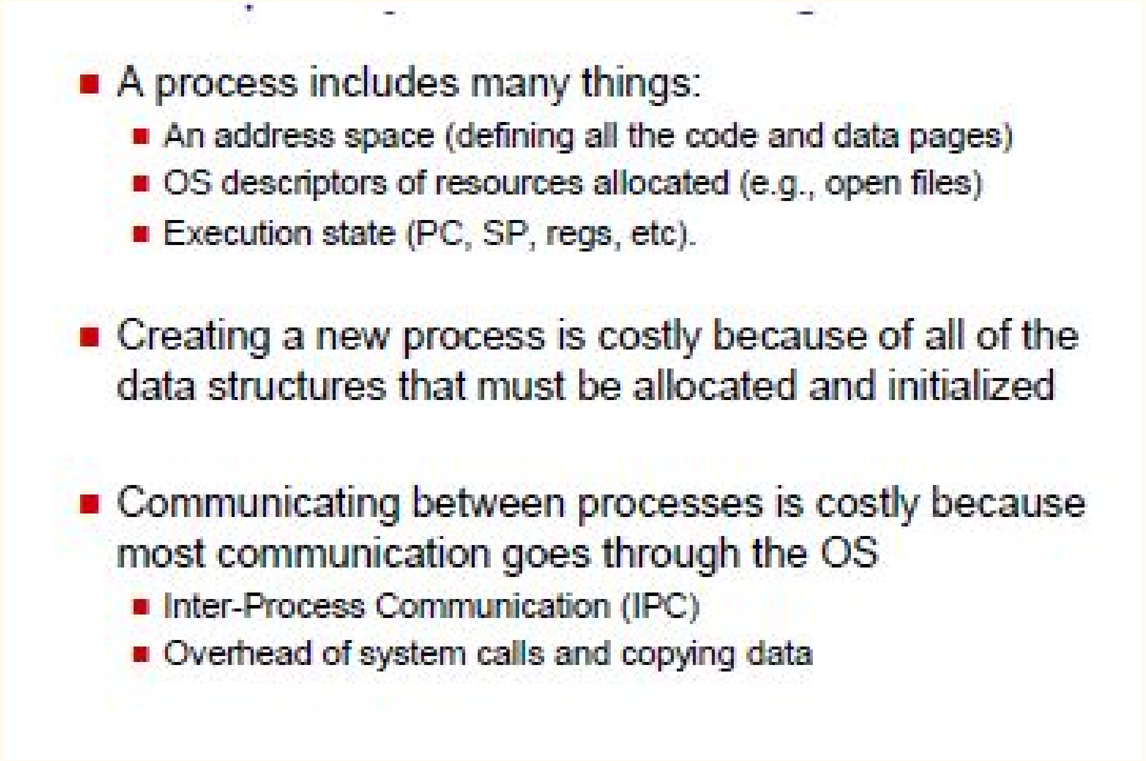
Recitation 8

Multithreading

Multithreading

1. Multithreading a program appears to do more than one thing at a time.
2. The idea of The idea of Multithreading Multithreading is same as Multiprogramming i.e. multitasking but within a single process.
3. E.g. a word processor has separate threads:
   a. One for displaying graphics
   b. Other for reading in keystrokes from the user
   c. Another to perform spelling and grammar checking in the background.

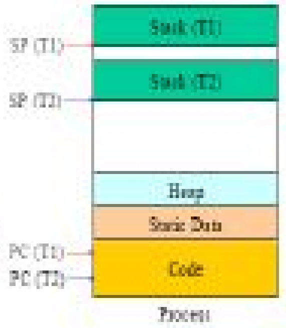
Why Do Multithreading?



- A process includes many things:
  - An address space (defining all the code and data pages)
  - OS descriptors of resources allocated (e.g., open files)
  - Execution state (PC, SP, regs, etc).
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
- Communicating between processes is costly because most communication goes through the OS
  - Inter-Process Communication (IPC)
  - Overhead of system calls and copying data

