3150 - Operating Systems

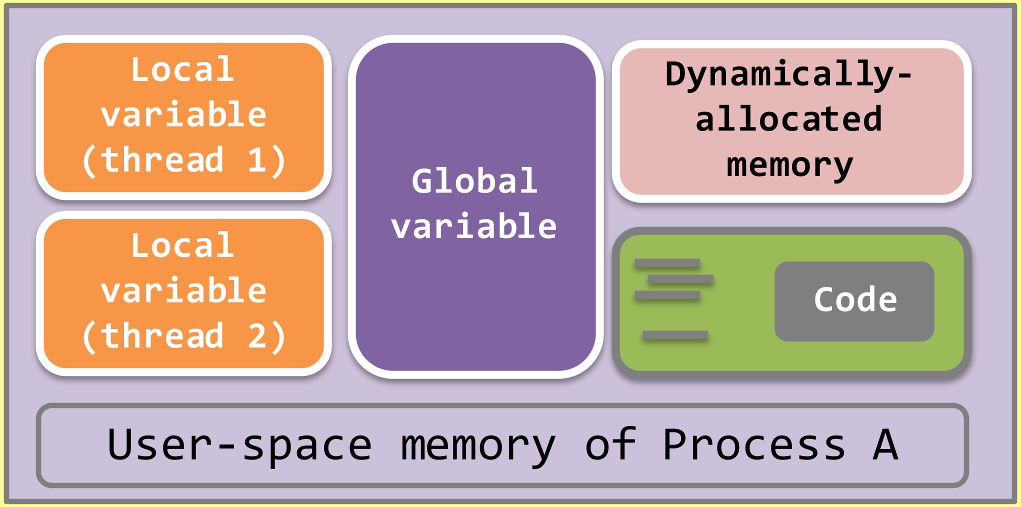
Dr. WONG Tsz Yeung

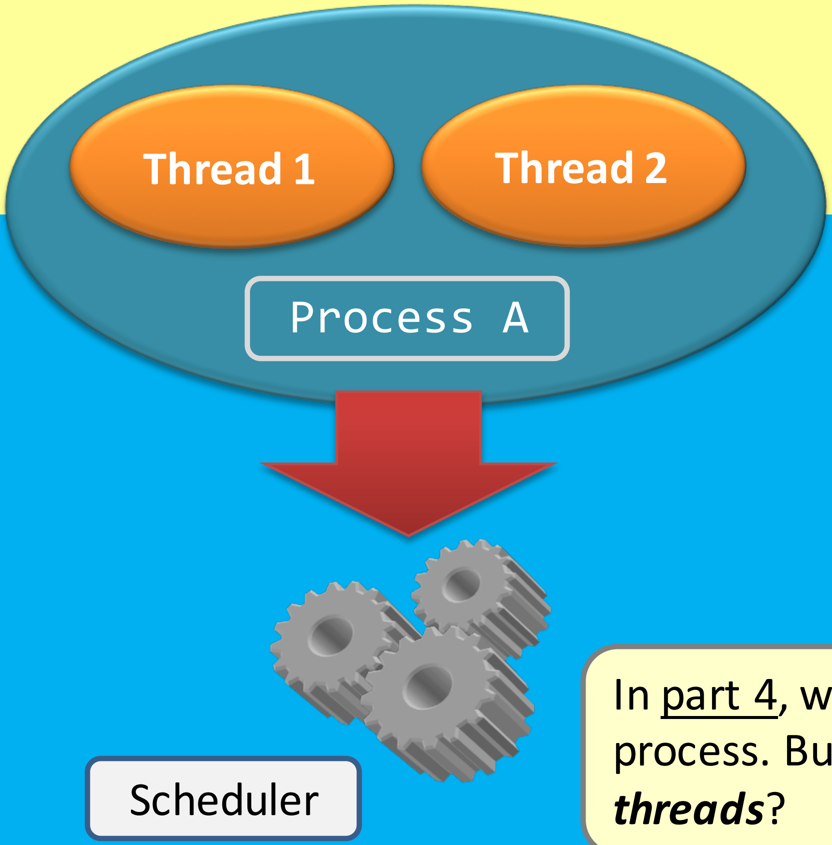
Chapter 2, part 6
Light-Weight Processes: Threads

-threads & processes: like spaghetti & meatballs mixed together…

they are delicious but complicated…

Outline





Unlike **`fork()`**, thread creation would not duplicate user-space memory. Worse, it is designed to be sharable!

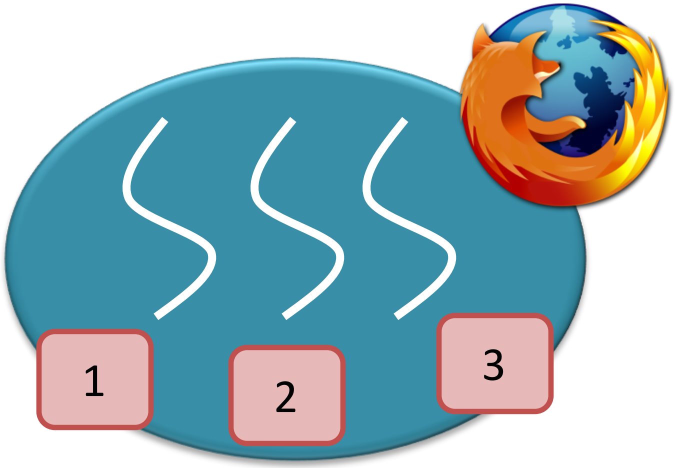
Will those horrible topics in <u>part 5</u> reappear?!

In <u>part 4</u>, we know how to schedule process. But, how about ***scheduling threads***?

Multi-threading
- Introduction.

Multi-thread – an example

- Threading is everywhere:
  - Software with GUI;
  - Network programs;



- Basically, we need threads for:
  - Executing parallel tasks for easy data sharing;
  - Distributing blocking calls to different threads.

Thread 1: Networking.

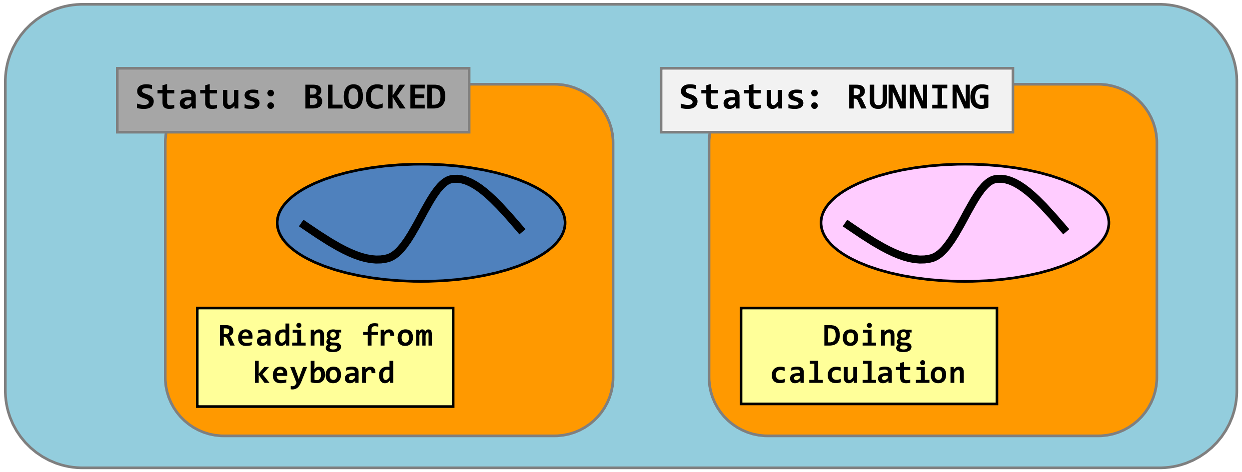
Thread 2: GUI control.

Thread 3: Flash player.

Multi-thread – merits

- Responsiveness and multi-tasking.
  – Multi-threading design allows an application to do parallel tasks simultaneously.
  – See from the following example?
    - Although a thread is blocked, the process can still depend on another thread to do other things!

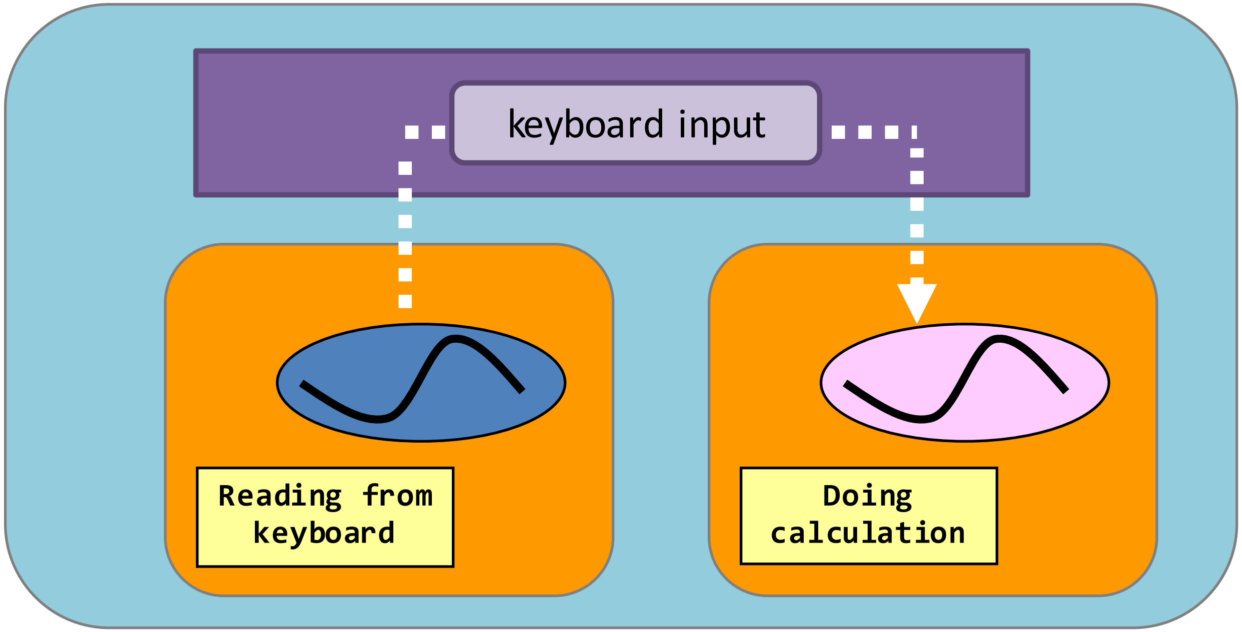
It'd be nice to assign **one thread for one blocking system/library call**.



Multi-thread – merits

- Ease in data sharing.
  - Data sharing can be done using:
    - global variables, and
    - dynamically allocated memory.

