

Operating Systems

Prof. Yongkun Li

中科大-计算机学院 特任教授

<http://staff.ustc.edu.cn/~ykli>

Ch4 Threads

Chapter 4: Threads

- Thread Concepts
 - Why use threads
 - Structure in Memory
 - Benefits and Challenges
 - Thread Models
- Programming
 - Basic Programming: Pthreads Library
 - Implicit Threading: Thread Pools & OpenMP

Multi-threading

- Motivation

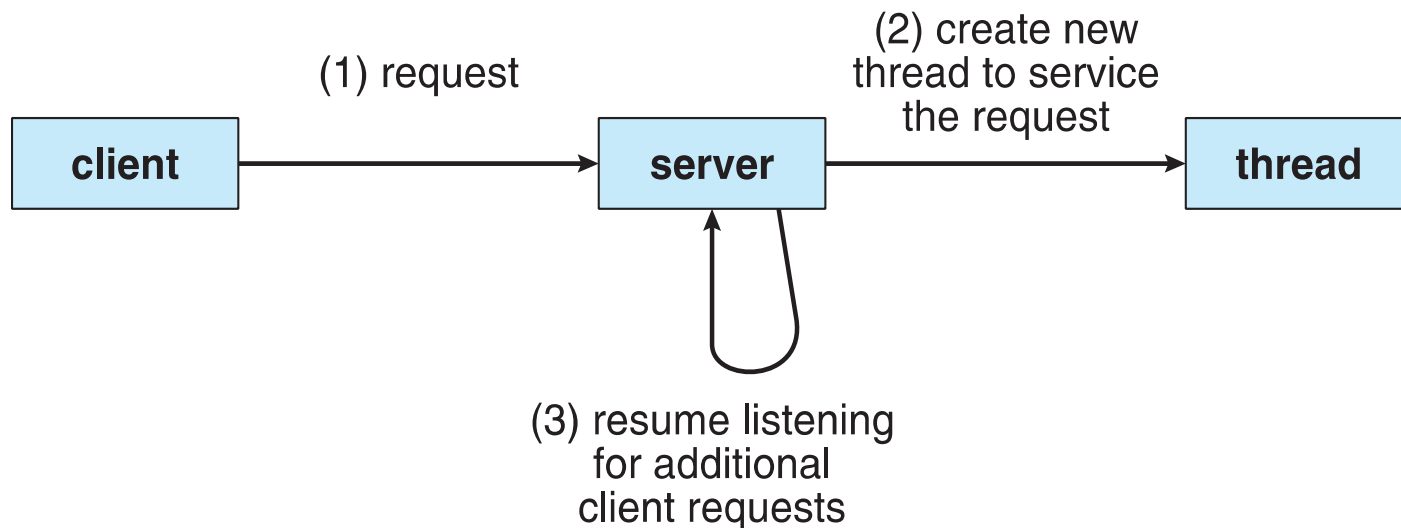


Motivation - Application Side

- Most software applications are multithreaded, each application is implemented as a process with several threads of control
 - Web browser
 - displays images, retrieve data from network
 - Word processor
 - display graphics, respond to keystrokes, spelling & grammar checking

Motivation - Application Side

- Most software applications are multithreaded
 - Web browser
 - Word processor
 - Similar tasks in a single application (web server)
 - Accept client requests, service the requests
 - Usually serve thousands of clients



Motivation – Application Side

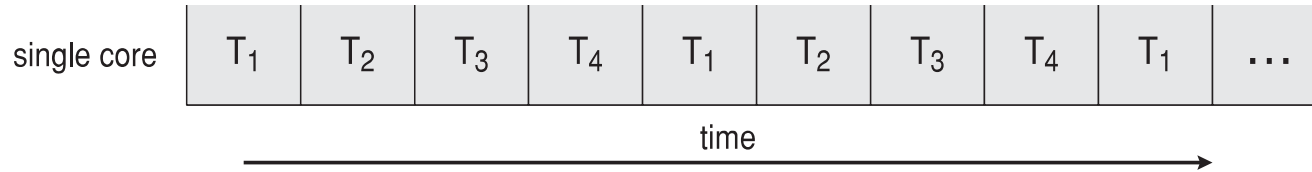
- Why not create a process for each task?
 - Process creation is
 - Heavy-weighted
 - Resource intensive
- Still remember what kinds of data are included in a process...
 - Text, data, stack, heap in user-space memory
 - PCB in kernel-space memory
- Many of the data can be shared between multiple tasks within an application

Motivation – System Side

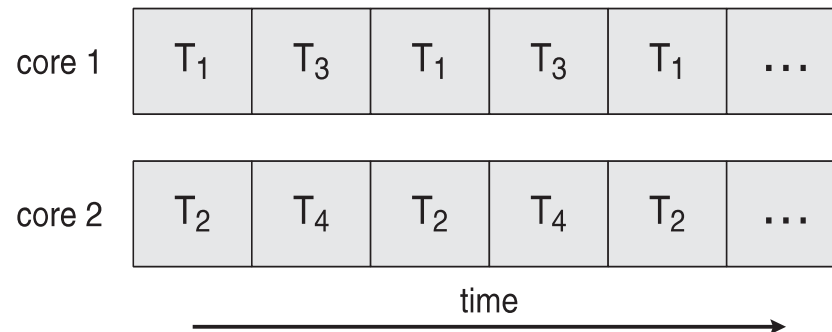
- Modern computers usually contain multicores
 - But, each processor can run only one process at a time
 - CPU is not fully utilized
- How to improve the efficiency?
 - Assign one task to each core
 - Real parallelism (not just concurrency with interleaving on single-core system)

Concurrency vs. Parallelism

■ Concurrent execution on single-core system:



■ Parallel execution on a multi-core system:

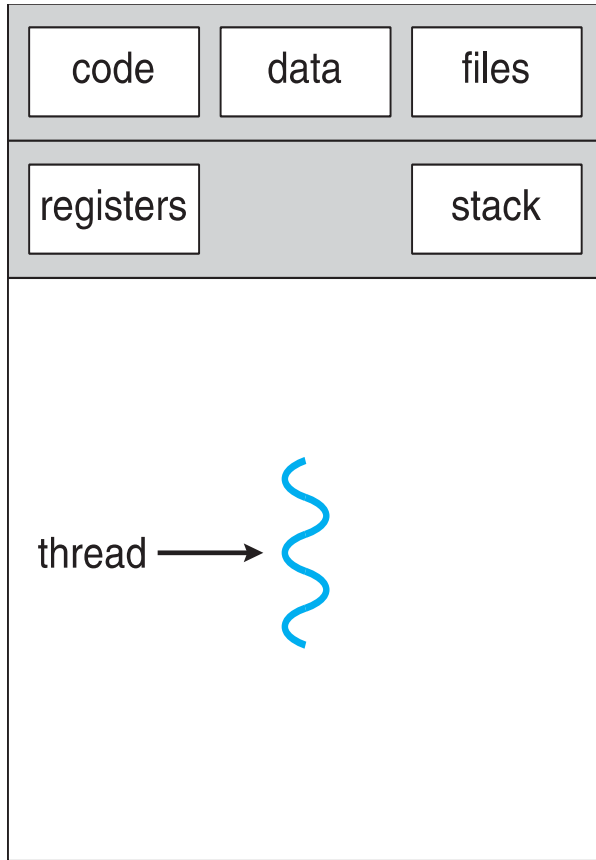


Multi-threading

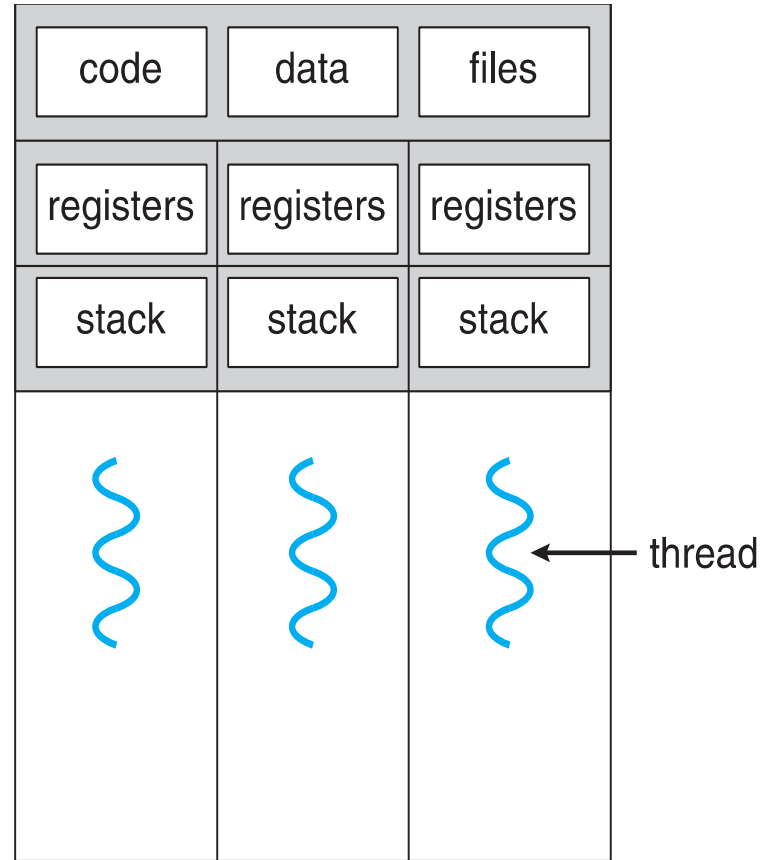
- Motivation
- Thread Concept



High-level Idea



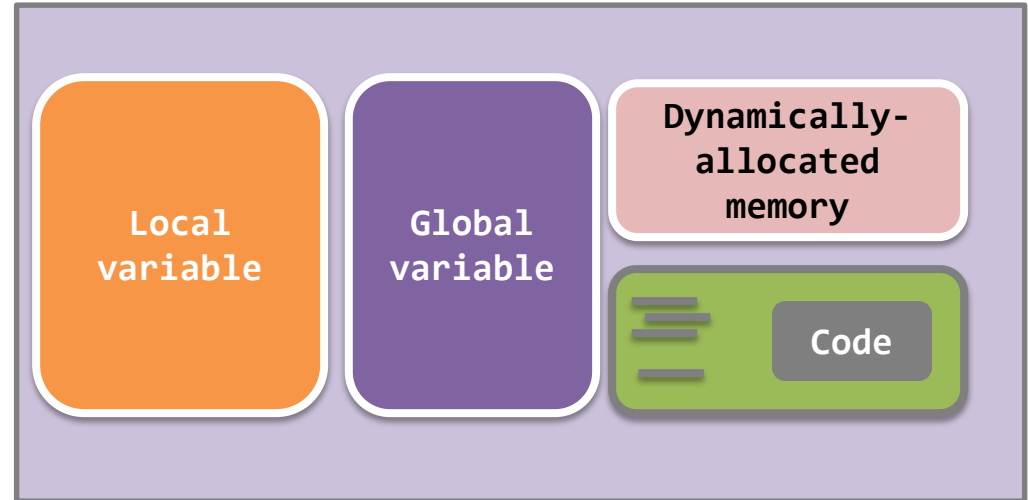
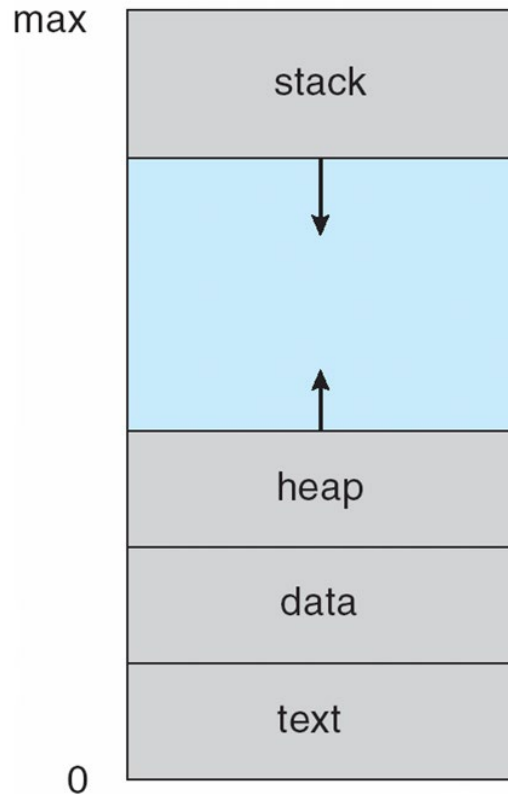
single-threaded process



multithreaded process

Recall: Process in Memory

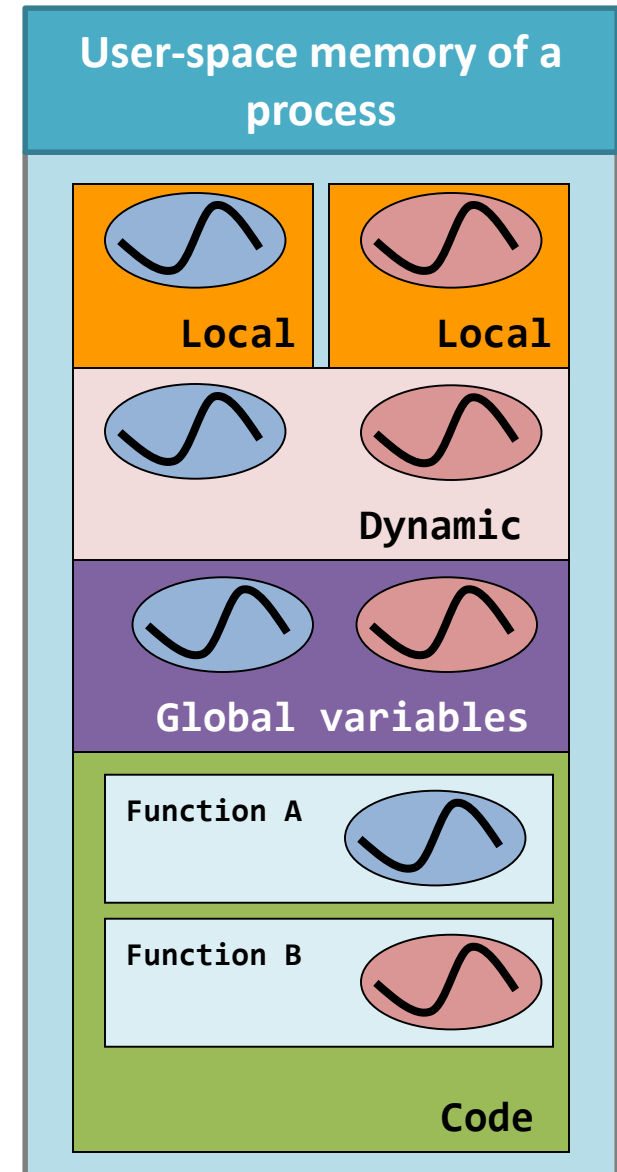
- User-space memory of Process A



Multi-thread – internals

Code

- All threads **share** the same code.
- A thread starts with **one specific function**.
 - We name it the **thread function**
 - Functions A & B in the diagram
- The thread function can invoke other functions or system calls
- But, a thread could never return to the caller of the thread function.