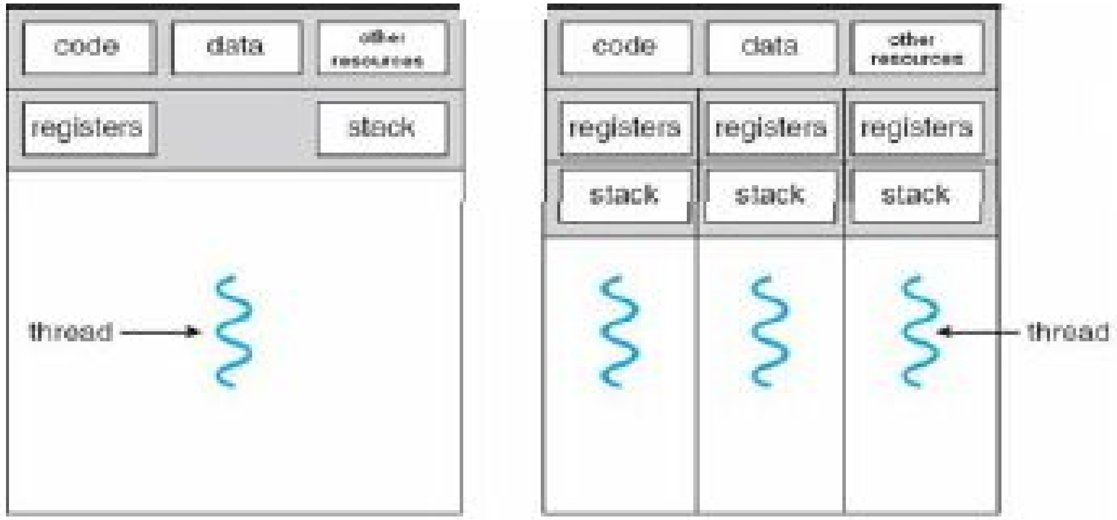
Multithreading

- Allow process to be subdivided into different threads of control
- A *thread* is the smallest schedulable unit in multithreading
- A thread in execution works with
  - thread ID
  - Registers (program counter and working register set)
  - Stack (for procedure call parameters, local variables etc.)
- A thread *shares* with other threads a process's (to which it belongs to)
  - Code section
  - Data section (static + heap)
  - Permissions
  - Other resources (e.g. files)



Process with 2 threads

Difference b/w single vs. multithread processes



single-threaded process multithreaded process

- A process by itself can be viewed a single thread and is traditionally known as a heavy weight process

Multithreading vs Multiprocessing

1. Inter-process communication is expensive: need to context switch. It is also secure: one process cannot corrupt another process. Inter-thread communication cheap: can use process memory and may not need to context switch. Not secure: a thread can write the memory used by another thread.
2. Process carry considerable state information. Multiple thread within a process share state as well as memory and other resources.
3. Processes interact only through system-provided inter-process communication mechanisms. There is an overhead of system calls and copying data. Context switching between threads in the same process is typically faster than context switching between processes.

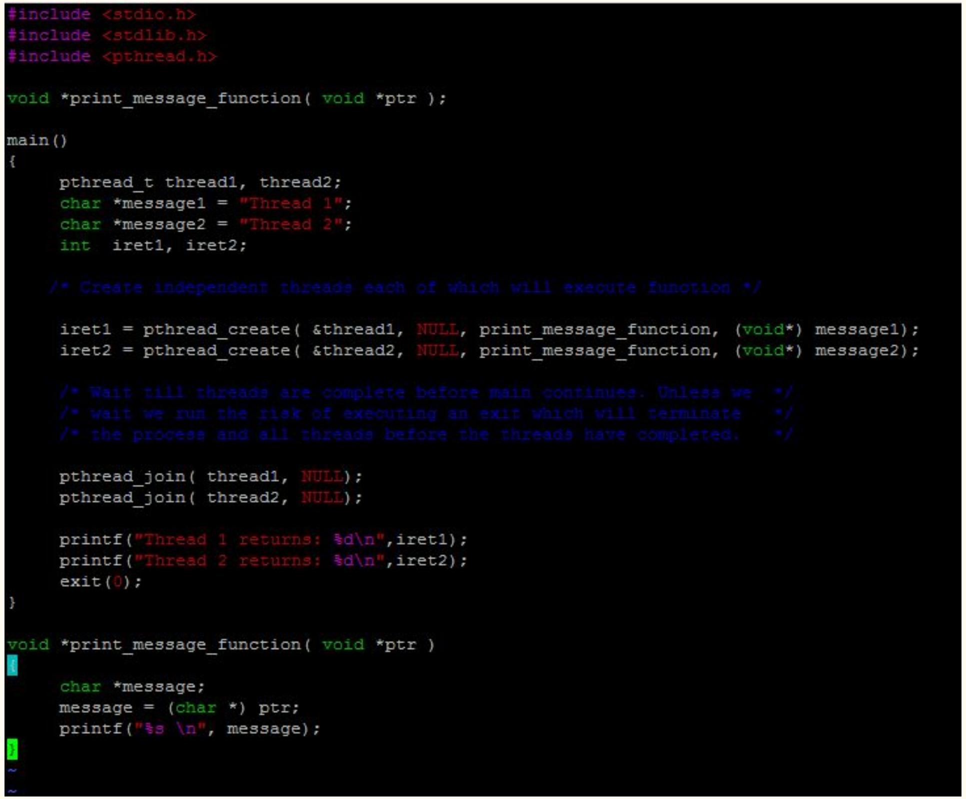
Threads Basics

1. Thread operations include thread creation, termination, synchronization (joining or blocking), scheduling, data management, and process interaction.
2. A thread does not maintain a list of threads it creates, neither does it know the thread that created it.
3. Threads in the same process share:
    a. Same address space
    b. Process instruction
    c. Most data
    d. Open files
    e. Signals and signal handlers
    f. Current working directory