Of course, this leads to the **mutual exclusion** & the **synchronization** problems.




```
[examples@3150] cat pthread_game.c
```

Multi-thread – internals

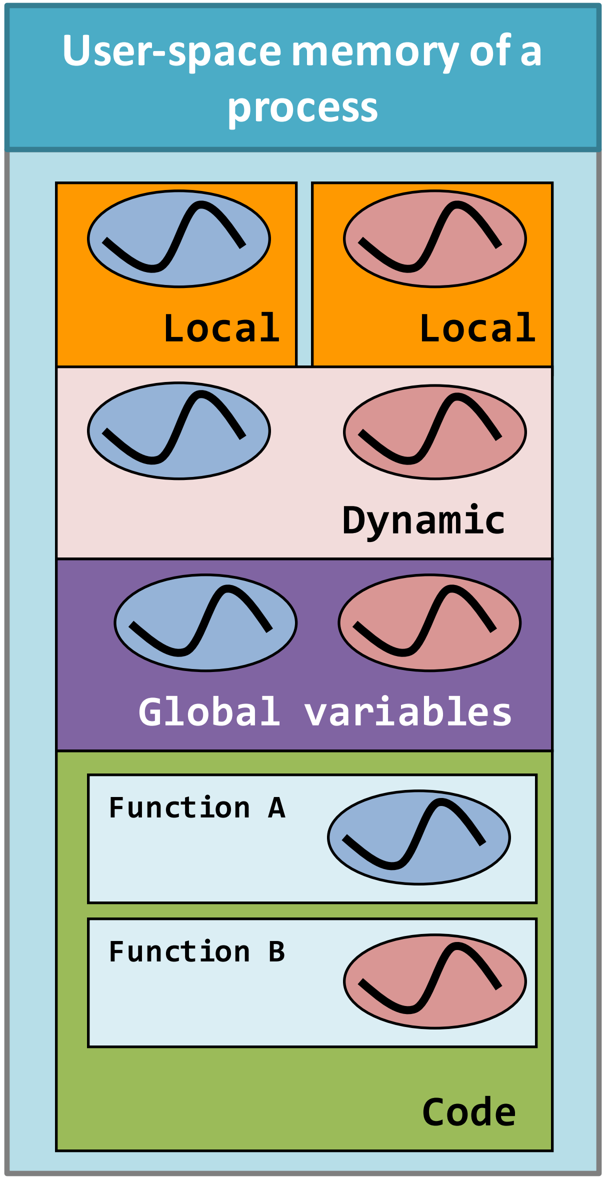
Code

- All threads shares the same code.

- A thread starts with **one specific function**. and we name it the **thread function**. E.g., Functions A & B in the diagram.

- Of course, the thread function can invoke other functions or system calls.

- But, a thread could <u>**never return to the caller of the thread function**</u>.



Multi-thread – internals

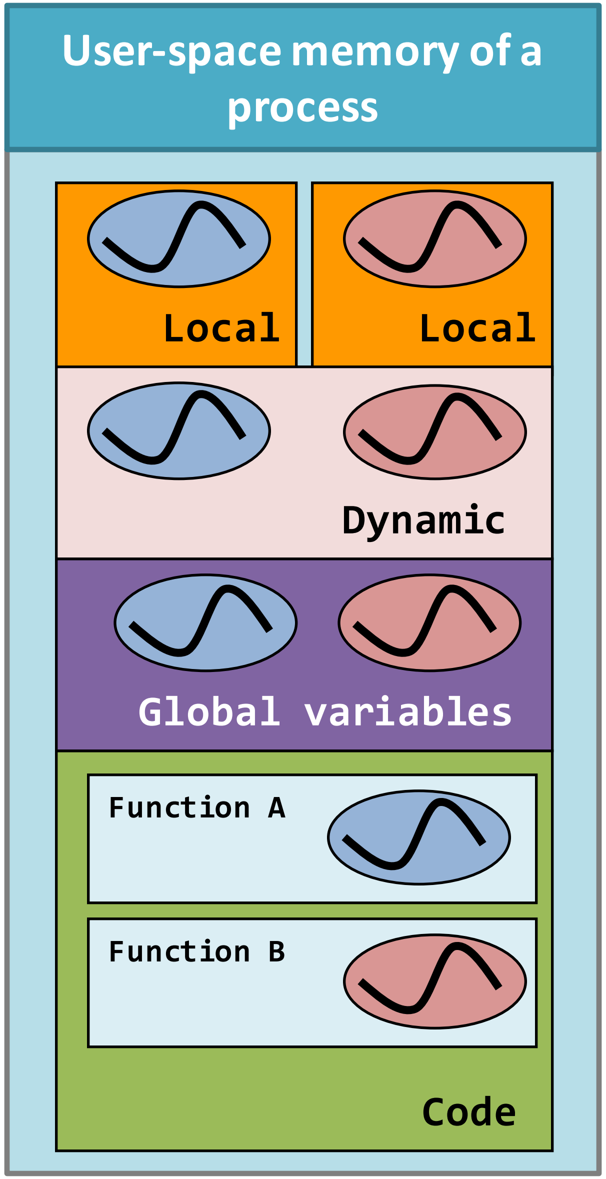
Dynamically allocated memory

Global variables

- All threads shares the same **global variable zone** and the same **dynamically allocated memory**.

- All threads can read from and write to both areas.

- So, will a multi-threaded process have the **race condition** problem built into the design?

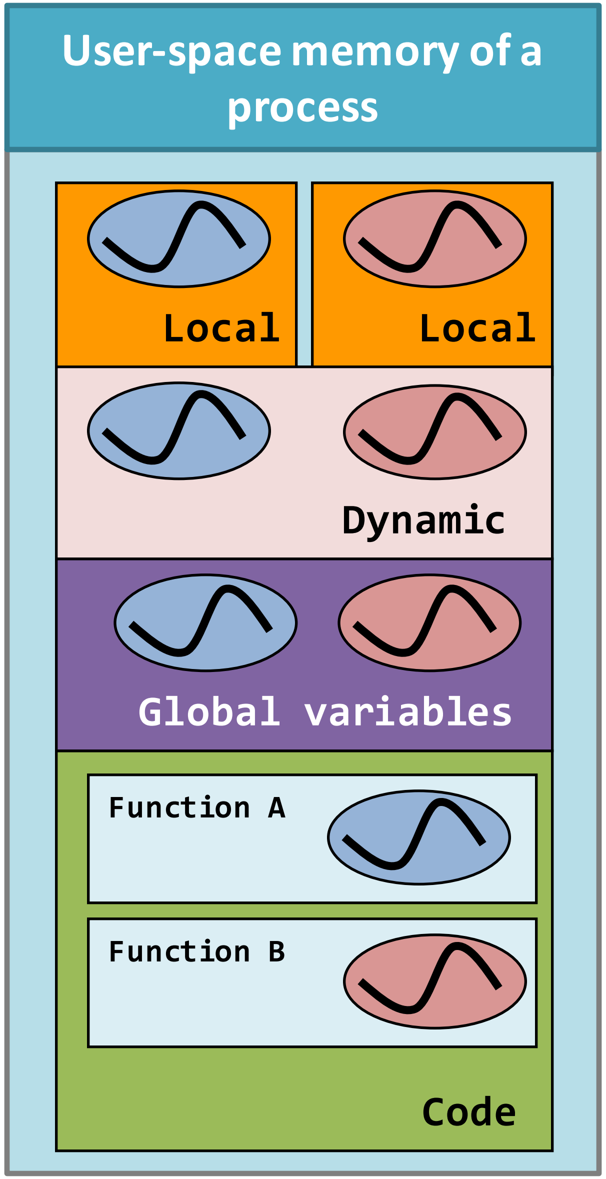


Multi-thread – internals

Local variables

- Each thread has <u>**its own memory range**</u> for the local variables.

- So, the stack is the private zone for each stack.



Multi-threading

- Introduction.
- Basic Programming.

The pthread library

- We use a library called **pthread**, POSIX thread.

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

Thread creation – **pthread_create()**

Thread Function

```
void * hello( void *input ) {
     printf("%s\n", (char *) input);
     pthread_exit(NULL);
}
```

Main Function

```
int main(void) {
     pthread_t tid;
     pthread_create(&tid, NULL, hello, "hello world");
     pthread_join(tid, NULL);
     return 0;
}
```

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

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

```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – **pthread_create()**

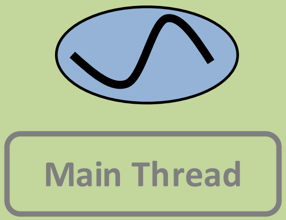
Thread Function

```
void * hello( void *input ) {
    printf("%s\n", (char *) input);
    pthread_exit(NULL);
}
```

Main Function

```
int main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, hello, "hello world");
    pthread_join(tid, NULL);
    return 0;
}
```

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




```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – **pthread_create()**

Thread Function

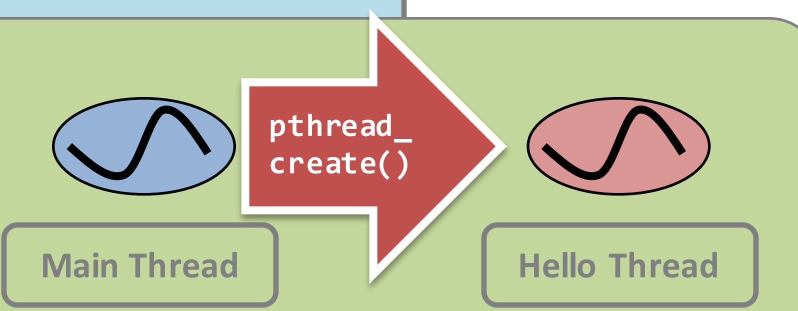
```c
void * hello( void *input ) {
     printf(“%s\n”, (char *) input);
     pthread_exit(NULL);
}
```

Main Function

```c
int main(void) {
     pthread_t tid;
     pthread_create(&tid, NULL, hello, “hello world”);
     pthread_join(tid, NULL);
     return 0;
}
```

The hello thread is created!

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




```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – `pthread_create()`

Thread Function

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

Main Function

```
int main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, hello, “hello world”);
    pthread_join(tid, NULL);
    return 0;
}
```

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

`pthread_join()` performs similarly.