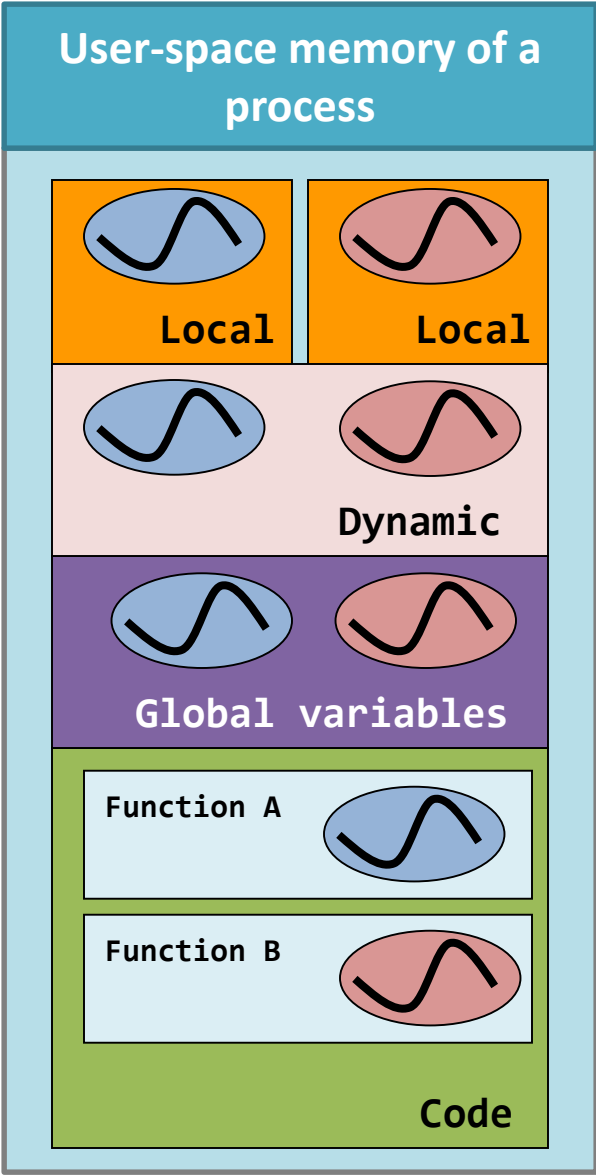


Multi-thread – internals

Dynamically allocated memory

Global variables

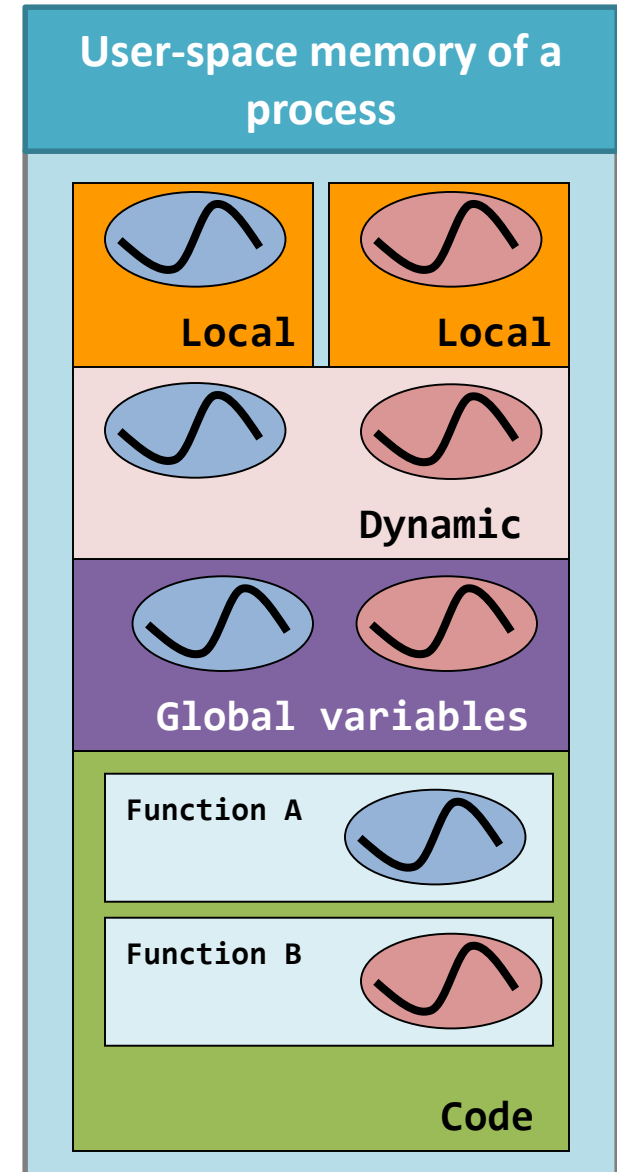
- All threads **share** the same global variable zone and the same dynamically allocated memory
- All threads can read from and write to both areas



Multi-thread – internals

Local variables

- Each thread has **its own memory range** for the local variables
- So, the stack is the private zone for each stack

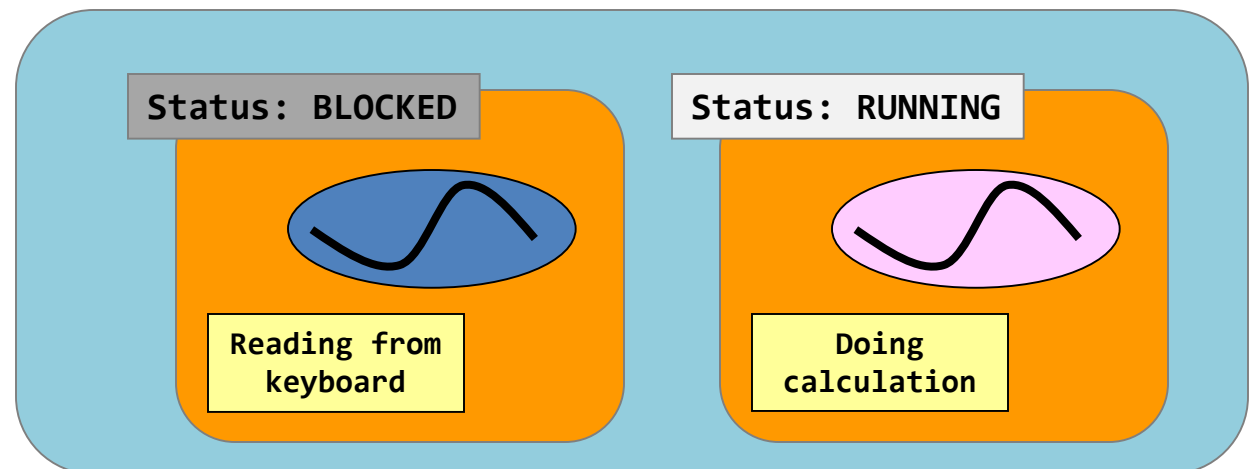


Benefits of Multi-thread

- **Responsiveness and multi-tasking**

- Multi-threading design allows an application to do **parallel tasks simultaneously**
- Example: Although a thread is blocked, the process can still depend on another thread to do other things!
- Especially important for interactive applications (user interface)

