit their results
    results = (int*) malloc(sizeof(int) * 2);

    // spin up the two worker threads!
    pthread_t tid[2];
    for (int i = 0; i < 2; i++) {
        int *threadID = (int*) malloc(sizeof(int));
        *threadID = i;
        pthread_create(&tid[i], NULL, worker, threadID);
    }

    // wait for all the threads to finish
    int sum = 0;
    for (int i = 0; i < 2; i++) {
        pthread_join(tid[i], NULL);
        sum += results[i];
    }
    printf("Total count: %d\n", sum);
    free(results);
    free(array);
    return 0;
}
```

Allocate the array on the heap. Then populate it.

`results[..]` ***will store each threads' partial counts.***

Prepare thread inputs: Just their IDs. Create both threads.

Join the threads, add up their partial counts, which are stored in `results[..]`

Amdahl's Law

- 1967 landmark keynote speech
  - *"Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities."* 1967 AFIPS Spring Joint Computer Conference (SJCC'67). [link here]

- How to express "parallel speedup?"
  - Diminishing returns of parallel architectures

$$Speedup = \frac{1}{(1-f) + \frac{f}{n}}$$

f = fraction of code that can be parallelized

n = degree of parallelization (threads/CPU Cores)

Gene Amdahl

Example: Diminishing Returns.

- Suppose 40% of your code can be parallelized by a factor of 8. What's the overall speedup of the parallel version?

$$Speedup = \frac{1}{(1-f)+\frac{f}{n}} = \frac{1}{(1-0.4)+\frac{0.4}{8}} = 1.54 = 54\,\%$$

  - (Yikes that's no where near an 8x (= 800%) speedup!)

- What if we had a 16 core machine? That's twice the number of cores!

$$Speedup = \frac{1}{(1-0.4)+\frac{0.4}{16}} = 1.6 = 60\,\%$$

Reality: Opposing Laws

- **Question:** Spend lots of time + money investing in parallel computing?
  - *Amdahl's Law says:* "No, extracting parallelism is hard!"
    - Diminishing returns! Error prone!
    - Amount of parallelism must overcome thread-management overhead.
  - *Moore's Law:*
    - Pre-2007: Amdahl is right... just wait for the faster processor to come out
    - After-2007: Processors getting slower, but we now have more cores.
      - Parallel computing becomes a necessity for current generation of coders
      - (ACM ugrad curriculum now calls for parallel computing as part of "Core CS")

Checklist for Making Parallel Programs

- Decide how to split up the work, so that each thread:
  - Performs the same nature of work, but works on a different "portion"
  - Write that function using
    - What info does each thread need to know?
    - (For instance, a pointer to an array? The start and end index of a portion?)

- Back to **main**():
  - Where do threads put their results?
    - Do they return it? Do they write it to place in the heap?
  - Prepares the input data for each thread, creates threads, joins them up
  - What to do with threads' results?

Your Turn!

- Some exercises you can try
  - Parallel sort algorithm
    - Split list into N segments
    - Each segment is sorted (insertion sort) by a different thread
    - Merge all sorted sublists into final sorted list
    - **Solution:** `https://github.com/davidtchiu/cs475-parInsertionSort`
  - Parallel search algorithm
    - Split list into N parts, assign each thread to search a different chunk of the array
  - Parallel Riemann Sum for computing integrals

Goals for Today...

- Threads
- Implementing Threads
- Pthread Examples in C
- Parallel Processing Paradigms
- User Threads vs. Kernel Threads
- Conclusion

User Threads

- In early systems, the OS only knew about processes.
  - Processes were a single execution stream
  - But users needed intra-process responsiveness:
    - Like, `computePI(); printAmericasFunnyEmail();`
- *"User Threads"* are completely managed by the programmer in unprivileged user mode
  - The OS wouldn't even know that multiple threads exist!
  - Users had to implement "thread support functions" in a **user library**
    - `CreateThread(), SwitchThread(), JoinThread(), ...`