```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – `pthread_create()`

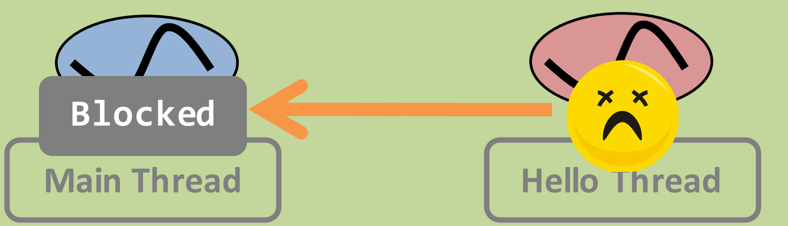
Thread Function

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

Main Function

```
int main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, hello, “hello world”);
    pthread_join(tid, NULL);
    return 0;
}
```

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




```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – passing parameter?

Thread Function

```c
void * do_your_job( void *input ) {
    printf(“child = %d\n”, *( (int *) input) );
    *((int *) input) = 20;
    printf(“child = %d\n”, *( (int *) input) );
    pthread_exit(NULL);
}
```

Main Function

```c
int main(void) {
    pthread_t tid;
    int input = 10;
    printf(“main = %d\n”, input);
    pthread_create(&tid, NULL, do_your_job, &input);
    pthread_join(tid, NULL);
    printf(“main = %d\n”, input);
    return 0;
}
```

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

What?! Each thread should have a separated stack, right?

Why do we have such results!!!

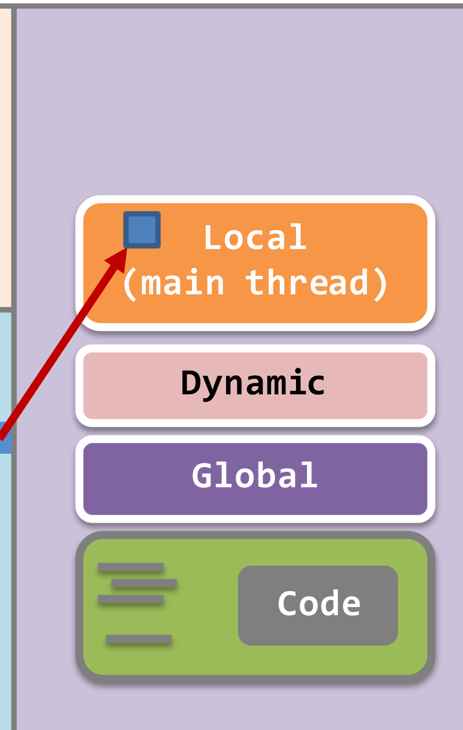
```
[examples@3150] cat pthread_evil_1.c
```

Thread creation – passing parameter?

Well, we all know that the local variable “**`input`**” is in **<u>the stack for the main thread</u>**.

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

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



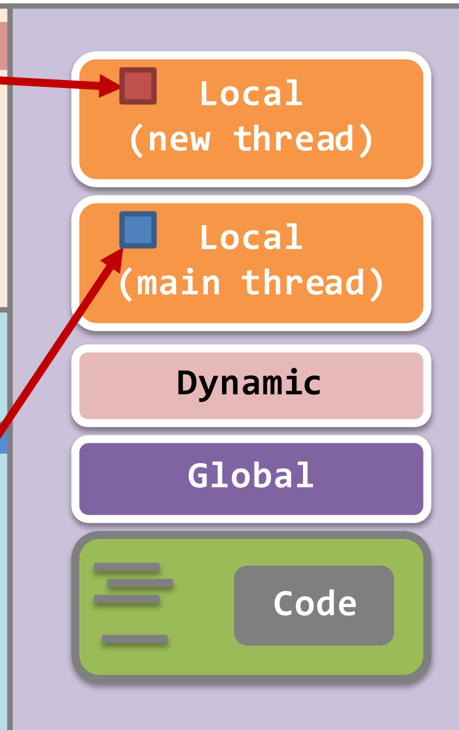

```
[examples@3150] cat pthread_evil_1.c
```

Thread creation – passing parameter?

Yet…the stack for the new thread is not on the another process, but is **<u>on the same piece of user-space memory as the main thread</u>**.

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

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



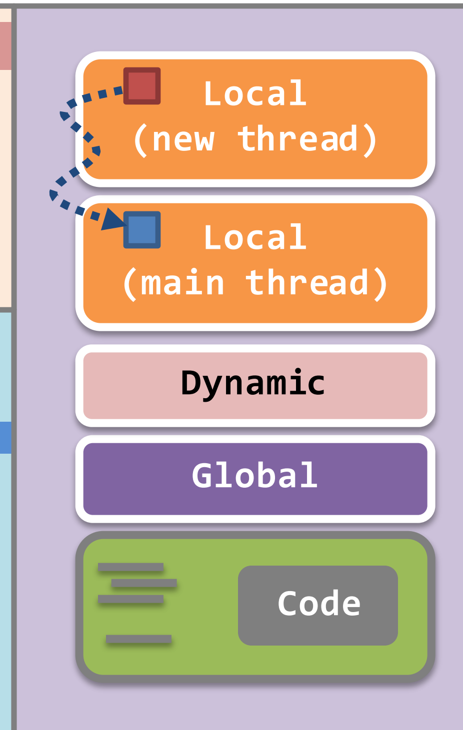

```
[examples@3150] cat pthread_evil_1.c
```

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**!

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

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



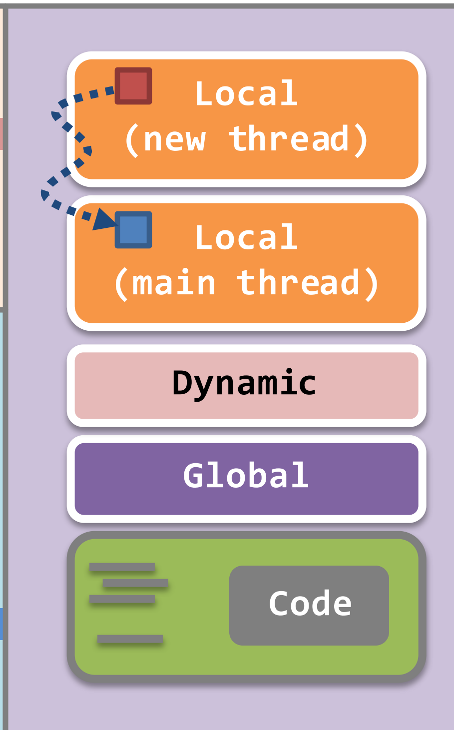

```
[examples@3150] cat pthread_evil_1.c
```

Thread creation – passing parameter?

Therefore, the new thread can change the value in the main thread, and **vice versa**.
Is it brain-damaging?

```
void * do_your_job( void *input ) {
    printf("child = %d\n", *( (int *) input) );
    *((int *) input) = 20;
    printf("child = %d\n", *( (int *) input) );
    pthread_exit(NULL);
}

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




```
[examples@3150] cat pthread_evil_1.c
```

Thread creation – passing parameter?

Thread Function

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

Doing backup!

Just want to avoid accessing the stack of the main thread.

Main Function

```c
int main(void) {
    int i;
    pthread_t tid[5];

    for(i = 0; i < 5; i++)
        pthread_create(&tid[i], NULL, do_your_job, &i);
    for(i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

Challenge!

Is there any problem?

If yes, how to correct it?

```
[examples@3150] cat pthread_evil_2.c pthread_evil_2sol.c
```

Thread termination – passing return value?

Thread Function

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

Important! Similar to the normal function call, you cannot return a pointer to a local variable.

```
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

```
int main(void) {
    pthread_t tid;
    int input = 10, *output;
    pthread_create(&tid, NULL, do_your_job, &input);
    pthread_join(tid, (void **) &output );
    printf("output = %d\n", *output);
    return 0;
}
```

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

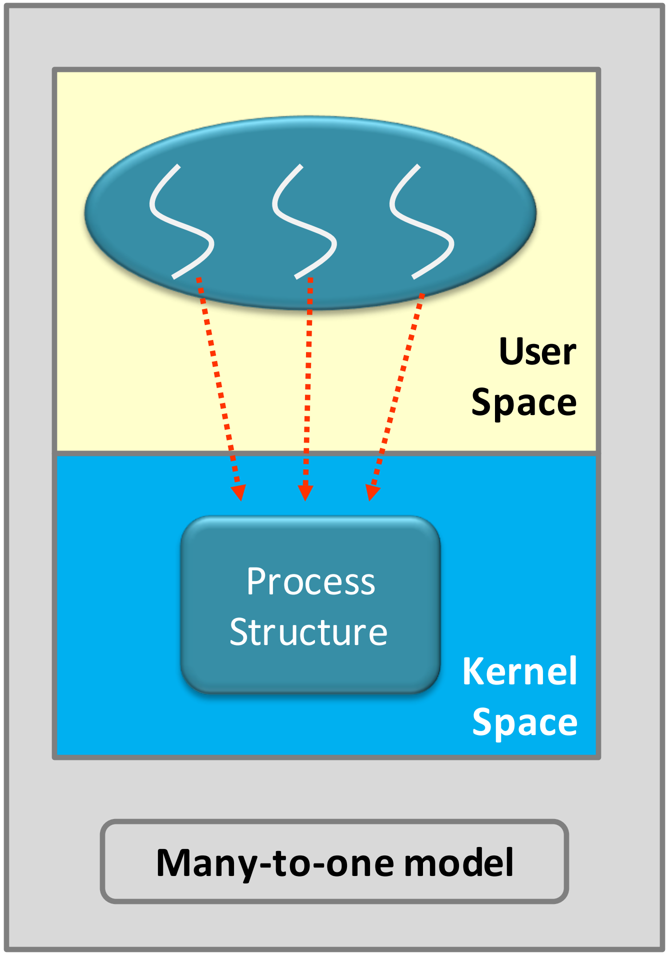
```
[examples@3150] cat pthread_exit.c
```

Multi-threading

- Introduction.
- Basic Programming.
- Thread scheduling?

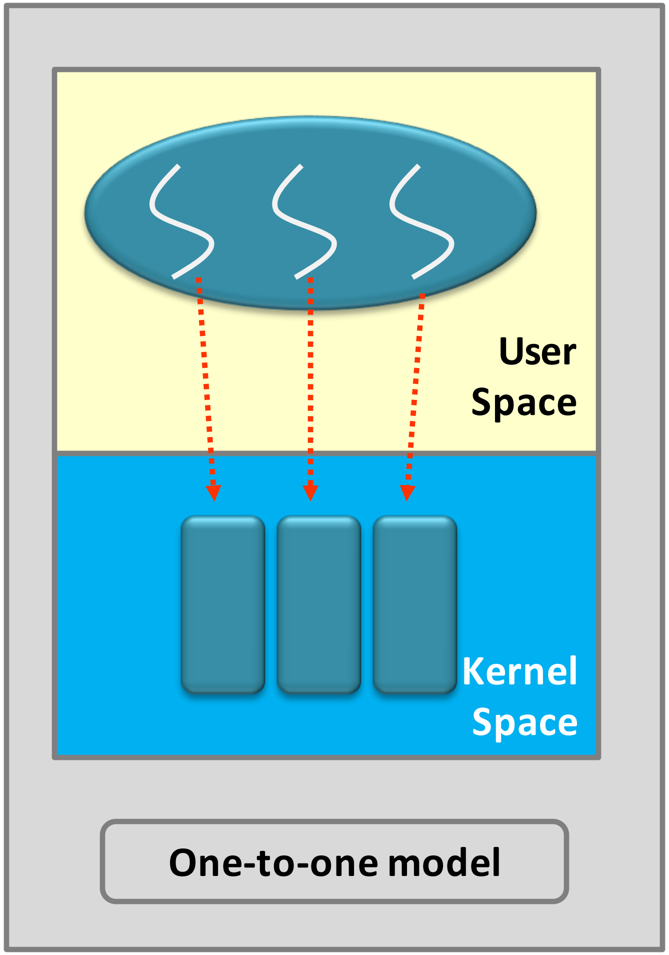
Thread models – 1 of 2

- **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.