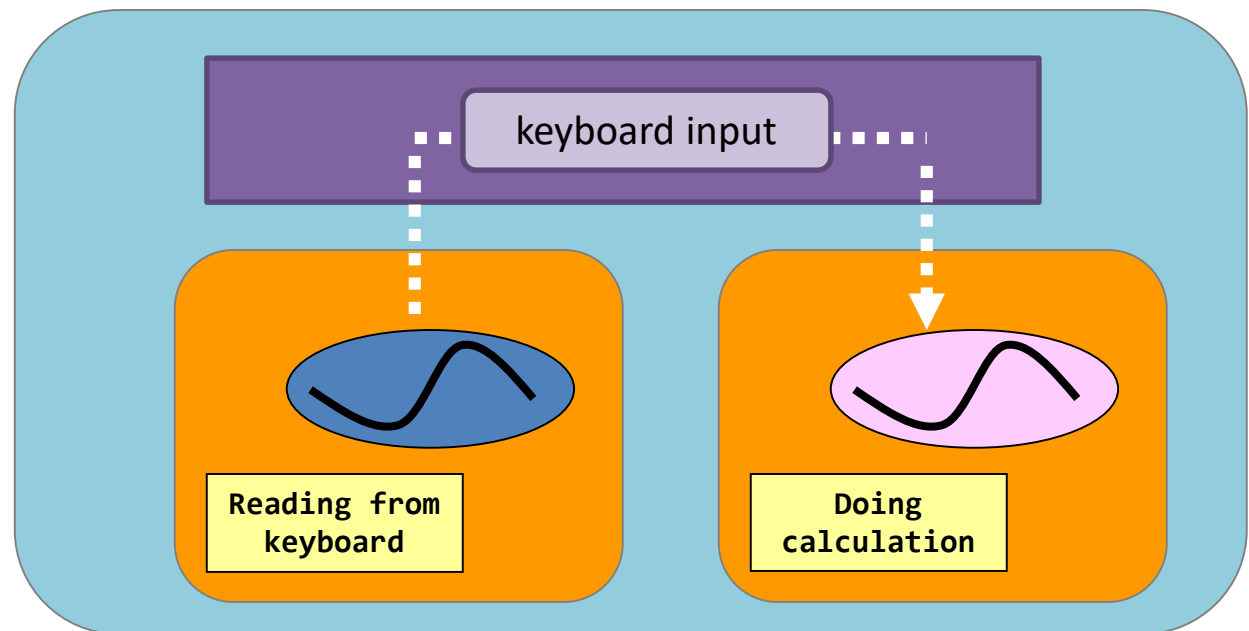
It'd be nice to assign **one thread for one blocking system/library call.**



Benefits of Multi-thread

- **Ease in data sharing**, can be done using:
 - global variables, and
 - dynamically allocated memory.
- Processes share resources via shared memory or message passing, which must be explicitly arranged by the programmer

Of course, this leads to the **mutual exclusion** & the **synchronization** problems (will be talked in later chapters)



Benefits of Multi-thread

- **Economy**

- Allocating memory and resources for process creation is costly, dozens of times slower than creating threads
- Context-switch between processes is also costly, several times of slower

- **Scalability**

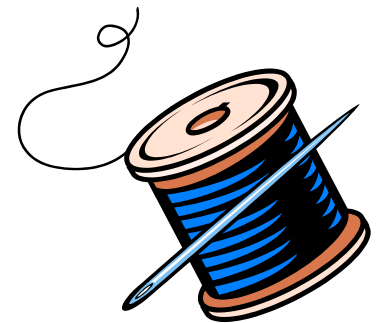
- Threads may be running in parallel on different cores

Programming Challenges

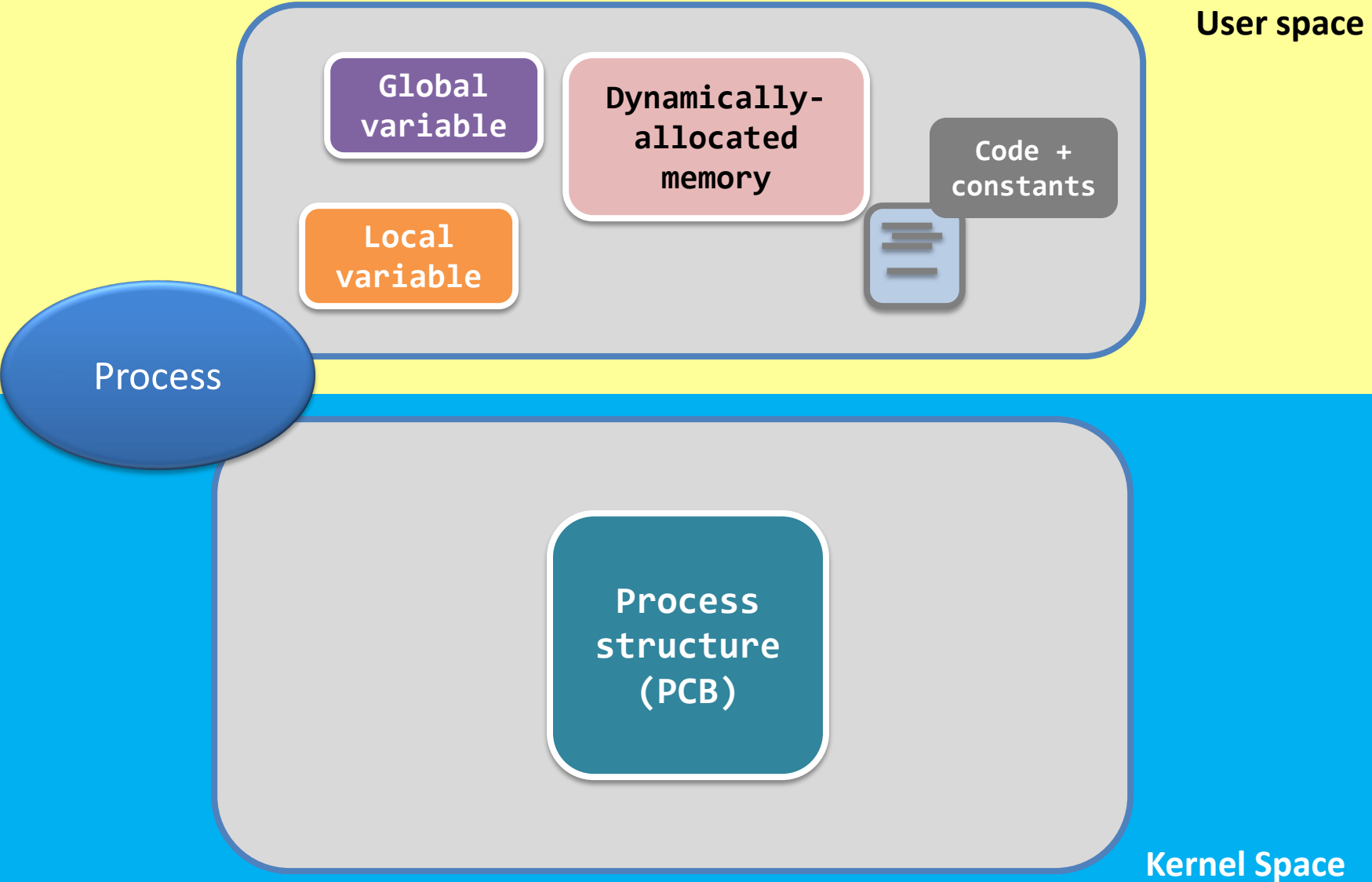
- **Identifying tasks**
 - Divide separate and concurrent tasks
- **Balance**
 - Tasks should perform equal work of equal value
- **Data splitting**
 - Data must be divided to run on separate cores
- **Data dependency**
 - Synchronization is needed
- **Testing and debugging**