Thread Basics

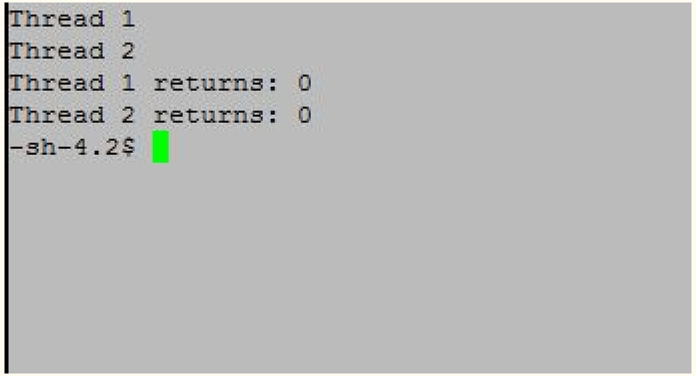
1. Each thread has a unique:
    a. Thread ID
    b. Set of registers, stack pointer
    c. Stack for local variables, return addresses
    d. Signal mask
    e. Return value: errno

Example



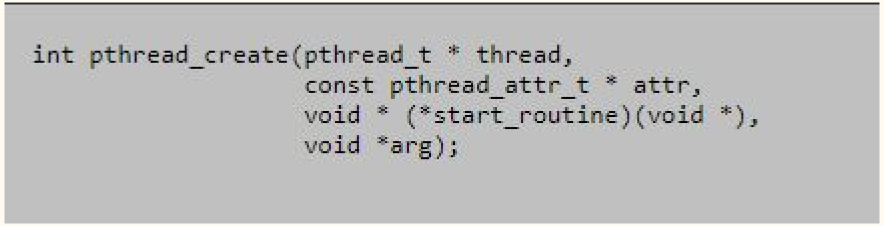
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
~
~
```

Output


```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
-sh-4.2$
```

Details

1. In this example, the same function is used in each thread
2. Threads terminate by explicitly calling **pthread_exit,** by letting the function return, or by a call to the function **exit,** which will terminate the process including any threads.

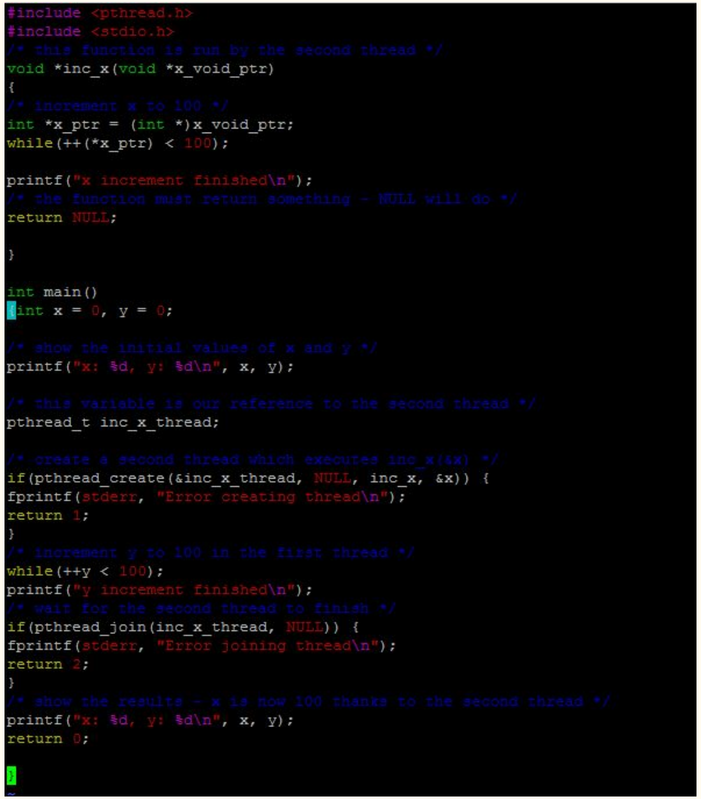


```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

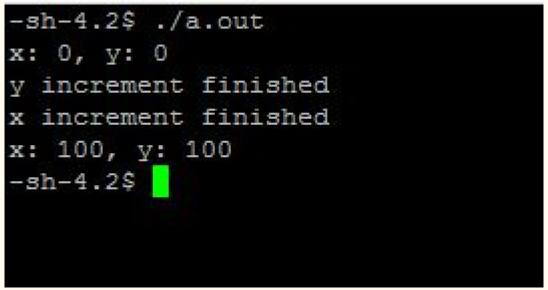
Function call pthread_create

1. **thread:** returns the thread id.
2. **attr:** Set to null if the default thread attributes are used. Other examples of attributes: For e.g. if you want to create a joinable thread: PTHREAD_CREATE_JOINABLE.
3. **void *(*start_routine):** pointer to the function to be threaded. Function has a single argument: pointer to void.
4. ***arg:** pointer to argument of function. If you