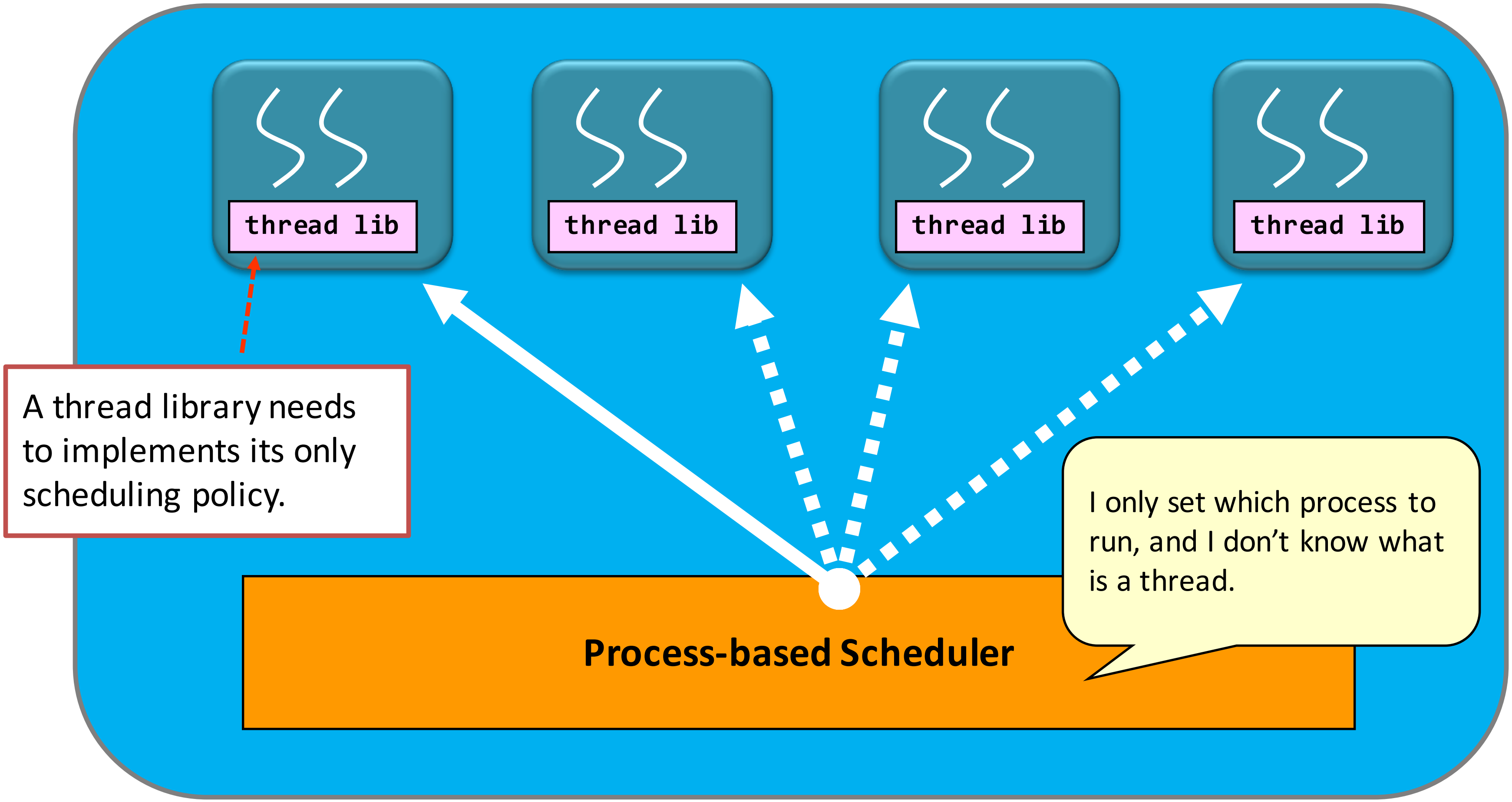
Thread models – 2 of 2

- **<u>One-to-One Model</u>**
  - 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.
  - **<u>Example.</u>** Linux and Windows follow this thread model.



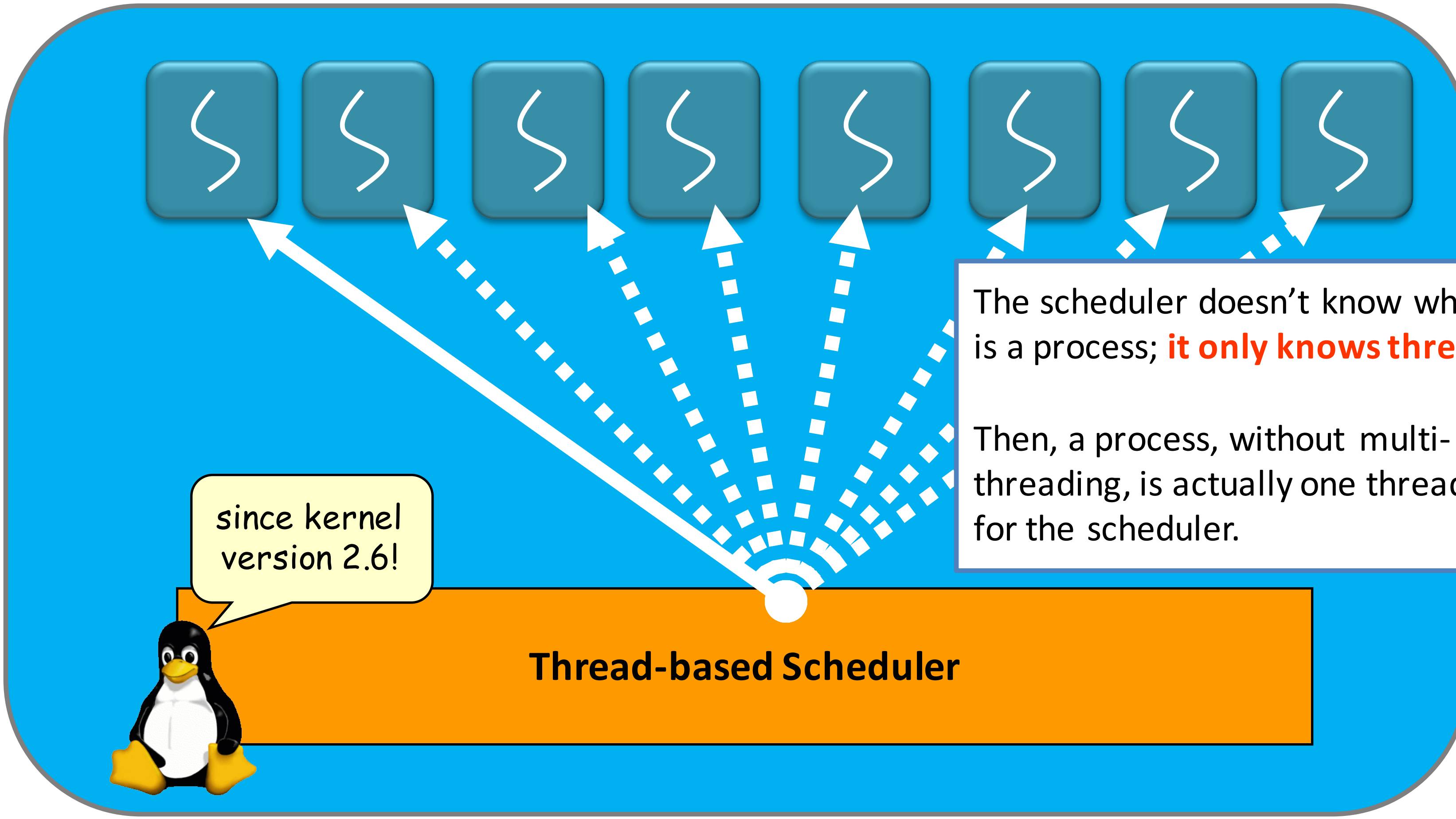
Scheduling – why & who cares?

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



Scheduling – why & who cares?

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



The scheduler doesn't know what is a process; **it only knows threads**.

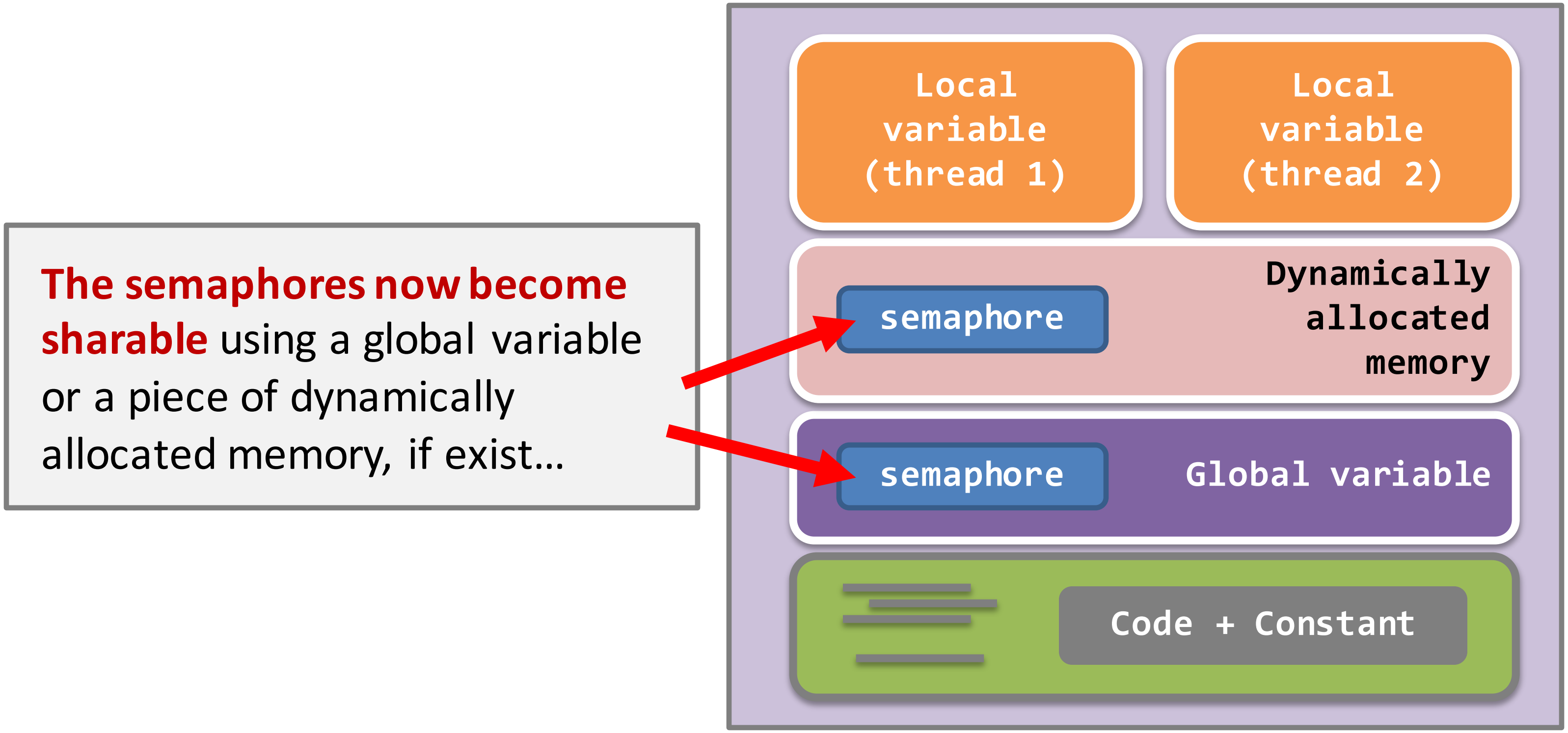
Then, a process, without multi-threading, is actually one thread for the scheduler.