Multi-threading

- Motivation
- Thread Concept
- Thread Models



Recall Process Structure



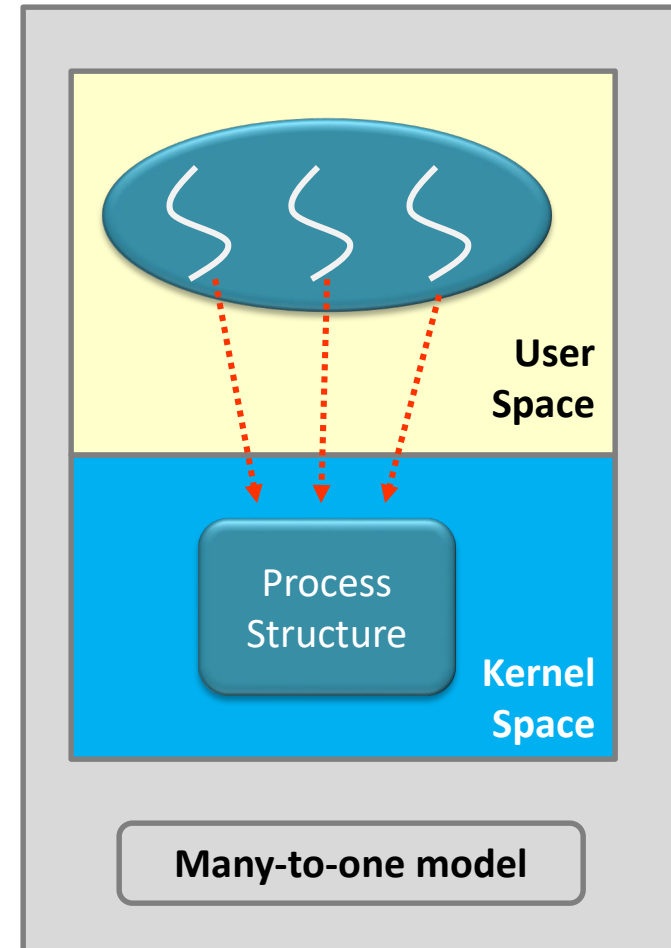
Similarly...

- Thread should also include
 - Data/resources in user-space memory
 - Structure in kernel
- How to provide thread support?
 - User thread
 - Implement in user space
 - Kernel thread
 - Supported and managed by kernel
- Thread models (relationship between user/kernel thread)
 - Many-to-one
 - One-to-one
 - Many-to-many

Thread models

- **Many-to-One Model**

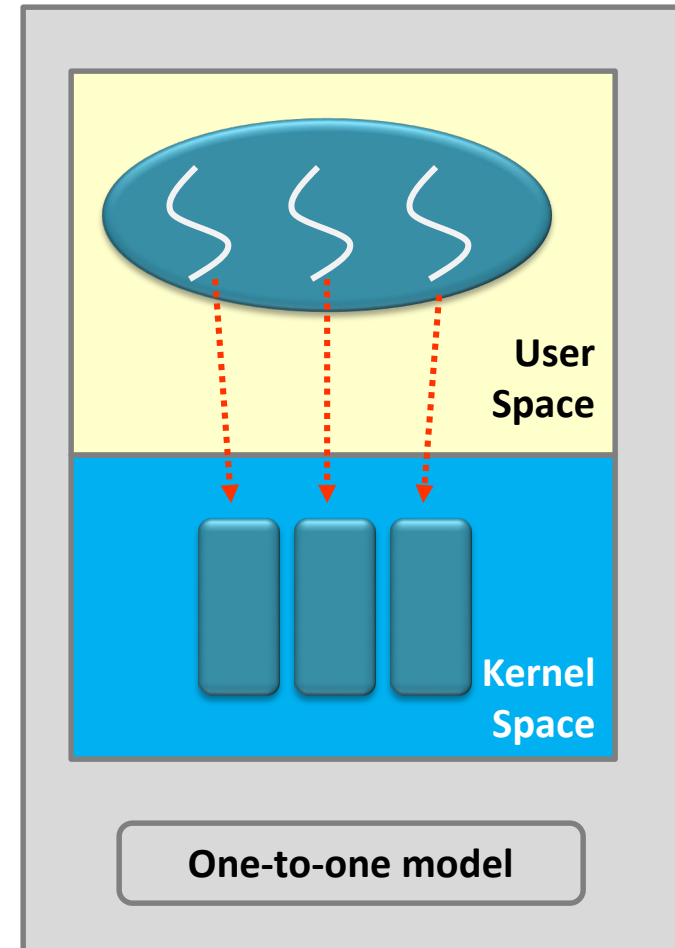
- All the threads are mapped to one process structure in the kernel.
- Merit
 - Easy for the kernel to implement.
- Drawback
 - When a blocking system call is called, all the threads will be blocked
- **Example.** Old UNIX & green thread in some programming languages.



Thread models

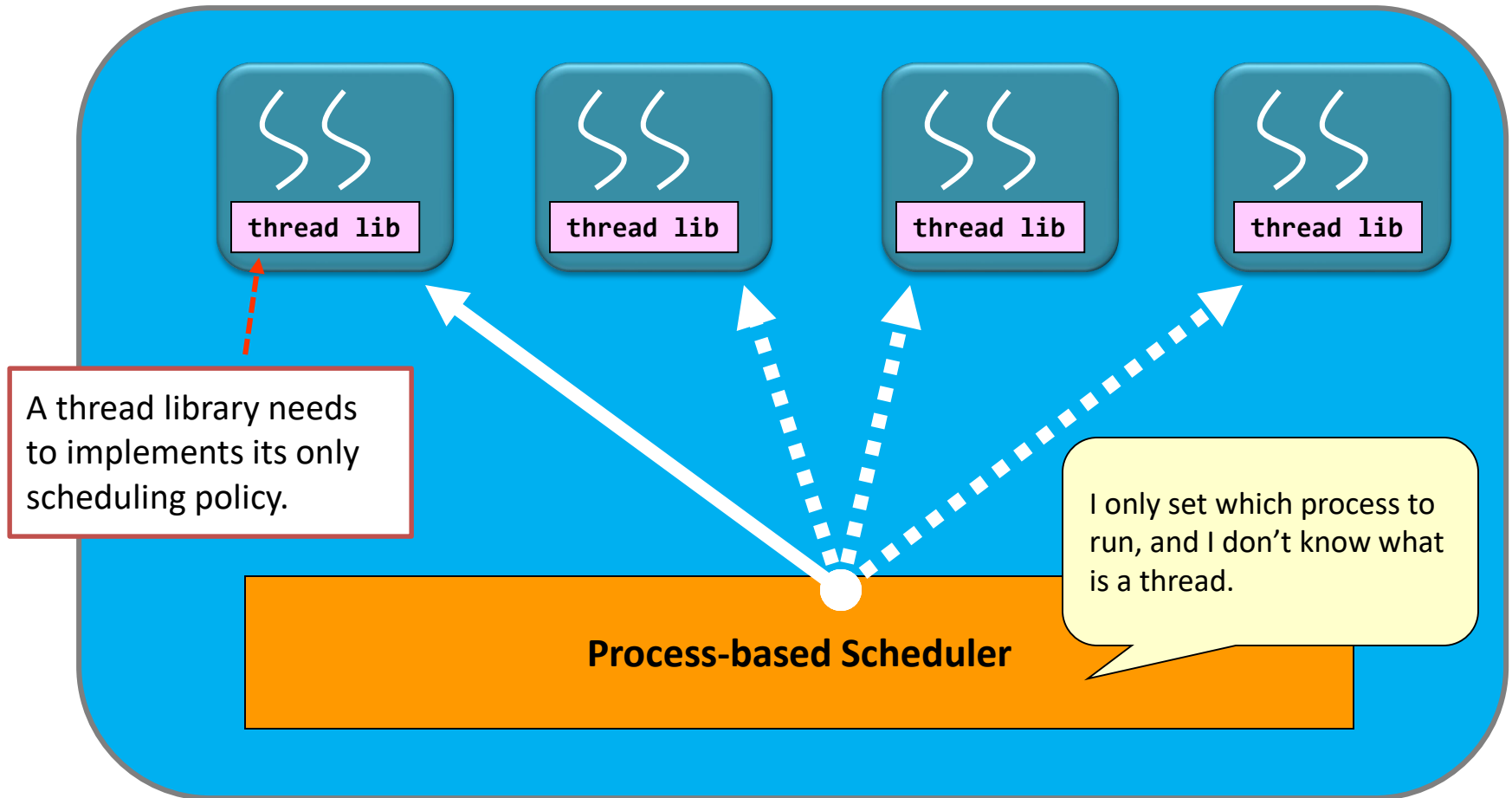
- **One-to-One Model**

- Each thread is mapped to a process or a thread structure
- Merit:
 - Calling blocking system calls only block those calling threads
 - **A high degree of concurrency**
- Drawback:
 - Cannot create too many threads as it is restricted by the size of the kernel memory
- **Example.** Linux and Windows follow this thread model