Another Example



```c
#include <pthread.h>
#include <stdio.h>
/* this function is run by the second thread */
void *inc_x(void *x_void_ptr)
{
/* increment x to 100 */
int *x_ptr = (int *)x_void_ptr;
while(++(*x_ptr) < 100);

printf("x increment finished\n");
/* the function must return something - NULL will do */
return NULL;

}

int main()
{int x = 0, y = 0;

/* show the initial values of x and y */
printf("x: %d, y: %d\n", x, y);

/* this variable is our reference to the second thread */
pthread_t inc_x_thread;

/* create a second thread which executes inc_x(&x) */
if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
fprintf(stderr, "Error creating thread\n");
return 1;
}
/* increment y to 100 in the first thread */
while(++y < 100);
printf("y increment finished\n");
/* wait for the second thread to finish */
if(pthread_join(inc_x_thread, NULL)) {
fprintf(stderr, "Error joining thread\n");
return 2;
}
/* show the results - x is now 100 thanks to the second thread */
printf("x: %d, y: %d\n", x, y);
return 0;

}
~
```



```
-sh-4.2$ ./a.out
x: 0, y: 0
y increment finished
x increment finished
x: 100, y: 100
-sh-4.2$
```

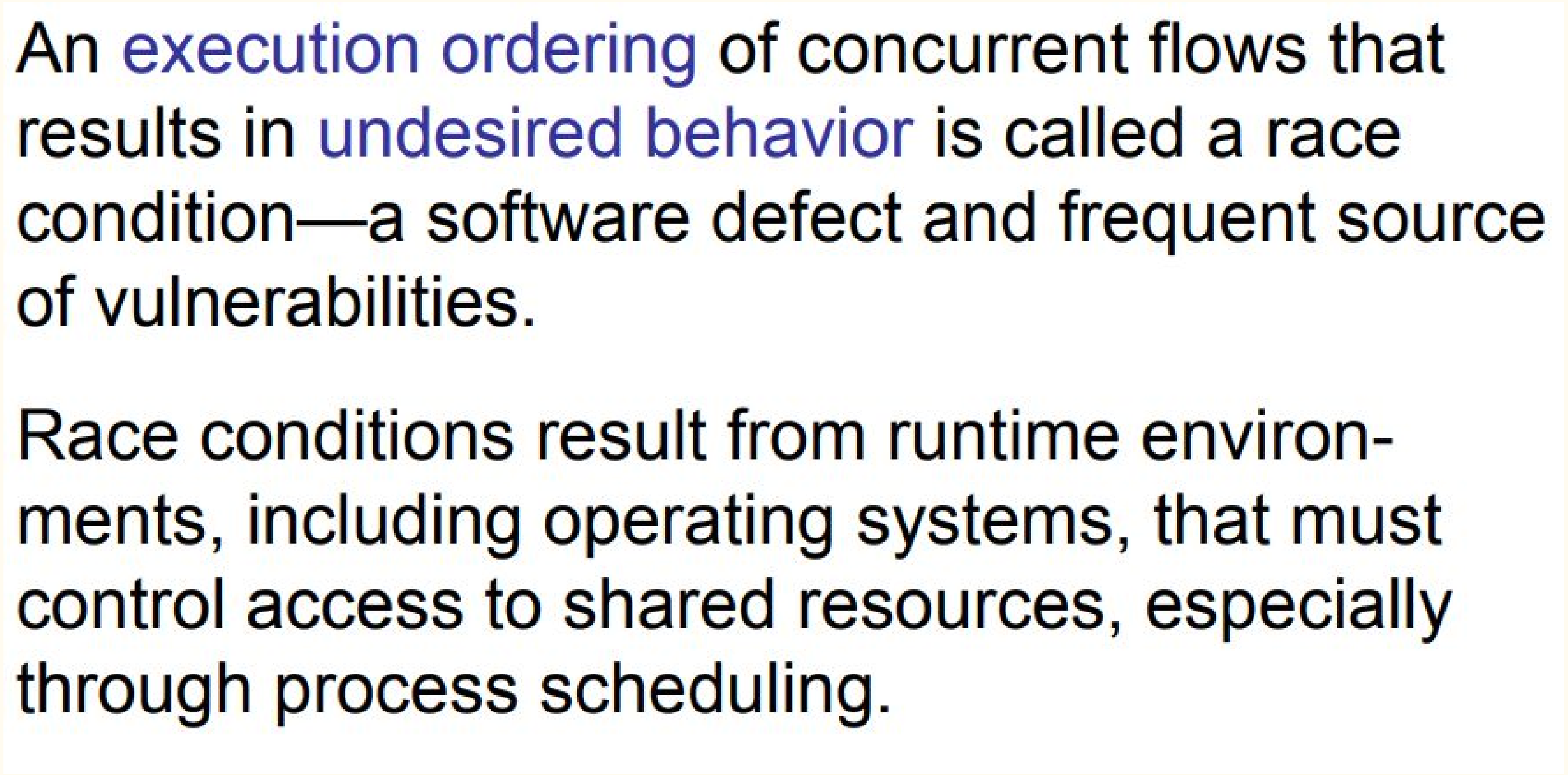
Function call pthread_join

1. **int pthread_join(pthread_t** *thread*, **void *****retval***);**

2. The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

3. On success, **pthread_join**() returns 0; on error, it returns an error number.

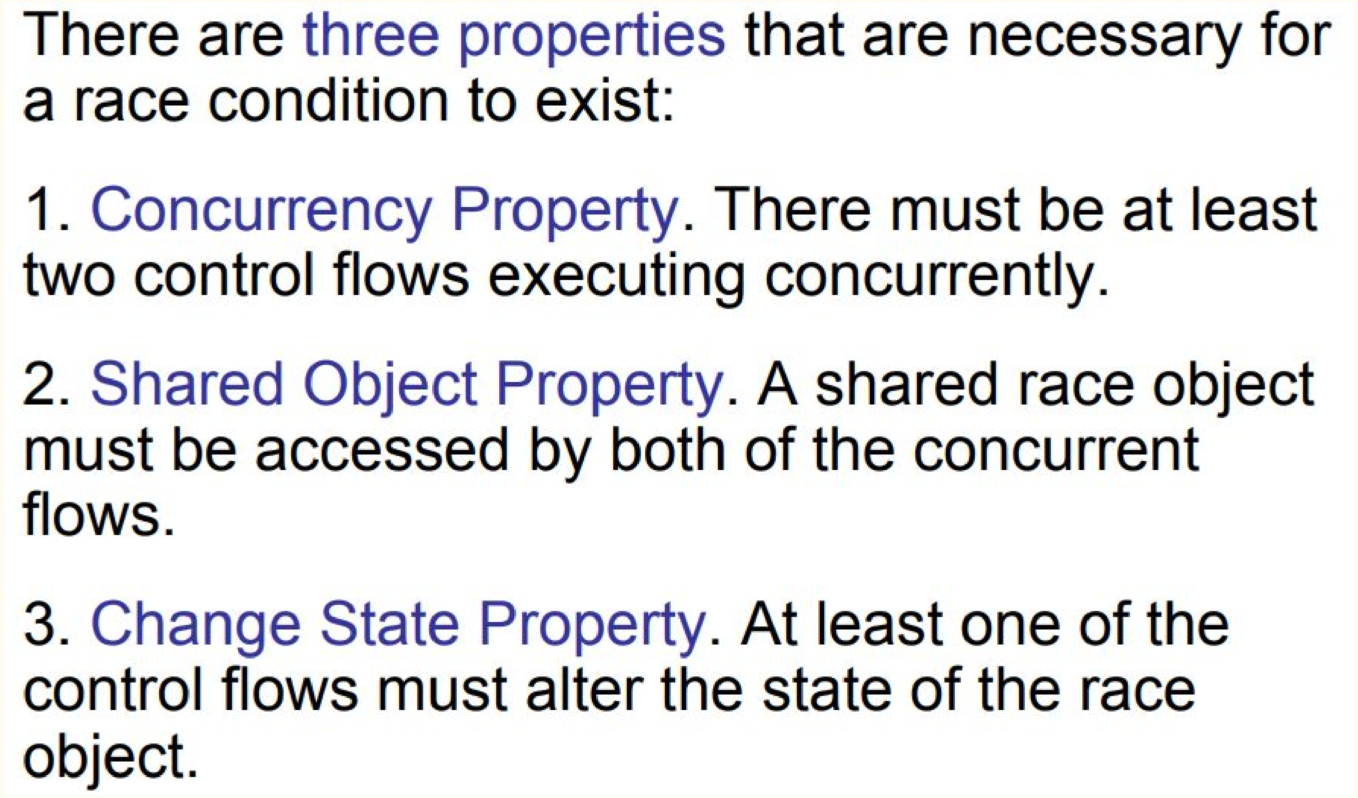
Race Conditions



An execution ordering of concurrent flows that results in undesired behavior is called a race condition—a software defect and frequent source of vulnerabilities.

Race conditions result from runtime environments, including operating systems, that must control access to shared resources, especially through process scheduling.