Multi-threading

- Introduction.
- Basic Programming.
- Thread scheduling?
- Mutual exclusion & synchronization!

Mutual exclusion…using semaphore?

- We meet our old friend…
  - Dynamically allocated memory & global variables are shared objects.
  - But, there is **no semaphore** in the pthread library!



Mutual exclusion...using mutex!

Data type
`pthread_mutex_t`

This is a binary semaphore.

Initialization #1	`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;` This is the initialization at the declaration statement only.
Initialization #2	`pthread_mutex_init(&mutex, NULL);` This is the initialization for any locations in a program.
Locking	`pthread_mutex_lock(&mutex);` - If the mutex is not locked, the call locks the mutex. - If the mutex is locked, the call blocks the calling thread.
Unlocking	`pthread_mutex_unlock(&mutex);` - If the mutex is locked, the call unlocks the mutex. If there are threads blocked because of this mutex, those threads will resume. - If the mutex is unlocked, the call does nothing.

Mutual exclusion…example

Data type
pthread_mutex_t

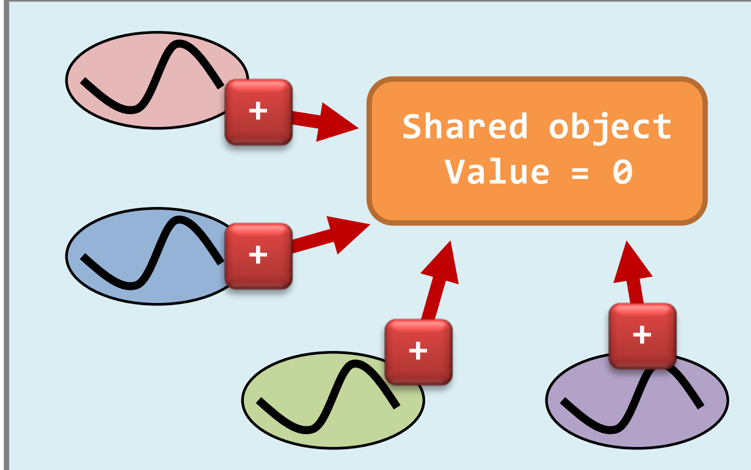
This is a binary semaphore.

Thread Function

```
while(TRUE) {
    pthread_mutex_lock(&mutex);
    if(shared < MAXIMUM)
        shared++;
    else {
        pthread_mutex_unlock(&mutex);
        break;
    }
    pthread_mutex_unlock(&mutex);
    sleep(rand() % 2);
}
```

Shared object.

Purpose of the program?




```
[examples@3150] cat pthread_mutex.c
```

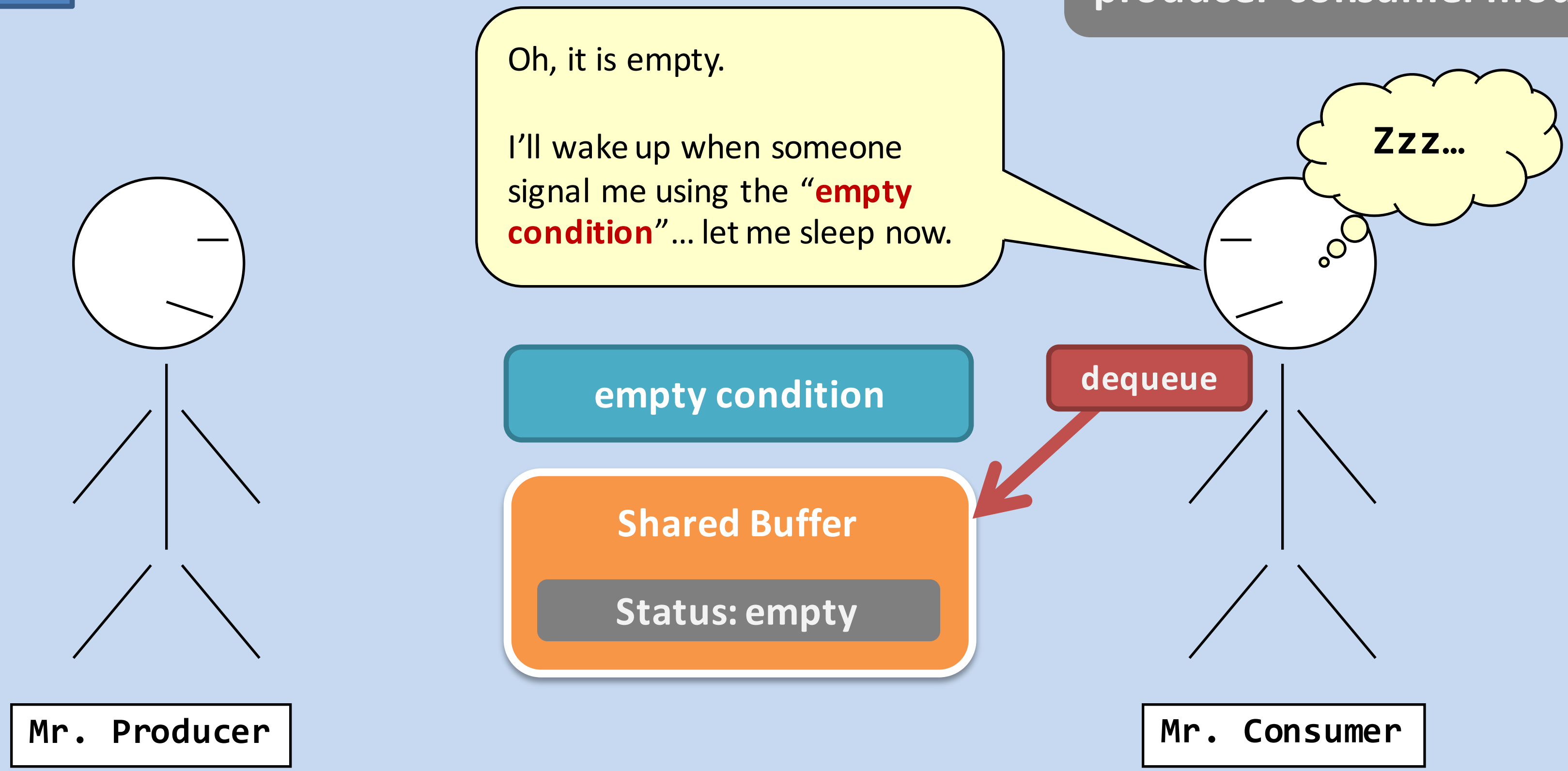
Synchronization…without semaphore?

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Case 1.

An illustration using producer-consumer model



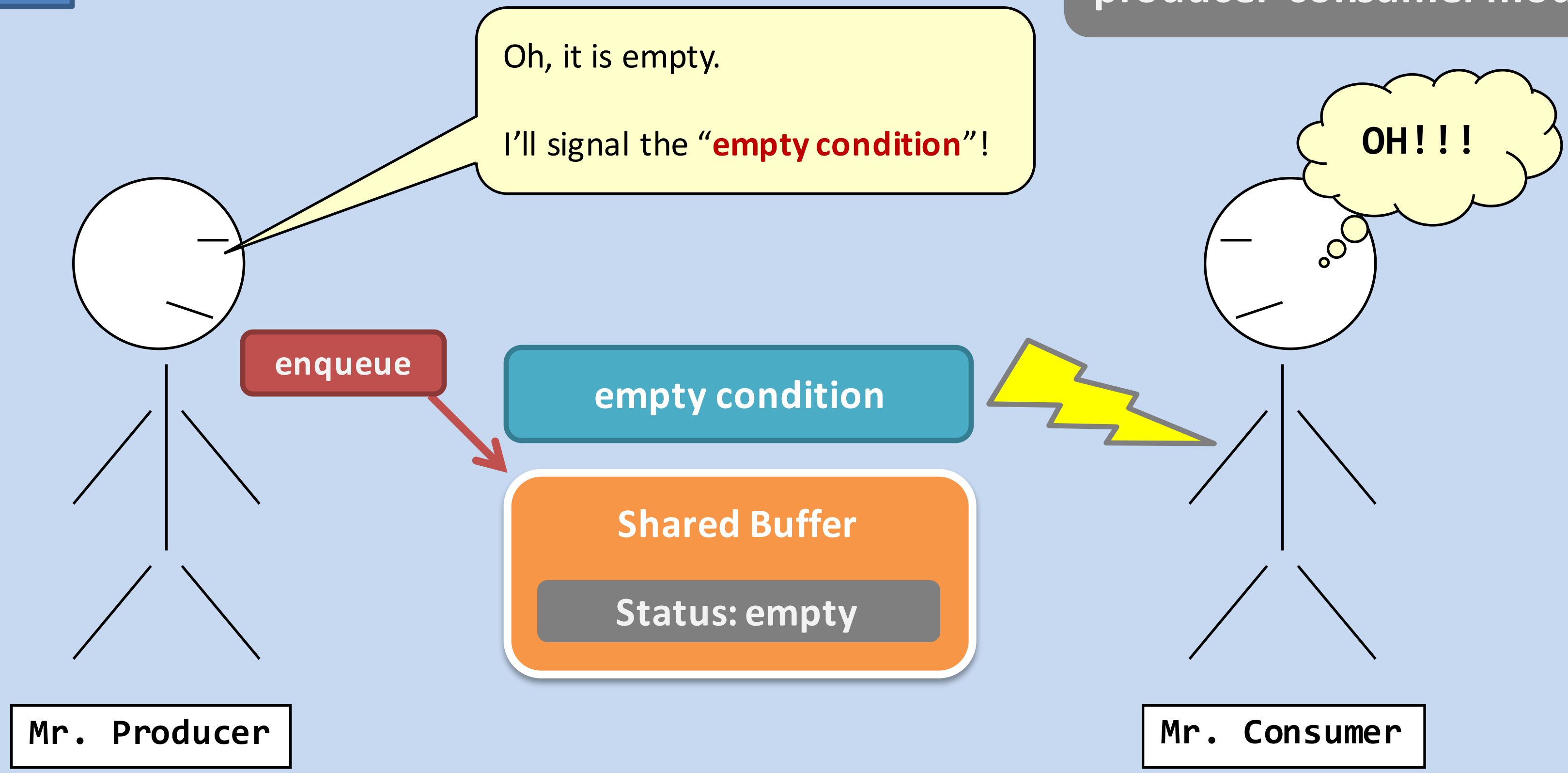
Synchronization…without semaphore?

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Case 1.