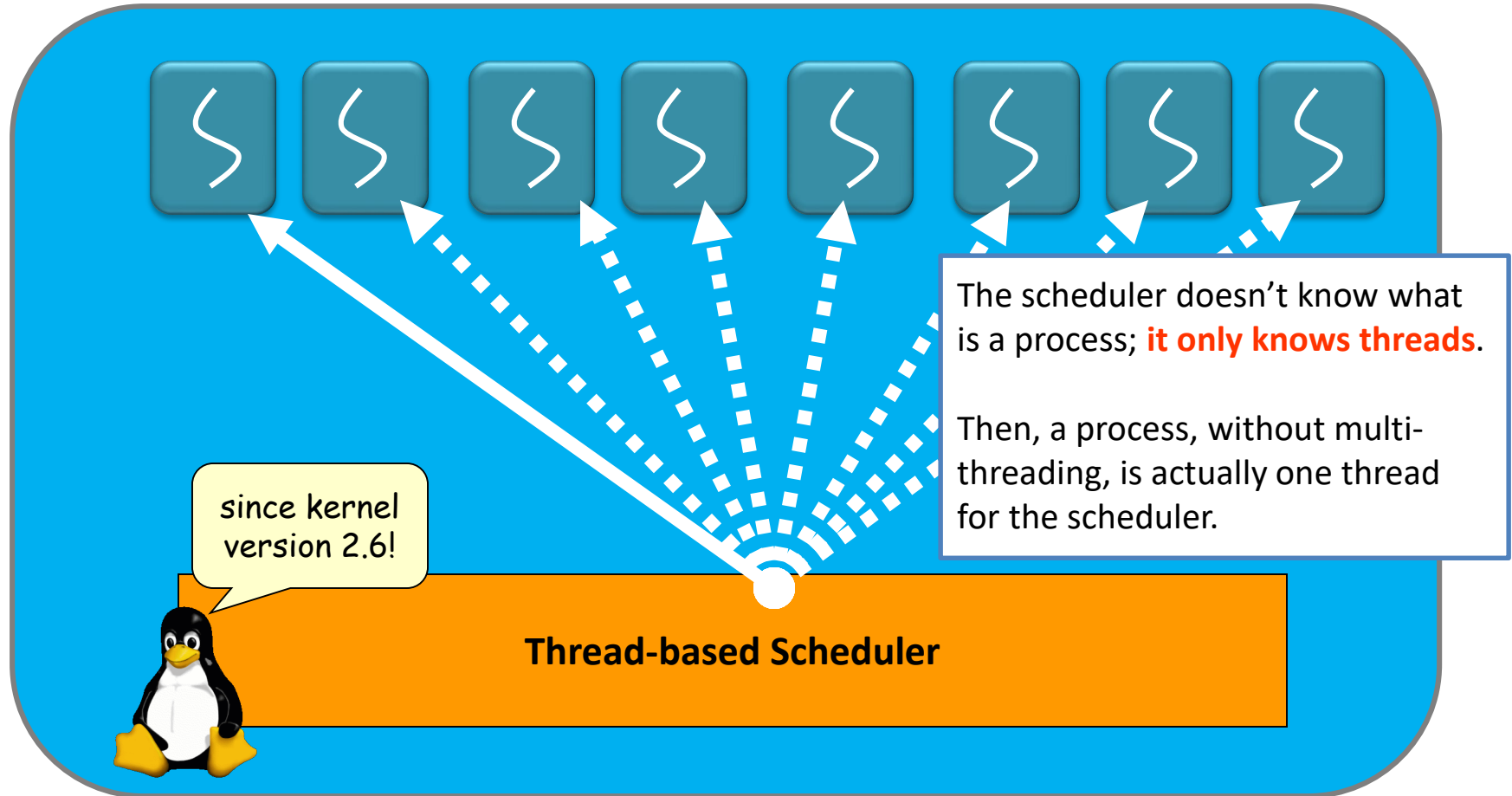
Scheduling – why & who cares?

- If a scheduler only interests in **processes**...



Scheduling – why & who cares?

- If a scheduler only interests in **threads**...



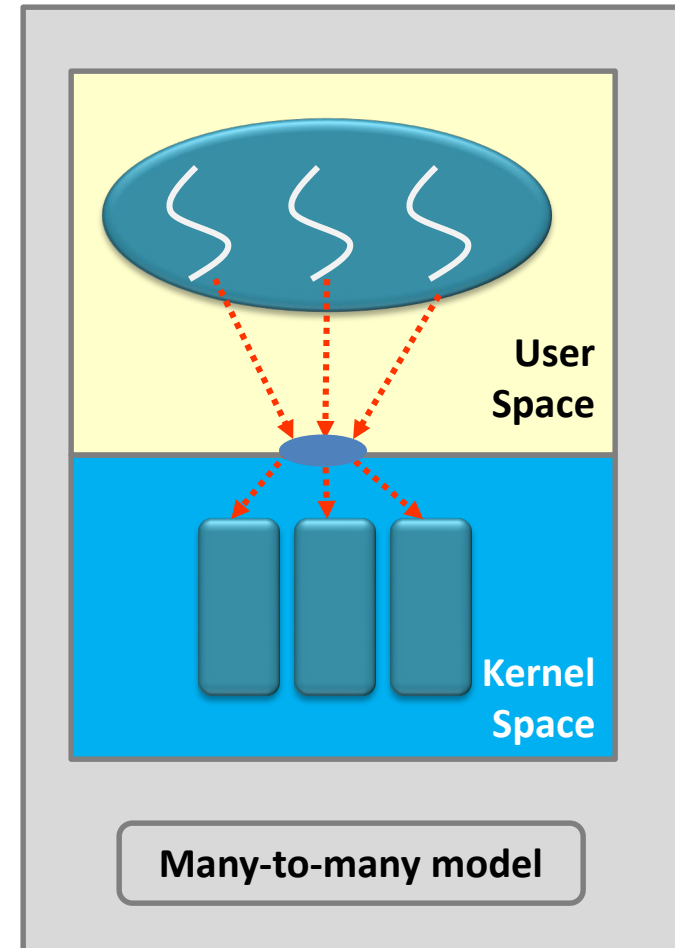
Thread models

- **Many-to-many Model**

- Multiple threads are mapped to multiple structures (group mapping)

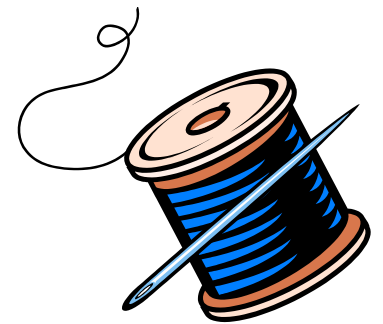
- Merit:

- Create as many threads as necessary
- Also have a high degree of concurrency



Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming



Thread Libraries

- A thread library provides the programmer with an **API** for creating and managing threads
 - Two ways of implementation: User-level or kernel-level
- Three main thread libraries
 - POSIX Pthreads (user-level or kernel-level)
 - Windows (kernel-level)
 - Java (implemented using Windows API or Pthreads)

Creating Multiple Threads

- Asynchronous threading
 - Parent resumes execution after creating a child
 - Parent and child execute **concurrently**
 - Each thread runs independently
 - **Little data sharing**
- Synchronous threading
 - Fork-join strategy: Parent waits for children to terminate
 - **Significant data sharing**

The Pthreads Library

- **Pthreads**: POSIX standard defining an API for thread creation and synchronization.
 - Specification, not implementation
- How to use Pthreads?

