Operating System Design

Threads

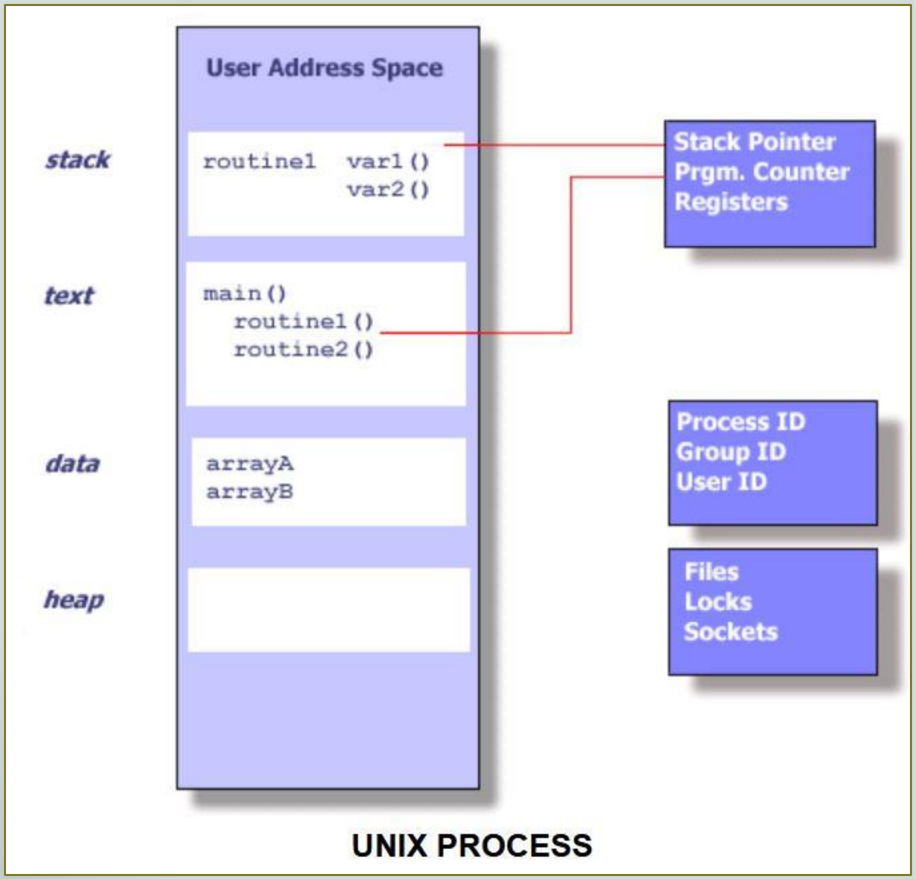
Neda Nasiriani
Fall 2018

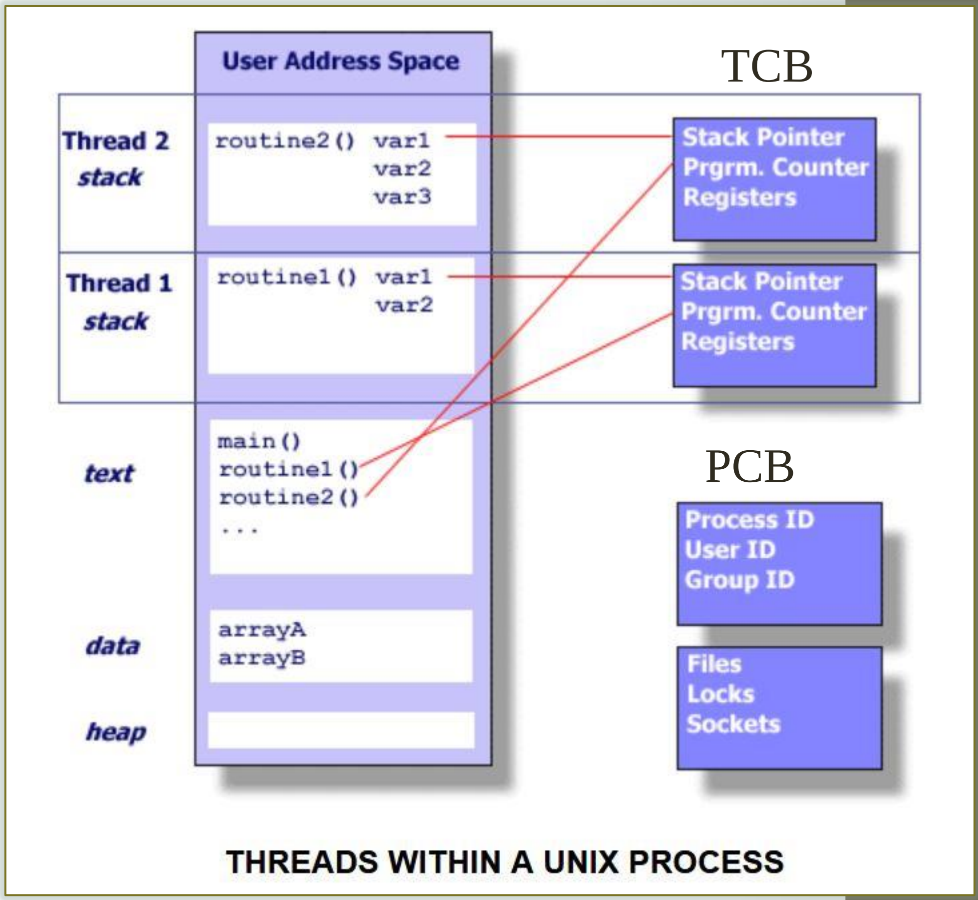
Why Threads?