An illustration using producer-consumer model



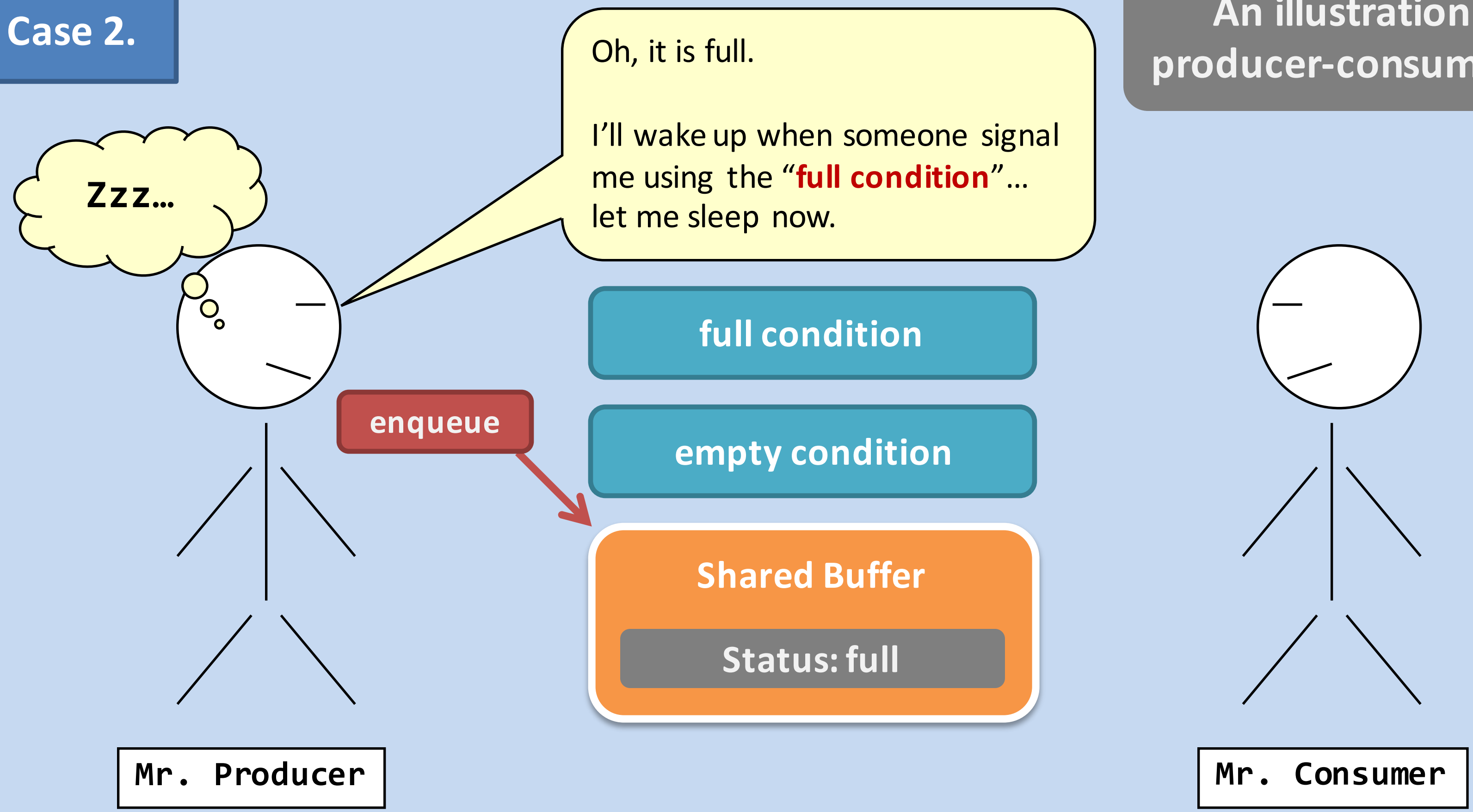
Synchronization…without semaphore?

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Case 2.

An illustration using producer-consumer model



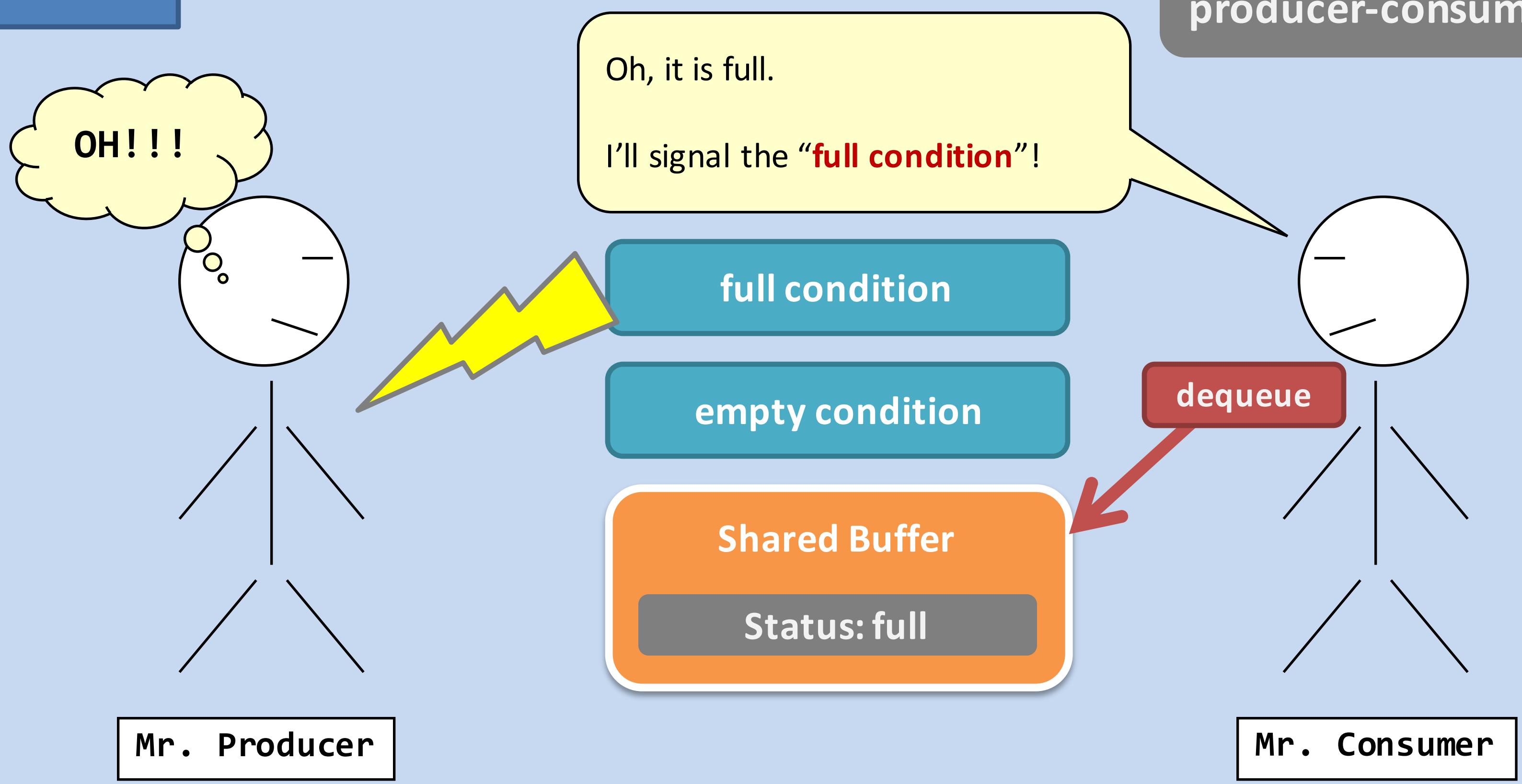
Synchronization…without semaphore?

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Case 2.

An illustration using producer-consumer model



Synchronization…using condition…

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Initialization #1	`pthread_cond_t condition = PTHREAD_COND_INITIALIZER;`
Initialization #2	`pthread_cond_init(&condition, NULL);`
Waiting #1	`pthread_cond_wait(&condition, &mutex);`

This function is to set the thread to wait on a condition variable. **But, why is there a "`mutex`" variable**?

It is there to break the "hold-and-wait" deadlocking condition. Before a thread reaches the waiting statement, **it is usually inside a critical section**. Therefore…[think about it by yourself].

When the thread is about it be blocked, the state of the "**`mutex`**" variable will be **unlocked "atomically" and automatically**.

When the thread is unblocked, the "**`mutex`**" variable is **locked "atomically" and automatically**.

Synchronization…using condition…

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Initialization #1	`pthread_cond_t condition = PTHREAD_COND_INITIALIZER;`
Initialization #2	`pthread_cond_init(&condition, NULL);`
Waiting #1	`pthread_cond_wait(&condition, &mutex);`
Waiting #2	`pthread_cond_timedwait(&condition, &mutex, &timeout);`

This is a fantastic function. At least, you can find it useful in getting out of an indefinite sleep.

The timeout variable is of the type "**struct timespec**". It should store the real time that the timeout reaches.

struct timespec		
time_t	tv_sec	Seconds
long	tv_nsec	Nano-second, 1×10^{-9}

Synchronization…using condition…

Data type
`pthread_cond_t`

This is a condition variable. Its main purpose is to **let a thread to wait** under certain conditions.

Initialization #1	`pthread_cond_t condition = PTHREAD_COND_INITIALIZER;`
Initialization #2	`pthread_cond_init(&condition, NULL);`
Waiting #1	`pthread_cond_wait(&condition, &mutex);`
Waiting #2	`pthread_cond_timedwait(&condition, &mutex, &timeout);`
Wakeup #1	`pthread_cond_signal(&condition);` This function is to wake up a thread which is waiting on the "**`condition`**" variable. If there are more than one thread waiting, then **at least one of those threads will wake up**.
Wakeup #2	`pthread_cond_broadcast(&condition);` This function is to **wake up all threads** which is waiting on the "**`condition`**" variable.

Producer-consumer – in **pthread**

Global variables

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  full  = PTHREAD_COND_INITIALIZER;
pthread_cond_t  empty = PTHREAD_COND_INITIALIZER;
/* the buffer structure */
```

Producer function

```
pthread_lock(&mutex);

if buffer is full ; then
  pthread_cond_wait(&full, &mutex);

if buffer is empty, then
  pthread_cond_signal(&empty);

produce & enqueue an item.

pthread_unlock(&mutex);
```

Consumer Function

```
pthread_lock(&mutex);

if buffer is empty ; then
  pthread_cond_wait(&empty, &mutex);

if buffer is full, then
  pthread_cond_signal(&full);

dequeue & consume an item.

pthread_unlock(&mutex);
```

```
[examples@3150] cat pthread_producer_consumer.c
```

How about other IPC problems?

- It seems that we have to **rethink** all the IPC solutions that we learnt before…
  - Is there any method to relieve such pain?

Semaphore data type

```
pthread_mutex_t mutex;
pthread_cond_t  cond;
int             value
```

void down(Semaphore *s)