	Process	Thread
Creation	<code>fork()</code>	<code>pthread_create()</code>
I.D. Type	PID, an integer	“pthread_t”, a structure
Who am I?	<code>getpid()</code>	<code>pthread_self()</code>
Termination	<code>exit()</code>	<code>pthread_exit()</code>
Wait for child termination	<code>wait()</code> or <code>waitpid()</code>	<code>pthread_join()</code>
Kill?	<code>kill()</code>	<code>pthread_kill()</code>

ISSUE 1: Thread Creation

Thread creation – pthread_create()

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```


Thread creation – pthread_create()

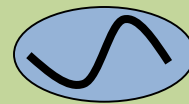
Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

At the beginning,
there is only one
thread running: **the
main thread.**



Main Thread

Thread creation – pthread_create()

Thread Function

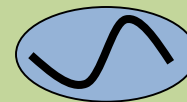
```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

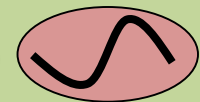
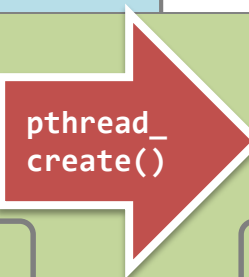
```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

The hello thread is created!

It is running *“together”* with the main thread.



Main Thread



Hello Thread

Thread creation – pthread_create()

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

The `pthread_create()` function allows one argument to be passed to the thread function.

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

This sets the thread function of the to-be-created thread as: `hello()`.

Remember: A thread starts with **one specific function (thread function)**

Thread creation – `pthread_create()`

Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

Remember `wait()`
and `waitpid()`?

`pthread_join()`
performs similarly.



Blocked

Main Thread



Hello Thread

Thread creation – pthread_create()

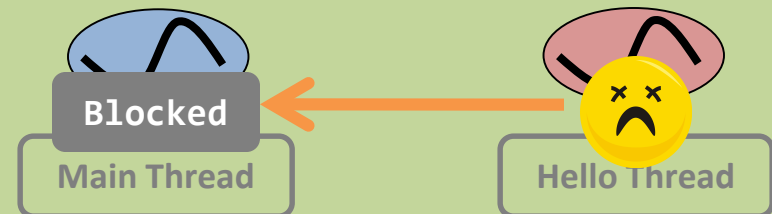
Thread Function

```
1 void * hello( void *input ) {  
2     printf(“%s\n”, (char *) input);  
3     pthread_exit(NULL);  
4 }
```

Main Function

```
5 int main(void) {  
6     pthread_t tid;  
7     pthread_create(&tid, NULL, hello, “hello world”);  
8     pthread_join(tid, NULL);  
9     return 0;  
10 }
```

Termination of the target thread causes `pthread_join()` to return.



ISSUE 2: Passing parameters

Thread creation – passing parameter

Thread Function

```
1 void * do_your_job( void *input ) {
2     printf("child = %d\n", *( (int *) input) );
3     *((int *) input) = 20;
4     printf("child = %d\n", *( (int *) input) );
5     pthread_exit(NULL);
6 }
```

Main Function

```
7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf("main = %d\n", input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf("main = %d\n", input);
14    return 0;
15 }
```

Guess: What is the output?

```
$ ./pthread_evil_1
main = 10
child = 10
child = 20
main = 20
$
```



Each thread has a separated stack.

Why do we have such results?

Thread creation – passing parameter

Well, we all know that the local variable “input” is in the stack for the main thread.

```
1 void * do_your_job( void *input ) {
2     printf(“child = %d\n”, *( (int *) input) );
3     *((int *) input) = 20;
4     printf(“child = %d\n”, *( (int *) input) );
5     pthread_exit(NULL);
6 }

7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf(“main = %d\n”, input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf(“main = %d\n”, input);
13    return 0;
14 }
```

Local
(main thread)

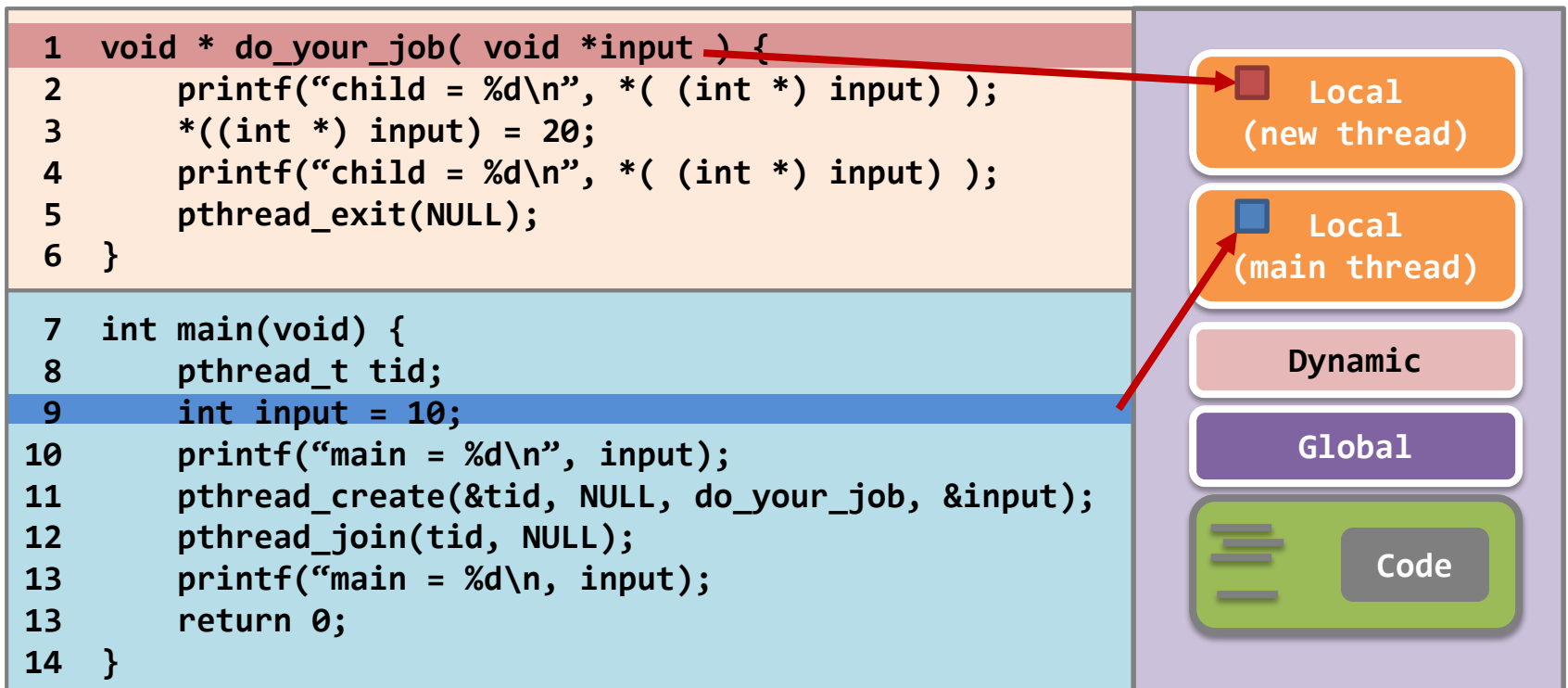
Dynamic

Global

Code

Thread creation – passing parameter

Yet...the stack for the new thread is not on another process, but is on the same piece of user-space memory as the main thread.