- Threads use and exist within the process resources
  - are able to be scheduled by the operating system
  - run as independent entities

- How can this independent flow of execution be accomplished for threads
  - Stack pointer (and stack space)
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data

PCB vs TCB





https://computing.llnl.gov/tutorials/pthreads/#Overview

FYR:Why Threads?

- A thread exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently (and dependently)
  - Dies if the parent process dies - or something similar
- Is "**lightweight**" because most of the overhead has already been accomplished through the creation of its process.
  - Because threads within the same process share resources: Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory locations is possible, and therefore requires explicit **synchronization** by the programmer

Why Threads?

- No Inter Process Communication (IPC) is necessary
- The only limit is the memory bandwidth which is way more than the shared memory bandwidth as an IPC among processes

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

https://computing.llnl.gov/tutorials/pthreads/#Overview

When use threads?

- The program should be parallelizable and can be broken down to independent tasks which can run in parallel
- “Several common models for threaded programs exist:
  - ***Manager/worker:*** a single thread, the *manager* assigns work to other threads, the *workers*
  - ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread
  - ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work”

https://computing.llnl.gov/tutorials/pthreads/#Overview

How are Threads Scheduled?

- Assume process P creates 5 threads T_1, T_2, T_3, T_4, T_5 in this order T_2, T_1, T_3, T_5, T_4
  - Which one of these threads executes first?
  - Which one of these threads finish its execution first?
  - On which core is thread T_3 scheduled to run? (if there are 4 cores)
  - The answer to all these questions is **WE DON'T KNOW**
- A good multi-threaded program successful execution should be independent of order of execution of its threads
- What can we control?
  - The pthreads API provides several routines that may be used to specify how threads are scheduled for execution
    - FIFO (first-in first-out)
    - RR (round-robin)
    - OTHER (operating system determines)
  - pthreads API also provides the ability to set a thread's scheduling priority value.
  - The Linux operating system may provide a way to set the CPU core to execute the process on using the sched_setaffinity routine.

Threads Synchronization

- If main() finishes before the created threads exit, all of the threads will be terminated because the main thread of execution is terminated
- How can we avoid this?
  - If main thread calls pthread_exit() as the last thing it does, main() will **block and be kept alive** to support the threads it created until they are done.
  - Using pthread_join(.) can block the thread to wait for the spawned threads

Demo

./thread_join_retval
./thread_no_join_retval

comment join, then comment pthread_exit in main

gcc -pthread thread_no_join_retval.c -o thread_no_join_retval -lm

User Threads and Kernel Threads

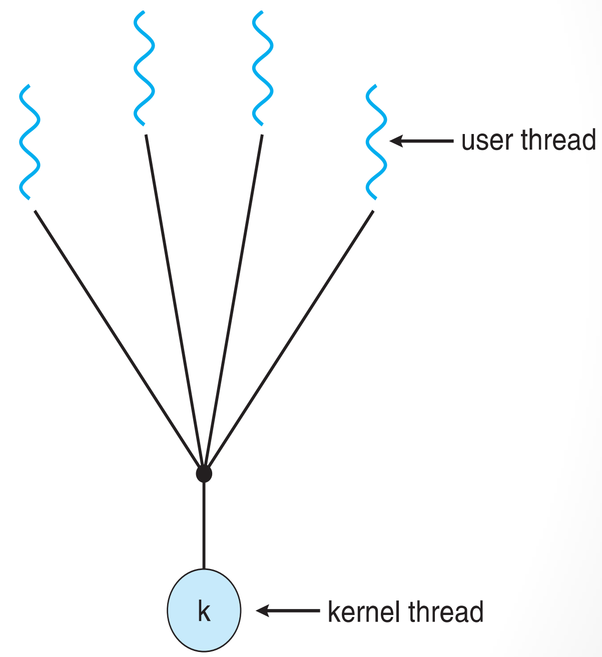
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