User Threads (Pros)

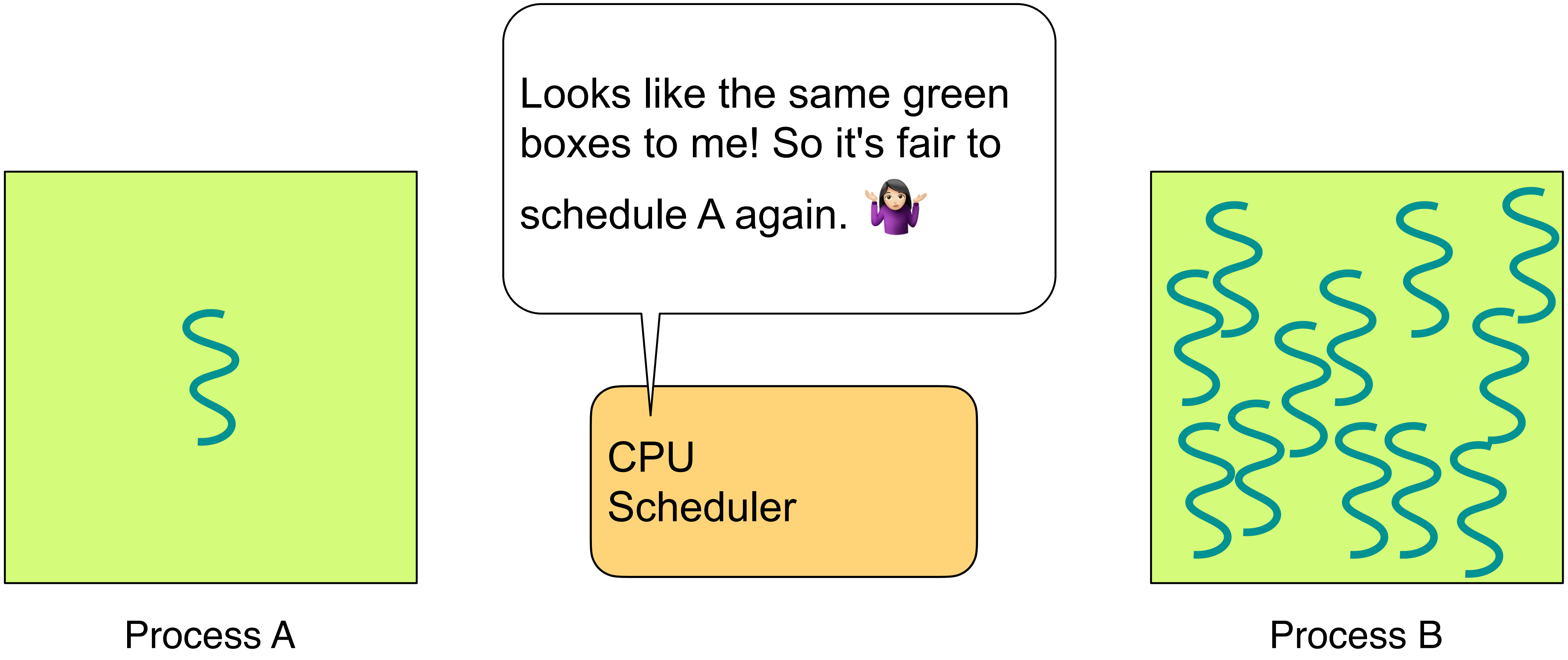
- Does not require any modification to the OS
  - Even if an OS doesn't support threads directly, a user can still write programs with threads.
- Thread operations (creation, termination, switching) are very fast
  - Orders of magnitude faster than switching processes
  - (Because the user code is doing the "switching," not the OS!)

User Threads (Cons)

- If one user thread blocks on I/O, *it stalls the entire process*
  - A thread makes a system call, all other threads wait with it! *(Why?)*
- Can't leverage parallel architectures
  - Without kernel support, the OS still dispatches jobs on the granularity of a process.
- No OS protection of private data
  - A user thread's stack & registers not private from another user thread
- Fairness in CPU Scheduling is now a concern
  - (Next slide)

User Threads (Cons)

- OS may not make *fair, insightful* decisions when scheduling
  - Say each process gets a fixed slice of CPU time
  - OS assigns same time slice, whether process has 1 thread or 1000!



Kernel Threads

- *Kernel Threads* (In use today)
  - OS *knows* about all threads in execution.
  - Thread management functions (creation, join, termination) done via system calls.
  - OS views kernel threads as schedule-able units and knows to context switch between them.

- All modern OS provides a **Thread API** to users to create and manage threads
  - In C, it's called the **pthread.h** library.

Kernel Threads (Cons)

- Creating, destroying threads would be system calls (slow)
- A context switch between two threads in same process is over an order of magnitude slower than in user threads!

"Null Fork" Times on VAX uniprocessor

User Thread	Kernel Thread	Processes
34 us	948 us	11,300 us

"NULL fork" means: Create, schedule, execute, complete a no-op

OS Examples

- The first OS's did not have native kernel thread support.
  - For responsiveness, you'd have to implement a user-thread library!
  - Fun fact: C's pthread library was originally a user library for thread support in early Unix systems

- Switch to kernel support in 1980-1990s
- All modern OS now support kernel threads.
- But some OS kept user threads around additionally!
  - (Example: Windows 10 "fibers")

OS Examples: Windows 10

- Windows 10 added support for fibers.
- What are *"fibers"?*
  - Fibers are user threads!
  - Users make a call to yield() CPU to another fiber.

Abstraction: Thread Assignment Models

- *N-to-1 Model*
  - Multiple user threads associated with one kernel thread
  - User-Level threads used this model
  - Examples: Xinu, early Unix, DOS

- *1-to-1* Model
  - One user-level thread is associated with one kernel thread
  - Most operating systems today: Linux, OS X, BSD, iOS

Goals for Today...

- Threads
- Implementing Threads
- Pthread Examples in C
- Parallel Processing Paradigms
- User Threads vs. Kernel Threads
- Conclusion

In Conclusion

- Why threads?
  - Processes are inefficient (in both space and time)
  - But programs need to multitask and share data
  - Threads are fast, and they are now the OS's unit of execution

```
while (1) {
    curTCB  = getRunThread();
    nextTCB = runScheduler();
    ctxsw(curTCB, nextTCB);
}
```

- So... how does `runScheduler()` work? *(Next Topic: CPU Scheduling)*

Administrivia 2/24

- Midterm exam Friday.
- Hwk 4 Monday 3/3.
- Last time... Finished process management
  - New topic: Limitations of processes (and why we need threads)
  - Implementation of threads
- Today: Read Chap 4.1-4.4 (Dinosaur book)
  - The POSIX thread (**pthread**) library for C
  - User threads vs. Kernel threads

Administrivia 3/3

- Hwk 4 due tonight!
- Hwk 5 posted, due Friday 3/14
- Last time...
  - Read Chap 4.1-4.4 (Dinosaur book)
  - The POSIX thread (**pthread**) library for C
- Today
  - Pthread example: Parallel Counter
  - Amdahl's Law on parallel speedup limitations
  - Where to implement threads?
    - User space vs. Kernel space