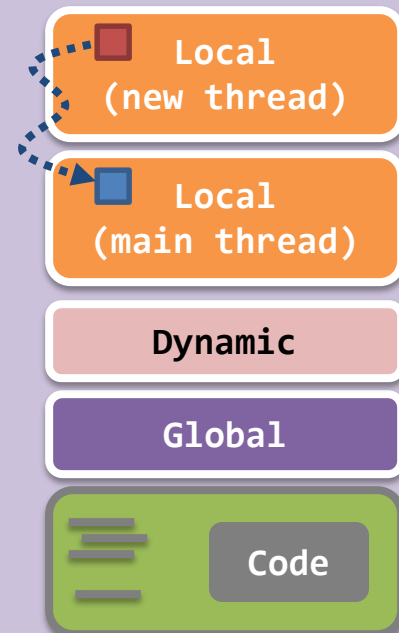


Thread creation – passing parameter

The `pthread_create()` function only passes an **address** to the new thread. Worse, the address is pointing to a variable in the stack of the main thread!

```
1 void * do_your_job( void *input ) {
2     printf("child = %d\n", *( (int *) input) );
3     *((int *) input) = 20;
4     printf("child = %d\n", *( (int *) input) );
5     pthread_exit(NULL);
6 }

7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf("main = %d\n", input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf("main = %d\n", input);
13    return 0;
14 }
```

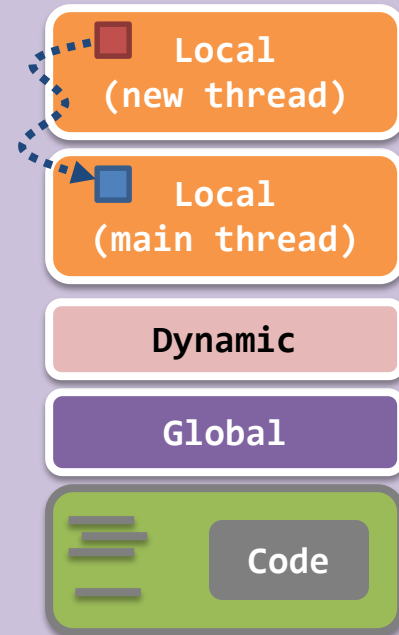


Thread creation – passing parameter

Therefore, the new thread can change the value in the main thread, and vice versa.

```
1 void * do_your_job( void *input ) {
2     printf("child = %d\n", *( (int *) input) );
3     *((int *) input) = 20;
4     printf("child = %d\n", *( (int *) input) );
5     pthread_exit(NULL);
6 }

7 int main(void) {
8     pthread_t tid;
9     int input = 10;
10    printf("main = %d\n", input);
11    pthread_create(&tid, NULL, do_your_job, &input);
12    pthread_join(tid, NULL);
13    printf("main = %d\n", input);
14    return 0;
15 }
```



ISSUE 3: Multiple Threads

Thread creation – multiple threads


Thread Function

```
1 void * do_your_job(void *input) {
2     int id = *((int *) input);
3     printf("My ID number = %d\n", id);
4     pthread_exit(NULL);
5 }
```

Main Function

```
6 int main(void) {
7     int i;
8     pthread_t tid[5];
9
10    for(i = 0; i < 5; i++)
11        pthread_create(&tid[i], NULL, do_your_job, &i);
12    for(i = 0; i < 5; i++)
13        pthread_join(tid[i], NULL);
14    return 0;
15 }
```

Waiting on several threads: enclose pthread_join() within a for loop



ISSUE 4: Return Value

Thread termination – passing return value

Thread Function

```
1 void * do_your_job(void *input) {  
2   int *output = (int *) malloc(sizeof(int));  
3   srand(time(NULL));  
4   *output = ((rand() % 10) + 1) * (*((int *) input));  
5   pthread_exit(output);  
6 }
```

`void pthread_exit(void *return_value);`

Together with termination, a pointer to a **global variable or a piece of dynamically allocated memory** is returned to the main thread.

Main Function

```
7 int main(void) {  
8   pthread_t tid;  
9   int input = 10, *output;  
10  pthread_create(&tid, NULL, do_your_job, &input);  
11  pthread_join(tid, (void **) &output);  
12  return 0;  
13 }
```

Using pass-by-reference, a pointer to the result is received in the main thread.

- For Windows threads and Java threads, you can refer to the textbook if you are interested in.

Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming
- Implicit Threading



Implicit Threading

- Applications are containing hundreds or even thousands of threads
 - Program correctness is more difficult with explicit threads
- How to address the programming difficulties?
 - Transfer the creation and management of threading from programmers to compilers and run-time libraries
 - Implicit threading
- We will introduce two methods
 - Thread Pools
 - OpenMP

Thread Pools

- Problems with multithreaded servers
 - Time required to create threads, which will be discarded once completed their work
 - Unlimited threads could exhaust the system resources
- How to solve?
 - Thread pool
 - Idea
 - Create a number of threads in a pool where they wait for work
 - Procedure
 - Awakens a thread if necessary
 - Returns to the pool after completion
 - Waits until one becomes free if the pool contains no available thread

Thread Pools

- Advantages
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

OpenMP

- Provides support for parallel programming in shared-memory environments
- Set of compiler directives and an API for C, C++, FORTRAN
- Identifies **parallel regions** – blocks of code that can run in parallel

Parallel for loop

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

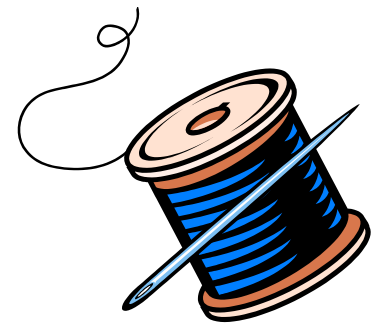
```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */
    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }
    /* sequential code */
    return 0;
}
```

When OpenMP encounters the directive, it creates as many threads as there are processing cores

Multi-threading

- Motivation
- Thread Concept
- Thread Models
- Basic Programming
- Implicit Threading
- Threading Issues



Semantics of `fork()` and `exec()`

- Two key system calls for processes: **fork**, **exec**
- **fork()**: Some UNIX systems have two versions
 - The new process duplicates all threads, or
 - Duplicates only the thread that invoked **fork()**
- **exec()**: usually works as normal
 - Replace the running process - including **all threads**

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred
 - Synchronous signal and asynchronous signal
 - Default handler or user-defined handler
- Where should a signal be delivered in multi-threaded program?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- Deliver a signal to a specified thread with Pthread
 - `pthread_kill(pthread_t tid, int signal)`

Thread Cancellation

- Terminating a thread before it has finished
 - Why needed?
 - Example: Close a browser when multiple threads are loading images
- Two general approaches
 - **Asynchronous cancellation** terminates the target thread immediately
 - Problem: Troublesome when canceling a thread which is updating data shared by other threads
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled (can be canceled safely)

Thread Cancellation (Cont.) - Pthreads

- Pthreads code example
 - **pthread_cancel()**
 - Indicates only a request

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

- Three cancellation modes

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- Default: deferred
 - Cancellation occurs only when it reaches a cancellation point, can be established by **pthread_testcancel()**

Thread-Local Storage

- Some applications, each thread may need its own copy of certain data
 - Transaction processing system: service each transaction (with a unique identifier) in a thread
 - How about local variables?
 - Visible only during a single function invocation
- Thread-local storage (TLS) allows each thread to have its own copy of data
 - TLS is **visible across function invocations**
 - Similar to **static** data
 - TLS data are unique to each thread

Summary of Threads

- Virtually all modern OSes support multi-threading
 - A thread is a basic unit of CPU utilization
 - Each comprises a thread ID, a program counter, a register set, and a stack
 - All threads within a process share code section, data section, other resources like open files and signals
- You should **take great care** when writing multi-threaded programs
- You also have to take care of (will be talked later):
 - Mutual exclusion and
 - Synchronization

End of Chapter 4