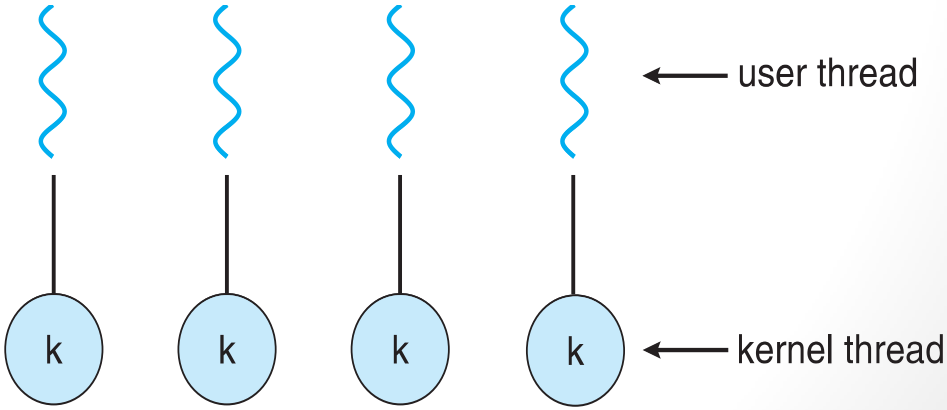
Many to One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



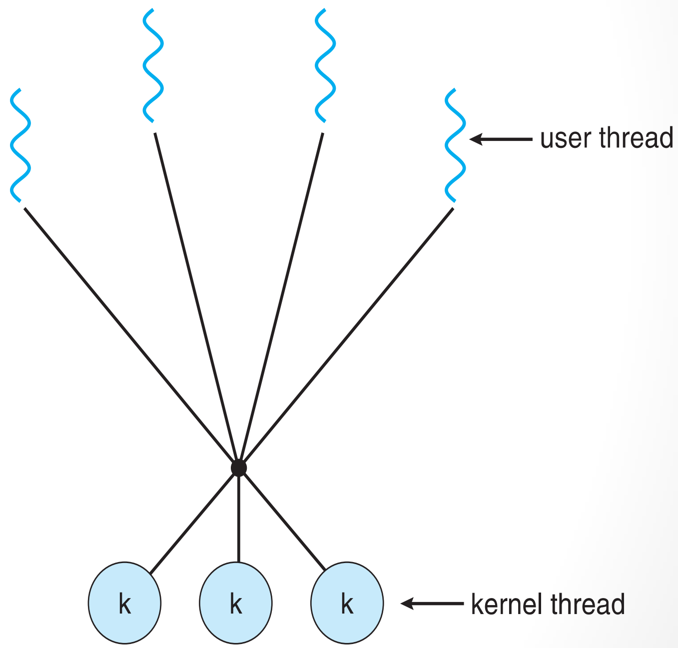
One to One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



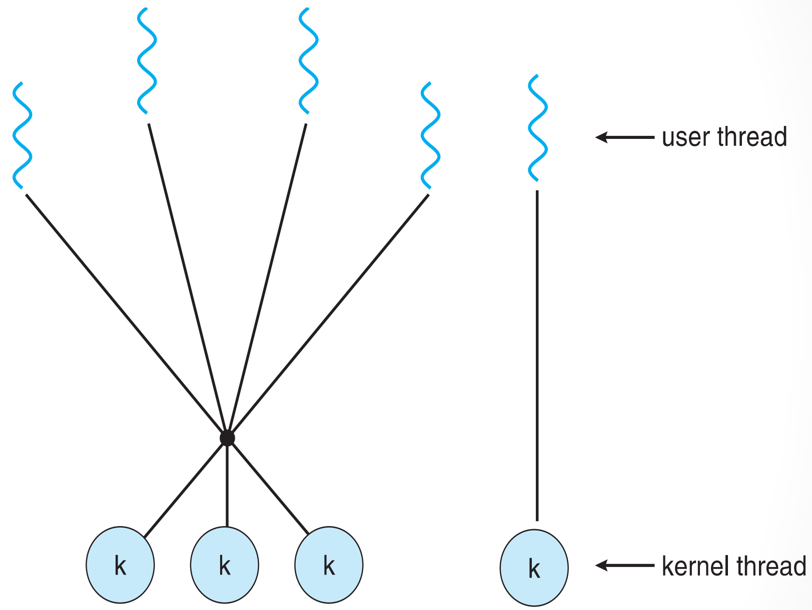
Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