Race Condition Properties



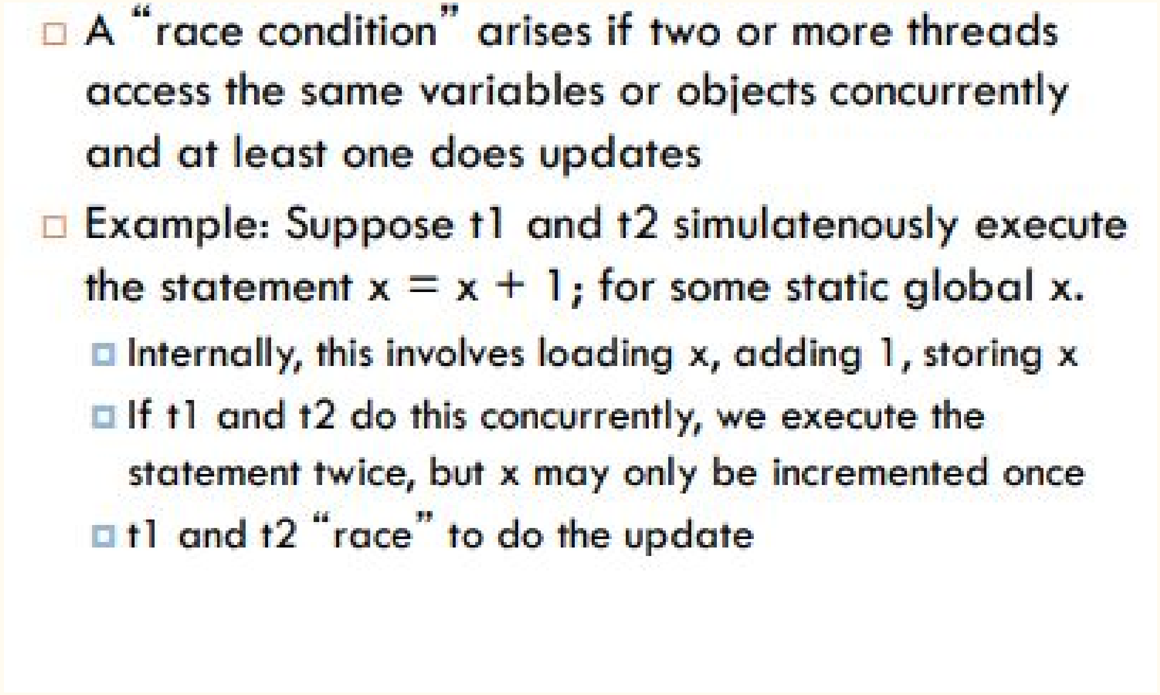
There are three properties that are necessary for a race condition to exist:

1. Concurrency Property. There must be at least two control flows executing concurrently.

2. Shared Object Property. A shared race object must be accessed by both of the concurrent flows.

3. Change State Property. At least one of the control flows must alter the state of the race object.

Example

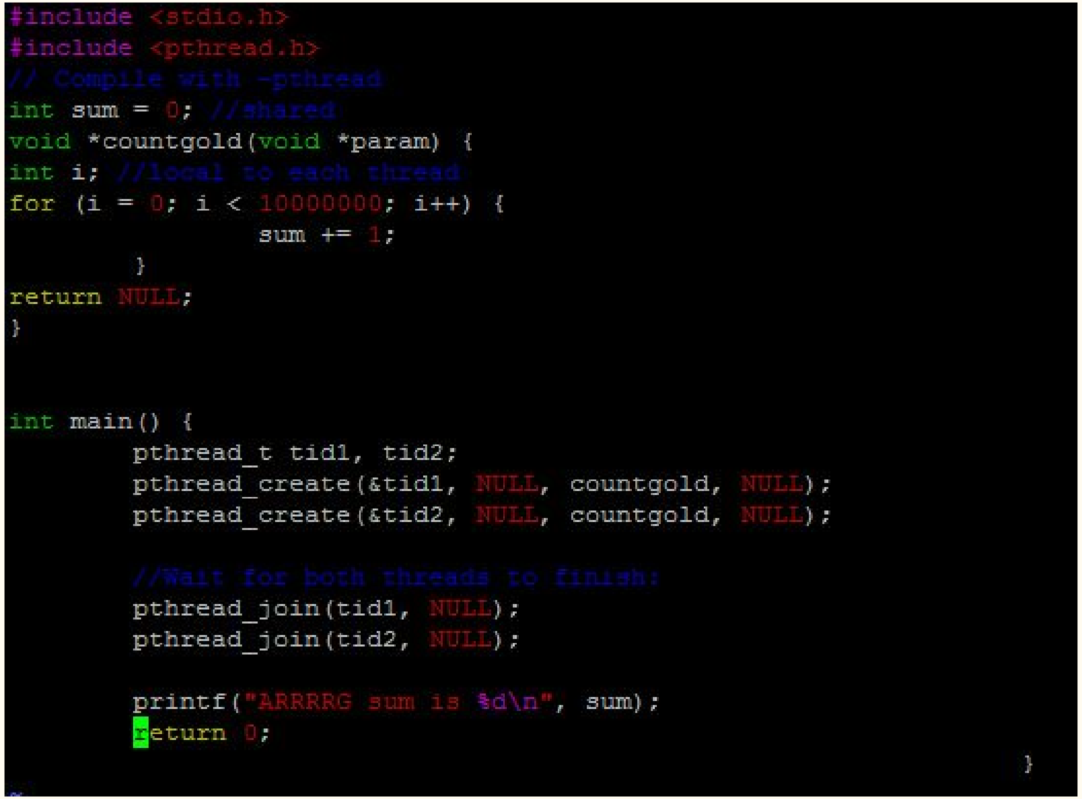


- A “race condition” arises if two or more threads access the same variables or objects concurrently and at least one does updates
- Example: Suppose t1 and t2 simulatenously execute the statement x = x + 1; for some static global x.
  - Internally, this involves loading x, adding 1, storing x
  - If t1 and t2 do this concurrently, we execute the statement twice, but x may only be incremented once
  - t1 and t2 “race” to do the update

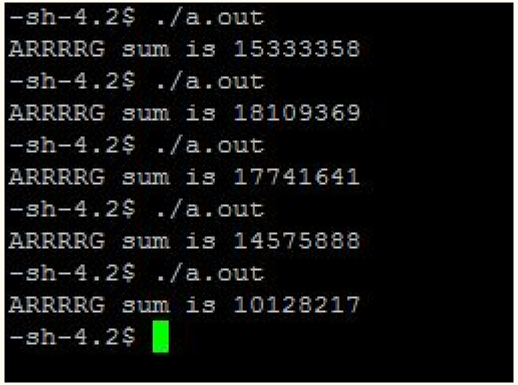
Solution: Thread Synchronization

1. **Thread synchronization** is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes.

Example of Race Condition

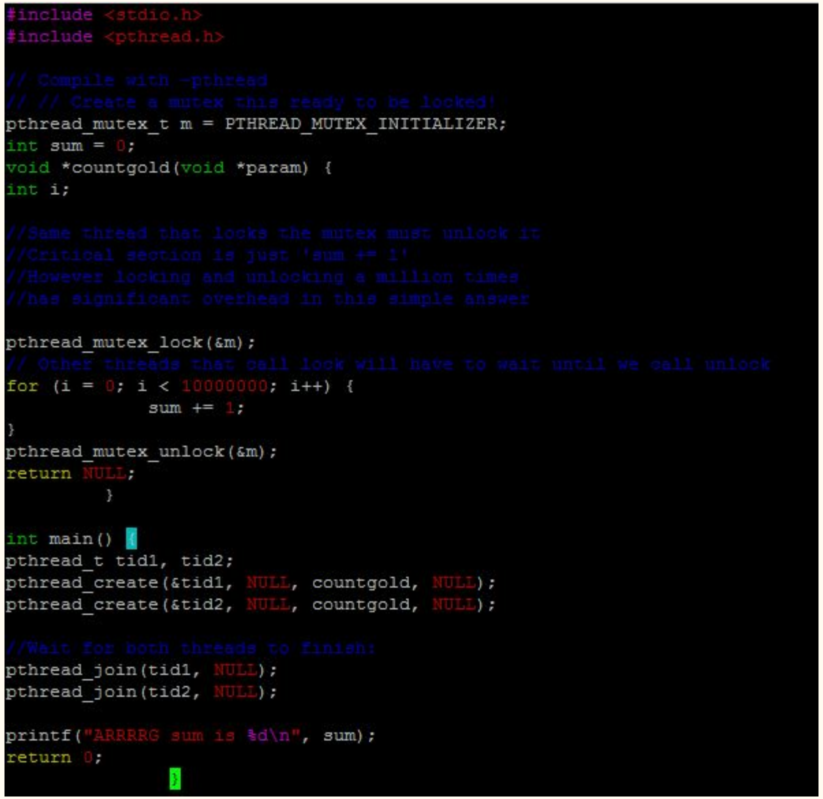


```c
#include <stdio.h>
#include <pthread.h>
// Compile with -pthread
int sum = 0; //shared
void *countgold(void *param) {
int i; //local to each thread
for (i = 0; i < 10000000; i++) {
                sum += 1;
        }
return NULL;
}


int main() {
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, countgold, NULL);
        pthread_create(&tid2, NULL, countgold, NULL);

        //Wait for both threads to finish:
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        printf("ARRRRG sum is %d\n", sum);
        return 0;
                                                                    }
```

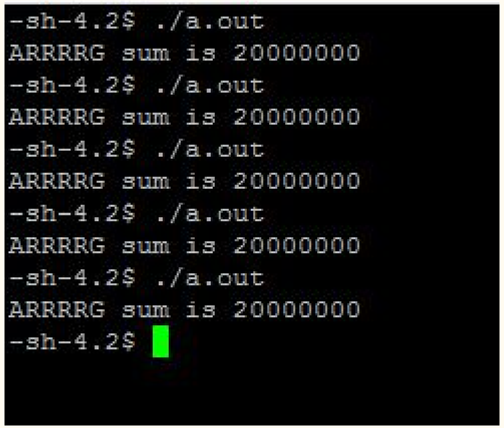


```
-sh-4.2$ ./a.out
ARRRRG sum is 15333358
-sh-4.2$ ./a.out
ARRRRG sum is 18109369
-sh-4.2$ ./a.out
ARRRRG sum is 17741641
-sh-4.2$ ./a.out
ARRRRG sum is 14575888
-sh-4.2$ ./a.out
ARRRRG sum is 10128217
-sh-4.2$
```