```
pthread_mutex_lock(s->mutex);
while(s->value == 0)
    pthread_cond_wait(&s->cond, &s->mutex);
s->value--;
pthread_mutex_unlock(&s->mutex);
```

Extra & Challenge.

Implement the solutions to those IPC problems using the pthread library!

void up(Semaphore *s)

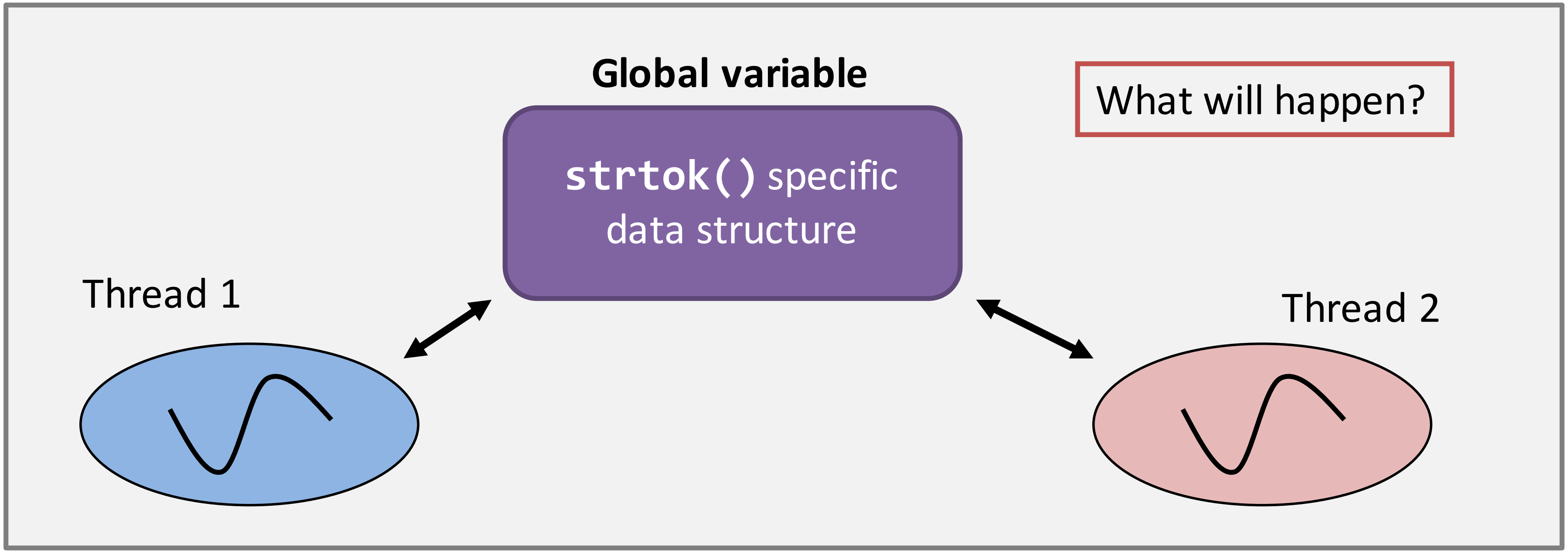
```
pthread_mutex_lock(s->mutex);
if(s->value == 0)
    pthread_cond_signal(&s->cond);
s->value++;
pthread_mutex_unlock(&s->mutex);
```

Multi-threading

- Introduction.
- Basic Programming.
- Thread scheduling?
- Mutual exclusion
  & synchronization!
- **Thread safety?!**

Thread Safety – an important concern

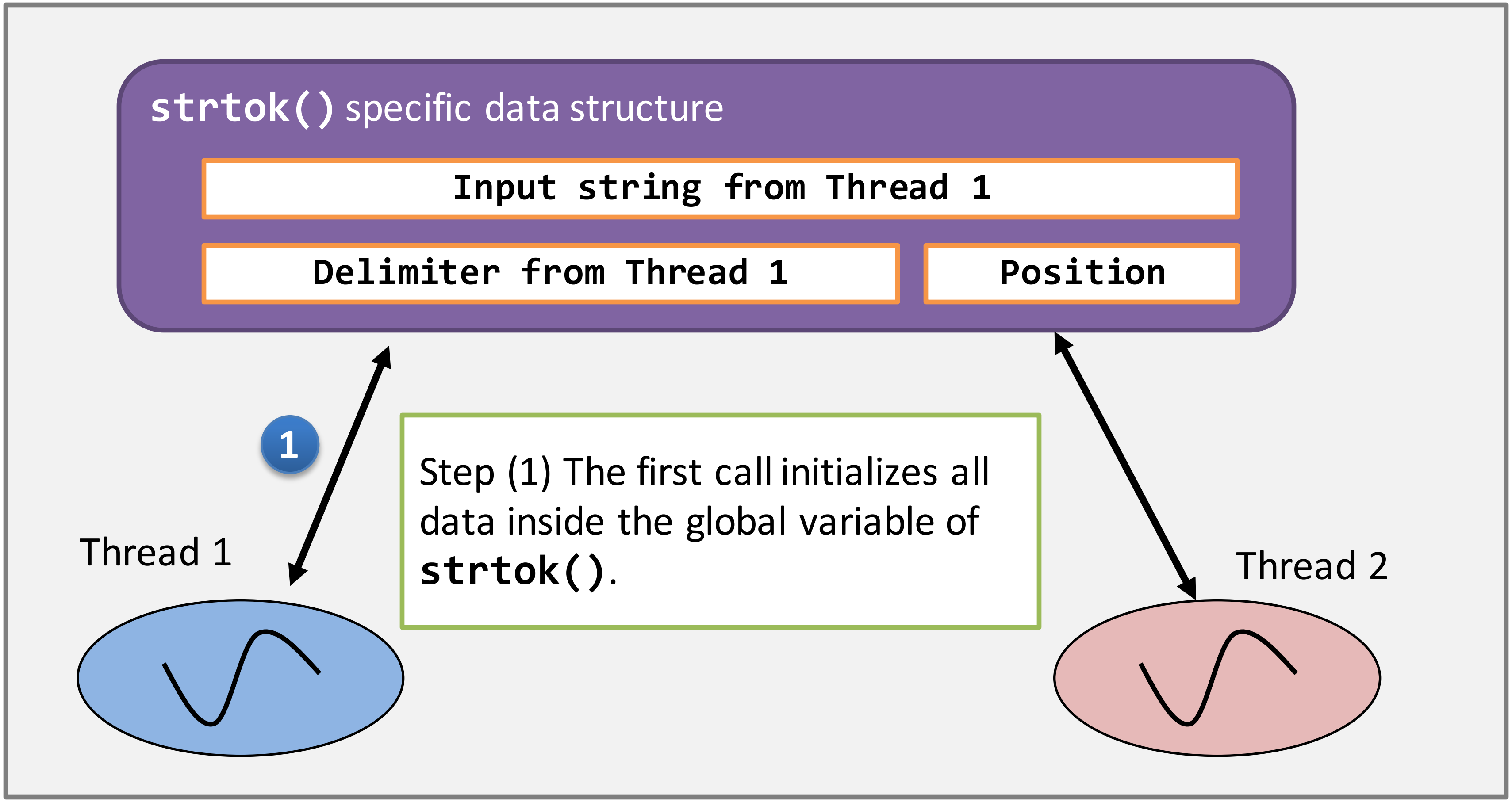
- Some functions, by design, has assumed a **single-threaded execution**. E.g., `strtok()`;
  - When such a function is deployed in a multi-threaded execution, we face a problem called **thread safety**.




```
[examples@3150] cat normal_strtok.c pthread_strtok.c
```

Thread Safety – an important concern

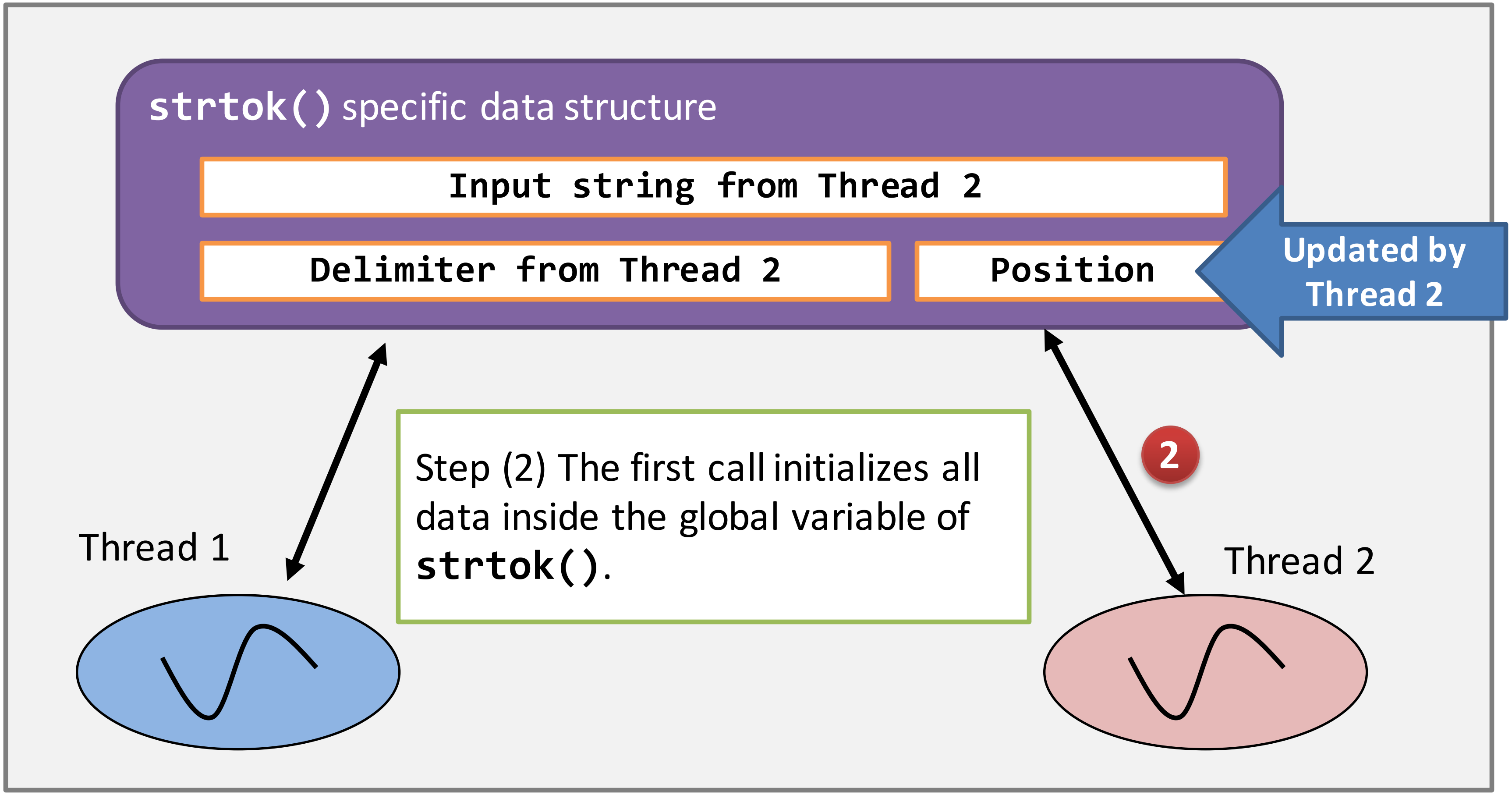
- Illustration of thread-unsafe function (1 of 3)