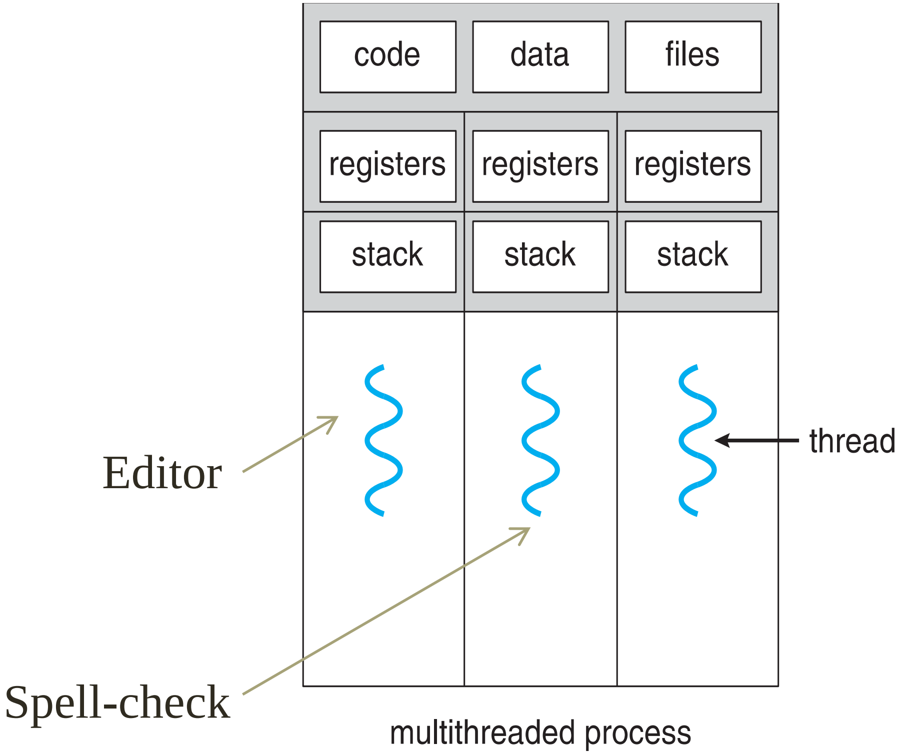


Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

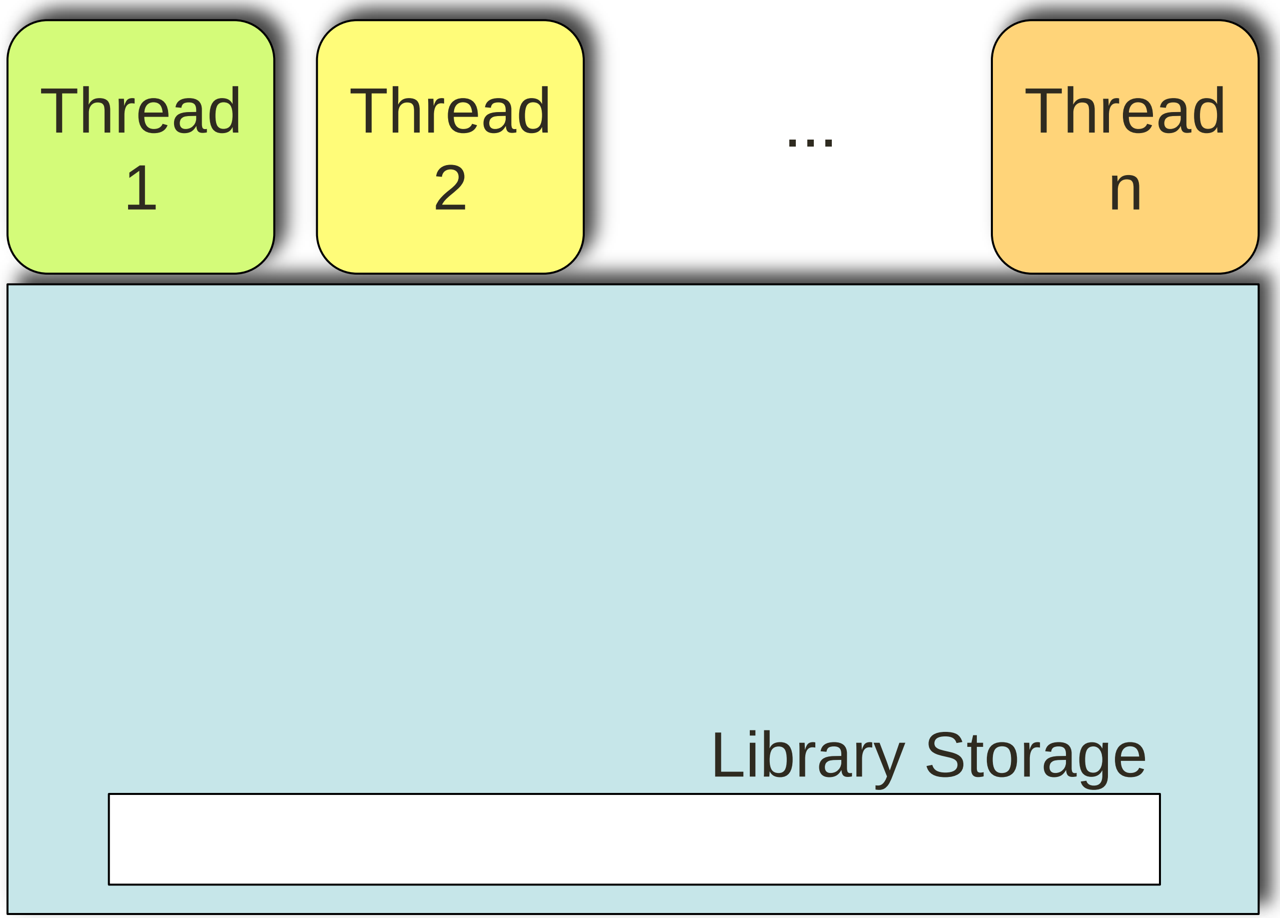


Shared Memory Model

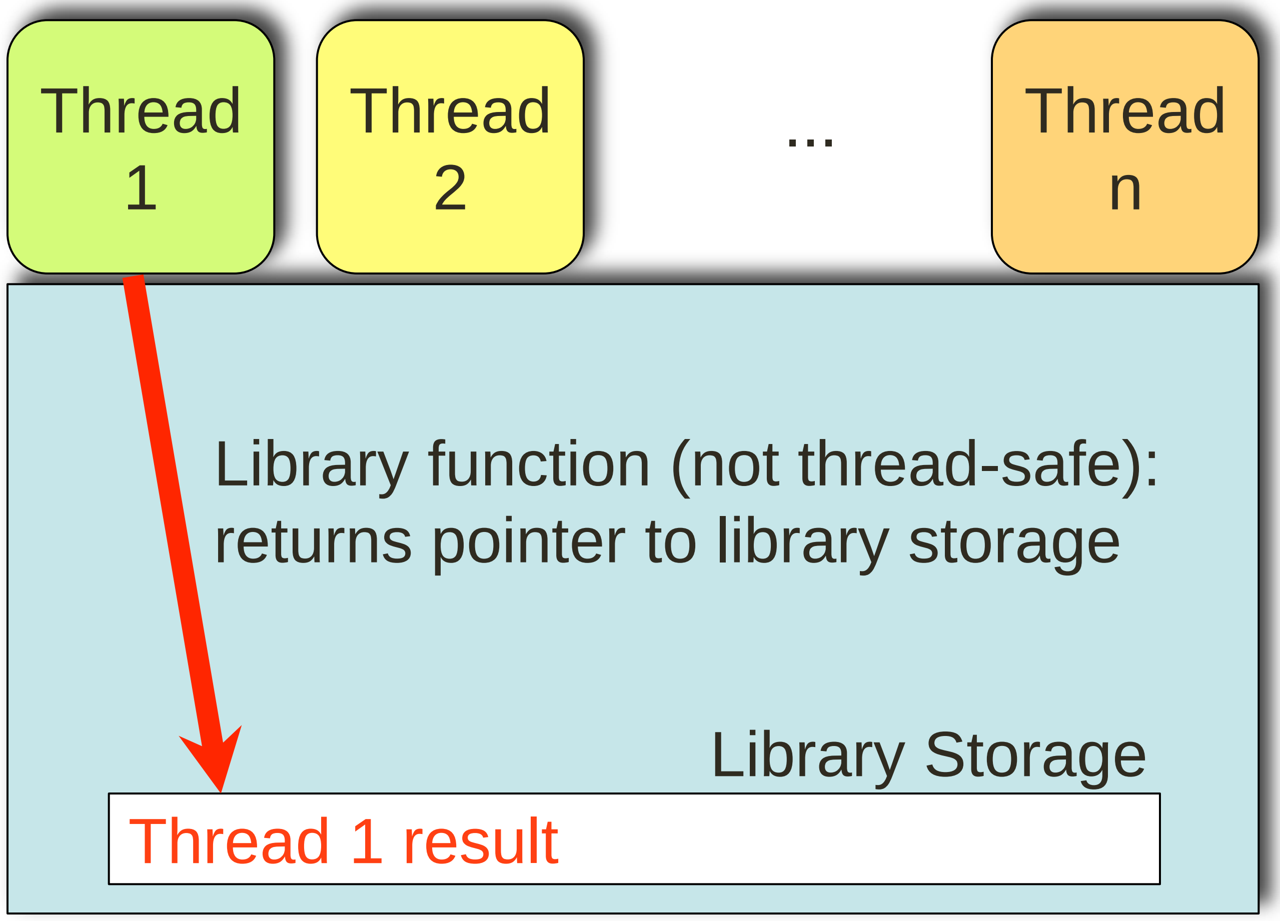


- All threads have access to the same global, shared memory
- Threads also have their own private data (how?)
- Programmers are responsible for protecting globally shared data