Mutex Lock

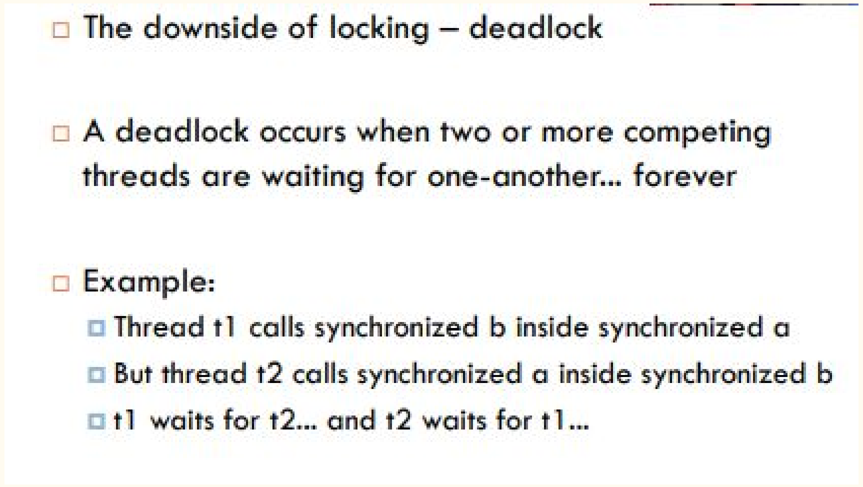


```c
#include <stdio.h>
#include <pthread.h>

// Compile with -pthread
// // Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int sum = 0;
void *countgold(void *param) {
int i;

//Same thread that locks the mutex must unlock it
//Critical section is just 'sum += 1'
//However locking and unlocking a million times
//has significant overhead in this simple answer

pthread_mutex_lock(&m);
// Other threads that call lock will have to wait until we call unlock
for (i = 0; i < 10000000; i++) {
             sum += 1;
}
pthread_mutex_unlock(&m);
return NULL;
         }

int main() {
pthread_t tid1, tid2;
pthread_create(&tid1, NULL, countgold, NULL);
pthread_create(&tid2, NULL, countgold, NULL);

//Wait for both threads to finish:
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

printf("ARRRRG sum is %d\n", sum);
return 0;
               }
```

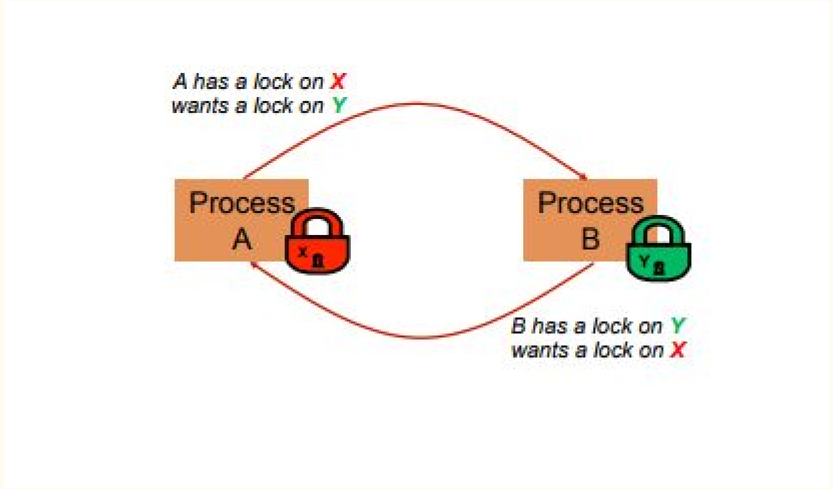


```
-sh-4.2$ ./a.out
ARRRRG sum is 20000000
-sh-4.2$ ./a.out
ARRRRG sum is 20000000
-sh-4.2$ ./a.out
ARRRRG sum is 20000000
-sh-4.2$ ./a.out
ARRRRG sum is 20000000
-sh-4.2$ ./a.out
ARRRRG sum is 20000000
-sh-4.2$
```

Deadlock



- The downside of locking – deadlock
- A deadlock occurs when two or more competing threads are waiting for one-another... forever
- Example:
  - Thread t1 calls synchronized b inside synchronized a
  - But thread t2 calls synchronized a inside synchronized b
  - t1 waits for t2... and t2 waits for t1...


A has a lock on X
wants a lock on Y
Process A
X
Process B
Y
B has a lock on Y
wants a lock on X