```
[examples@3150] cat normal_strtok.c pthread_strtok.c
```

Thread Safety – an important concern

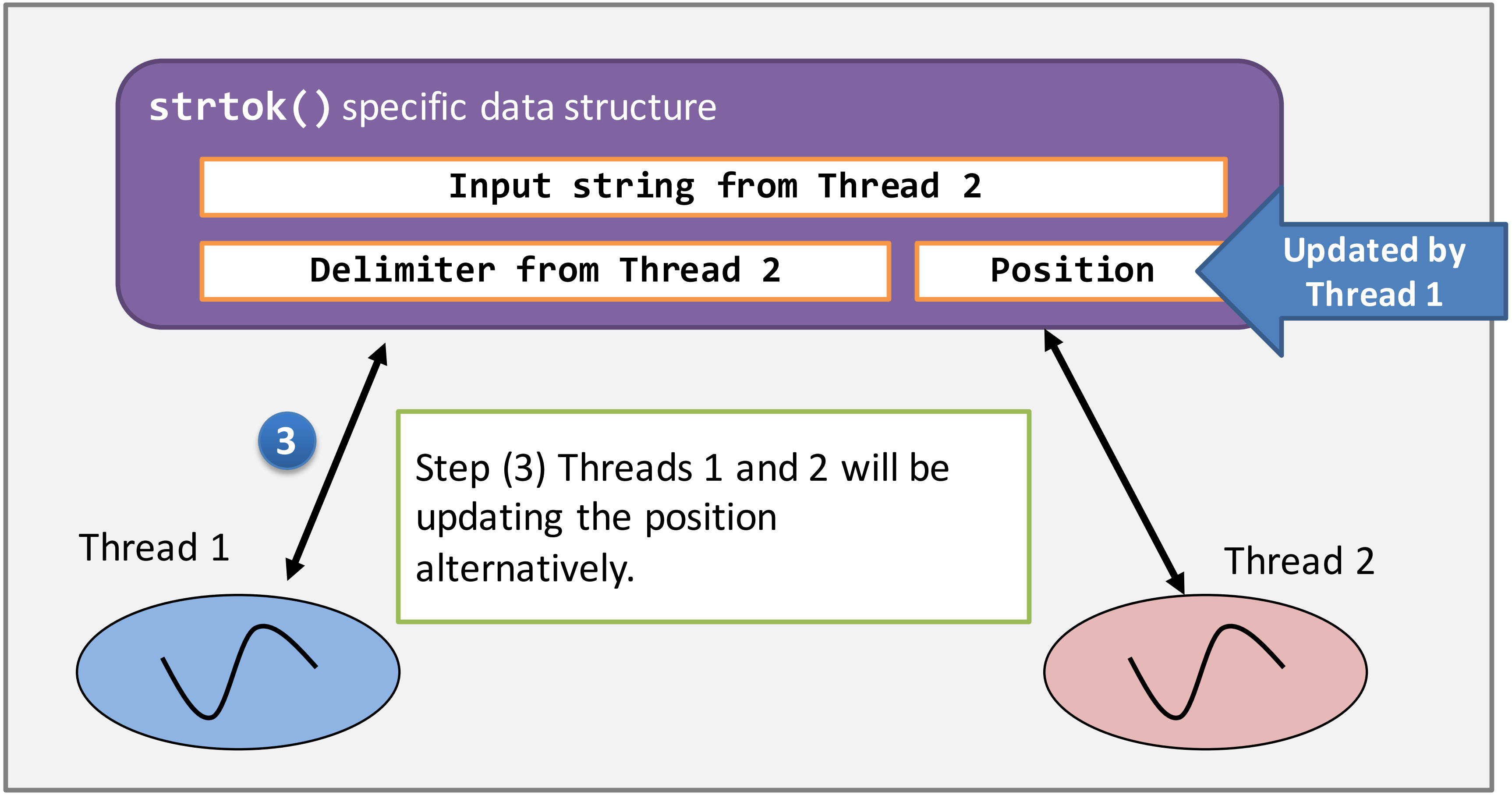
- Illustration of thread-unsafe function (2 of 3)




```
[examples@3150] cat normal_strtok.c pthread_strtok.c
```

Thread Safety – an important concern

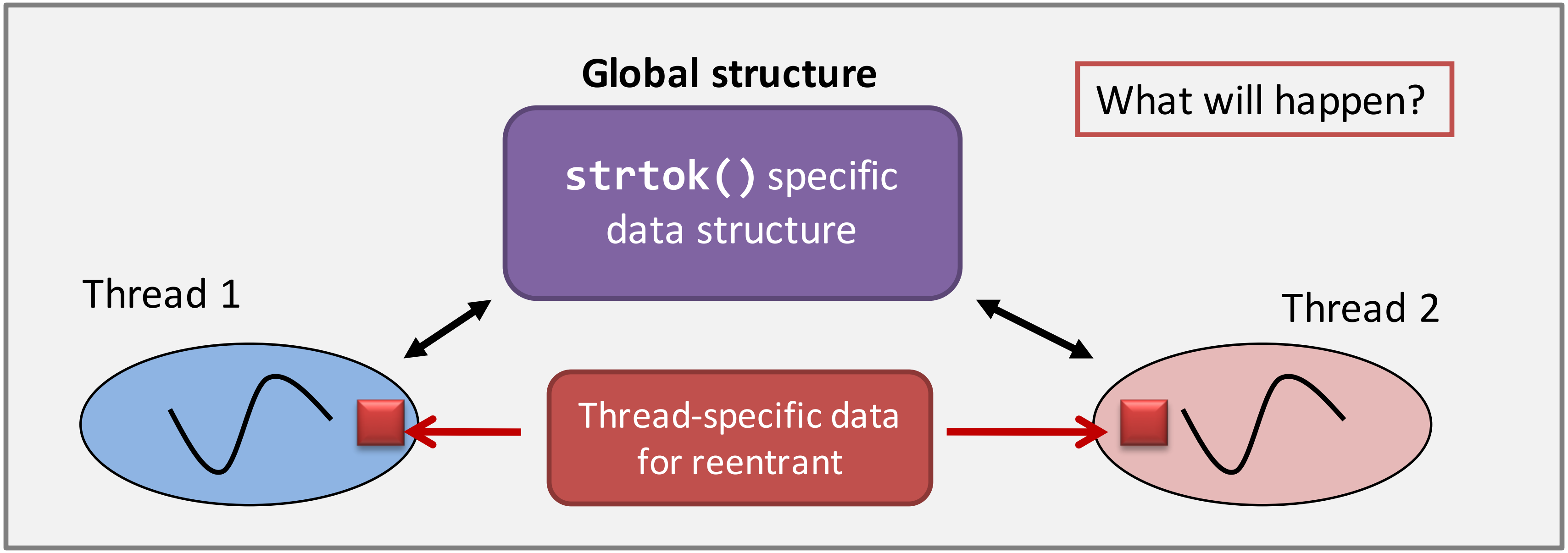
- Illustration of thread-unsafe function (3 of 3)




```
[examples@3150] cat normal_strtok.c pthread_strtok.c
```

Thread Safety – an important concern

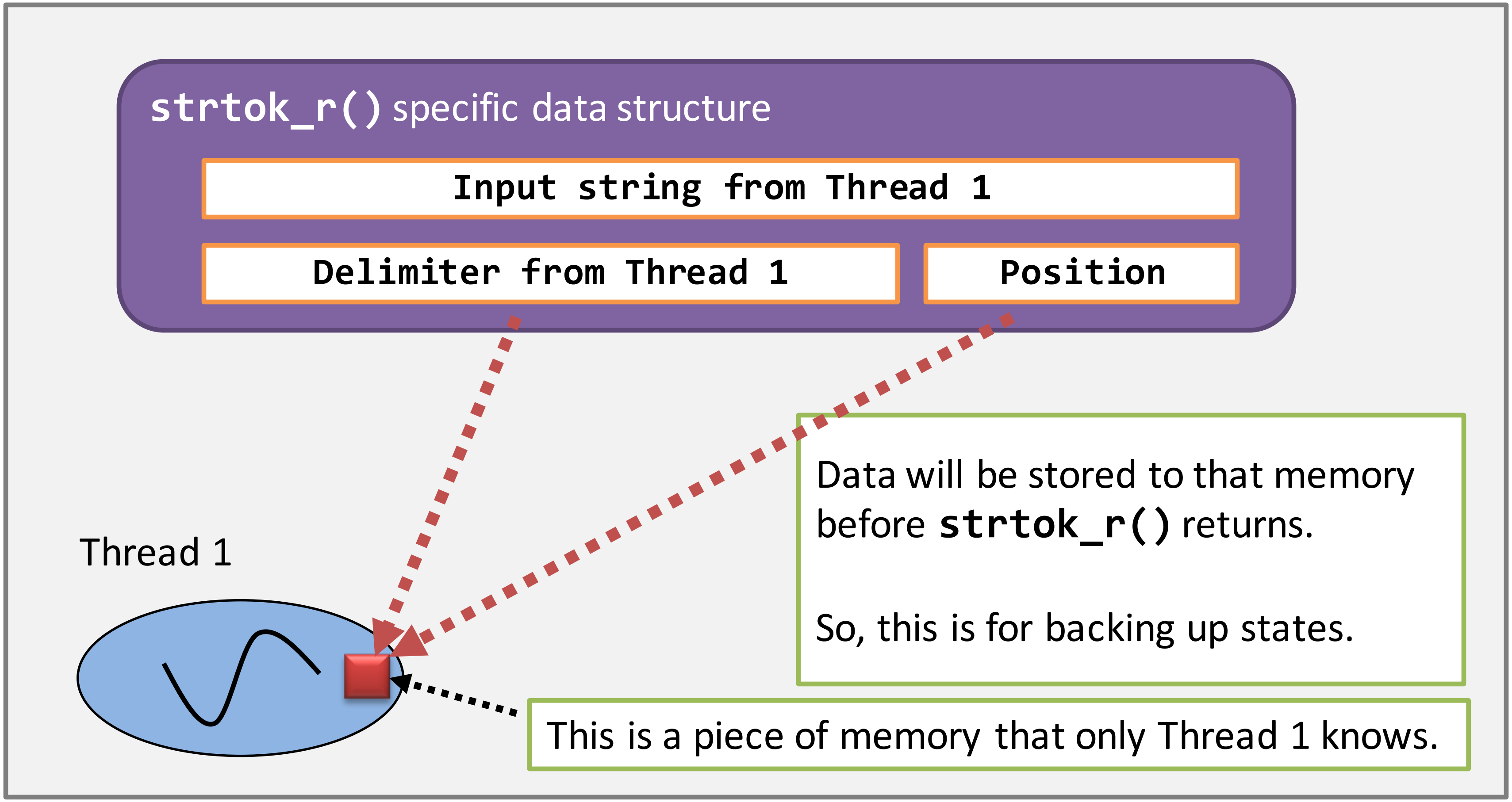
- A thread-safe function is a function that has **the reentrant property** implemented.
  - The **`strtok()`** thread-safe counterpart: **`strtok_r()`**.




```
[examples@3150] cat pthread_strtok_r.c
```

Thread Safety – an important concern

- Using re-entrant data.




```
[examples@3150] cat normal_strtok.c pthread_strtok.c
```

Conclusion

- Threading is a convenient tool to achieve the following purposes:
  - Multi-tasking within a process;
    - E.g., GUI is a thread and the main method is another one.
  - Simple shared memory environment.

- But, you have to take care of:
  - Mutual exclusion;
  - Synchronization; and
  - Thread safety.

- You should **take great care** when writing both multi-threaded and multi-processed programs.

Chapter Conclusion

- We have finished the **Chapter of Process Management**. Hope you enjoyed!