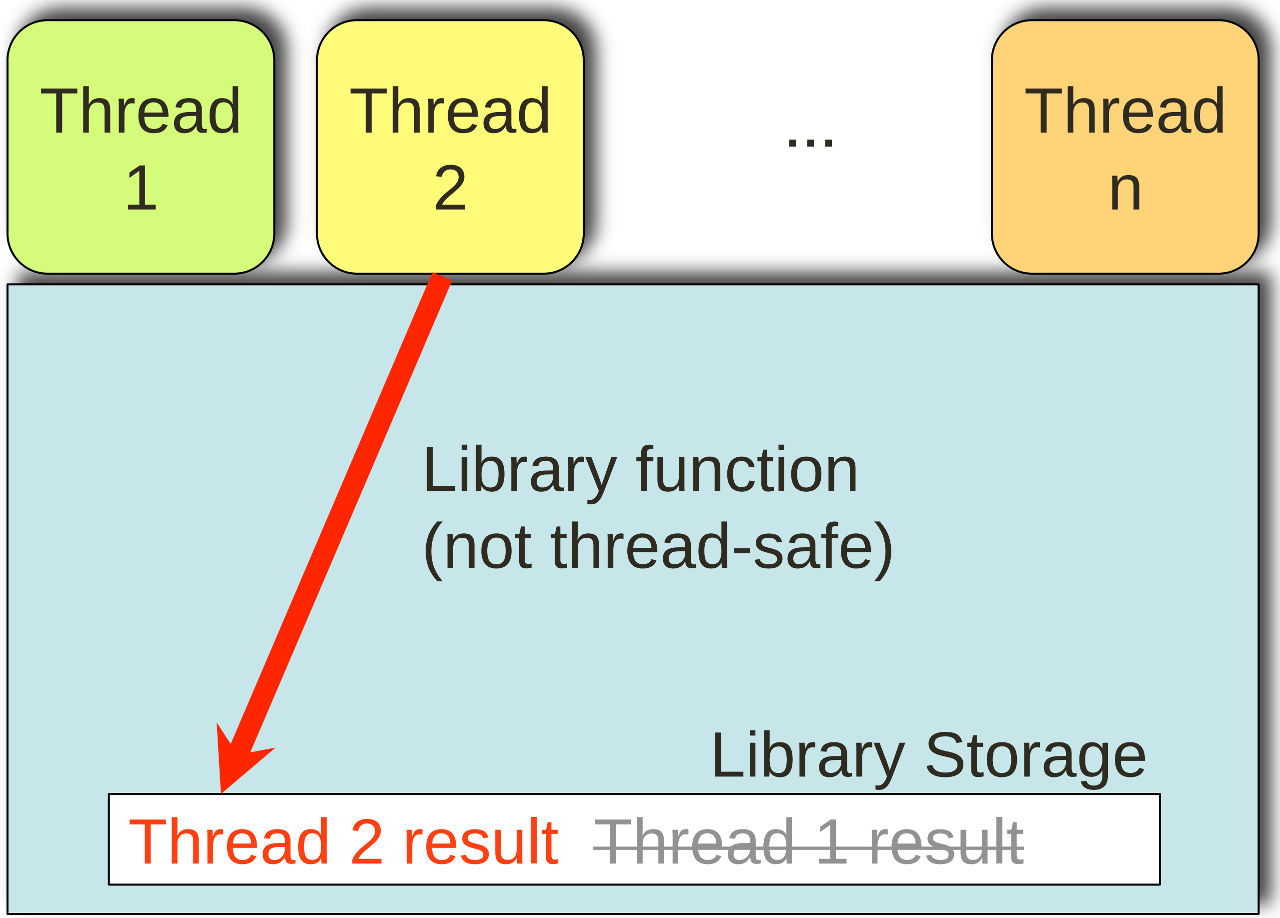
Thread Safeness



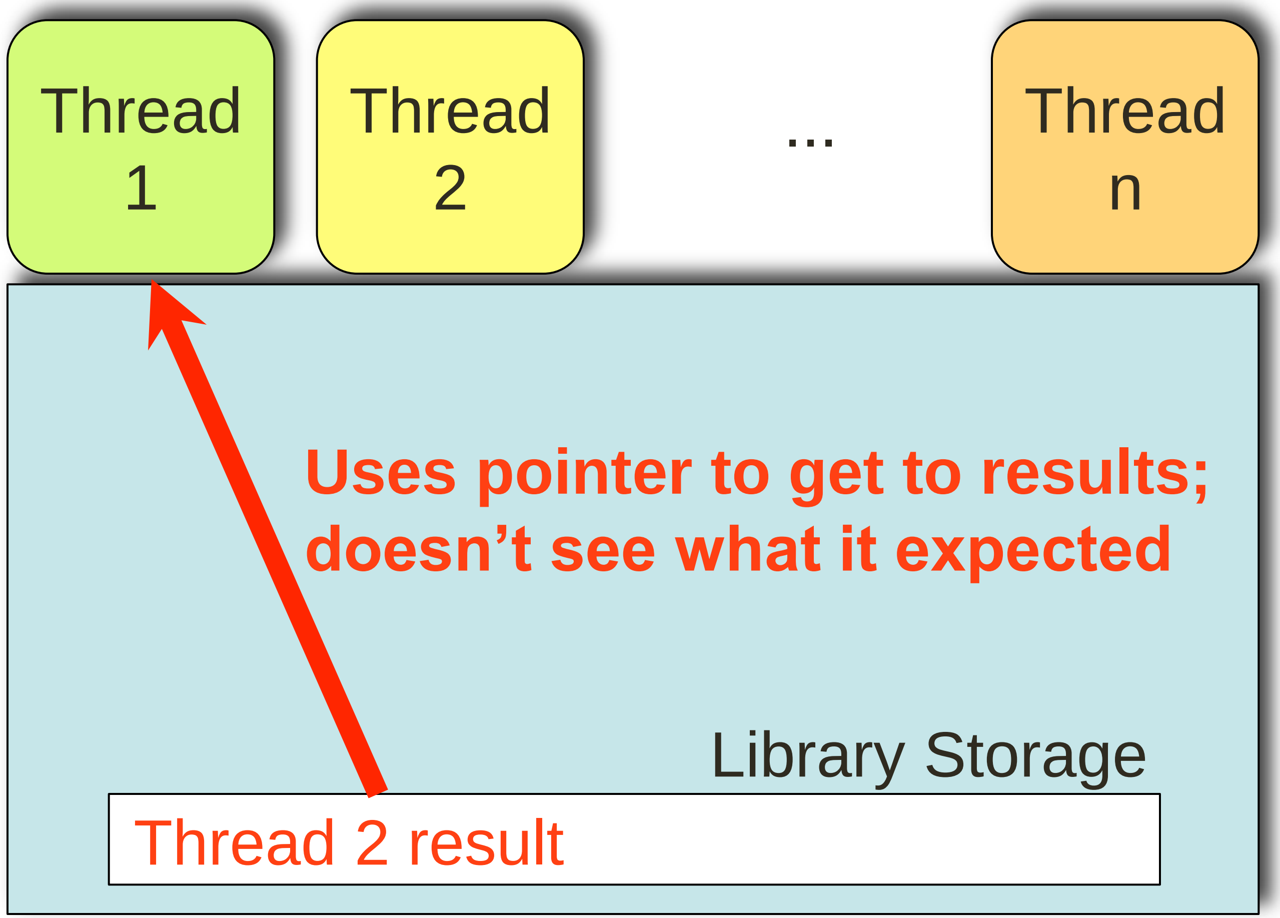
Thread Safeness



Thread Safeness



Thread Safeness



Create Threads

```
NAME
    pthread_create - create a new thread

SYNOPSIS
    #include <pthread.h>

    int pthread_create(pthread_t *thread,
                       const pthread_attr_t *attr,
                       void *(*start_routine) (void *),
                       void *arg);

    Compile and link with –pthread
```

- (*start_routine) is a functin pointer to a function that returns a void * and has one argument of type void *
- pthread_t is a unsigned long (%lu)

Thread Termination

- If any thread within a process calls exit, then the entire **process** terminates
- A thread can exit in three ways
  1. Return from the start routine. The return value is the thread's exit code
  2. The thread can be canceled by another thread in the same process
  3. The thread can call pthread_exit

❖ To allow other threads to continue execution, the main thread should terminate by calling pthread_exit() rather than exit().

NAME

pthread_exit – terminate calling thread

void pthread_exit (void *retval);

Thread Join and Return Value

- If a thread has return values from its start routine, it can send it to other threads in the process by calling pthread_exit (void* retval) or simply returning a pointer of type void * to the return value
- retval is a type-less pointer like the input argument for pthread_create
- How can other threads access this value?
  - If a thread needs an input argument from another thread it can use **join function** to block its execution until the other thread exits

NAME

pthread_join – calling thread will block until the specific thread calls pthread_exit

```
pthread_join (pthread_t tid, void **retval_ptr);
```

retval_ptr has the return value of the thread with ID tid

Thread Input Argument and Return Value (Output Argument)

- The typeless pointer passed to pthread_create and pthread_exit can be used to pass the address of a structure containing more complex information.
- Be careful that the memory used for the structure is still valid when the caller has completed.
  - If the **input structure** was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used.
  - If a thread allocates an **output structure** on its stack and passes a pointer to this structure to pthread_exit, then the stack might be destroyed and its memory reused for something else by the time the caller of pthread_join tries to use it.

Examples

Good idea?!?
Why?

```
void * thr_fn2(void *arg)
{
    int tmp = 6;
    printf("thread 2 exiting\n");
    pthread_exit((void *) &tmp);
}
```

```
int rc; long t;
for(t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
    ...
}
```

Good idea?!?
Why?

Other Thread Operations

```
NAME
    pthread_self – returns the thread ID
    pthread_t pthread_self (void);

NAME
    pthread_equal – compare thread IDs
    pthread_equal (pthread_t tid1, pthread_t tid2);
```

- Just as every process has a process ID, every thread has a thread ID. Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs.
- pthread_t data type can be implemented as a structure. Therefore, a function must be used to compare two thread IDs.

Activity Answer!

Example

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

```
void * sum(void * a){
    int * val = (int *)a;
    int result = 0;
    for (int i = 1; i <= *val; i++){
        result += i;
    }
    printf("Running thread %lu value in thread %d \n",pthread_self(),result);
    return (NULL);

}

int main(){
    int val = 100;
    int * a = &val;
    void * (*func)(void *) = &sum;

    pthread_t id;
    int err;
    err = pthread_create(&id, NULL, func,(void *)a);
    printf("thread create ret val %d\n",err);
    pthread_join(id, NULL);
    return 0;
}
```

Passing Multiple Arguments

```
struct input{
    int a;
    int b;
};
```

```
void * sum(void * a){
    struct input * val = (struct input *)a;
    int result = val->a + val->b;
    return ((void *) &result);

}
```

```
int main(){
    struct input args;
    void * retval;
    args.a = 10;
    args.b = 20;
    pthread_t id;
    int err;

    err = pthread_create(&id, NULL, sum,(void *)&args);
    printf("thread create ret val %d\n",err);
    pthread_join(id, &retval);
    printf("thread return value is %d\n",(*(int *)retval));
    return 0;
}
```

Passing Multiple Arguments

```
void * sum(void * a){
    int * val = (int *)a;
    int result = 0;
    for (int i = 1; i <= *val; i++){
        result += i;
    }
    printf("Running thread %lu value in thread %d \n",pthread_self(),result);
    return (NULL);

}
```

Good idea?!?
Why?

Return Value – Bad Practice!

```
struct input{
    int a;
    int b;
};
```

```
void * sum(void * a){
    struct input * val = (struct input *)a;
    int result = val->a + val->b;
    return ((void *) &result);

}
```

```
int main(){
    struct input args;
    void * retval;
    args.a = 10;
    args.b = 20;
    pthread_t id;
    int err;

    err = pthread_create(&id, NULL, sum,(void *)&args);
    printf("thread create ret val %d\n",err);
    pthread_join(id, &retval);
    printf("thread return value is %d\n",(*(int *)retval));
    return 0;
}
```

```
thread create ret val 0
thread return value is 32609
```

Return Value – Good Practice!

```c
struct input{
    int a;
    int b;
    int result;
};
```

```c
void * sum(void * a){
    struct input * val = (struct input *)a;
    val -> result = val->a + val->b;
    return ((void *) a);

}

int main(){
    struct input args;
    void * retval;
    args.a = 10;
    args.b = 20;
    pthread_t id;
    int err;
    err = pthread_create(&id, NULL, sum,(void *)&args);
    printf("thread create ret val %d\n",err);
    pthread_join(id, &retval);
    printf("thread return value is %d\n",((struct input *)retval)->result);
    return 0;
}
```

```
thread create ret val 0
thread return value is 30
```

Multiple Threads

```
struct input{
    int a;
    int b;
    int result;
};
```

```
#define NUM_THREADS 5
struct input args[NUM_THREADS];
pthread_t ids[NUM_THREADS];
```

```
void * sum(void * a){
    struct input * val = (struct input *)a;
    val -> result = val->a + val->b;
    printf("Thread with index %d is running now\n",val->a/10);
    return ((void *) a);

}
```

```
int main(){
    void * retval[NUM_THREADS];
    for (int i=0; i < NUM_THREADS; i++){
        args[i].a = i * 10;
        args[i].b = i * 20;

        int err;
        err = pthread_create(&ids[i], NULL, sum,(void *)&args[i]);
        printf("thread %d return val %d\n",i,err);
    }

    for (int i = 0; i < NUM_THREADS; i++){
        pthread_join(ids[i], &retval[i]);
        printf("thread return value is %d\n",((struct input *)retval[i])->result);
    }
    return 0;
}
```

```
thread 0 return val 0
thread 1 return val 0
Thread with index 0 is running now
thread 2 return val 0
Thread with index 1 is running now
Thread with index 2 is running now
thread 3 return val 0
thread 4 return val 0
thread return value is 0
thread return value is 30
Thread with index 3 is running now
Thread with index 4 is running now
thread return value is 60
thread return value is 90
thread return value is 120
```