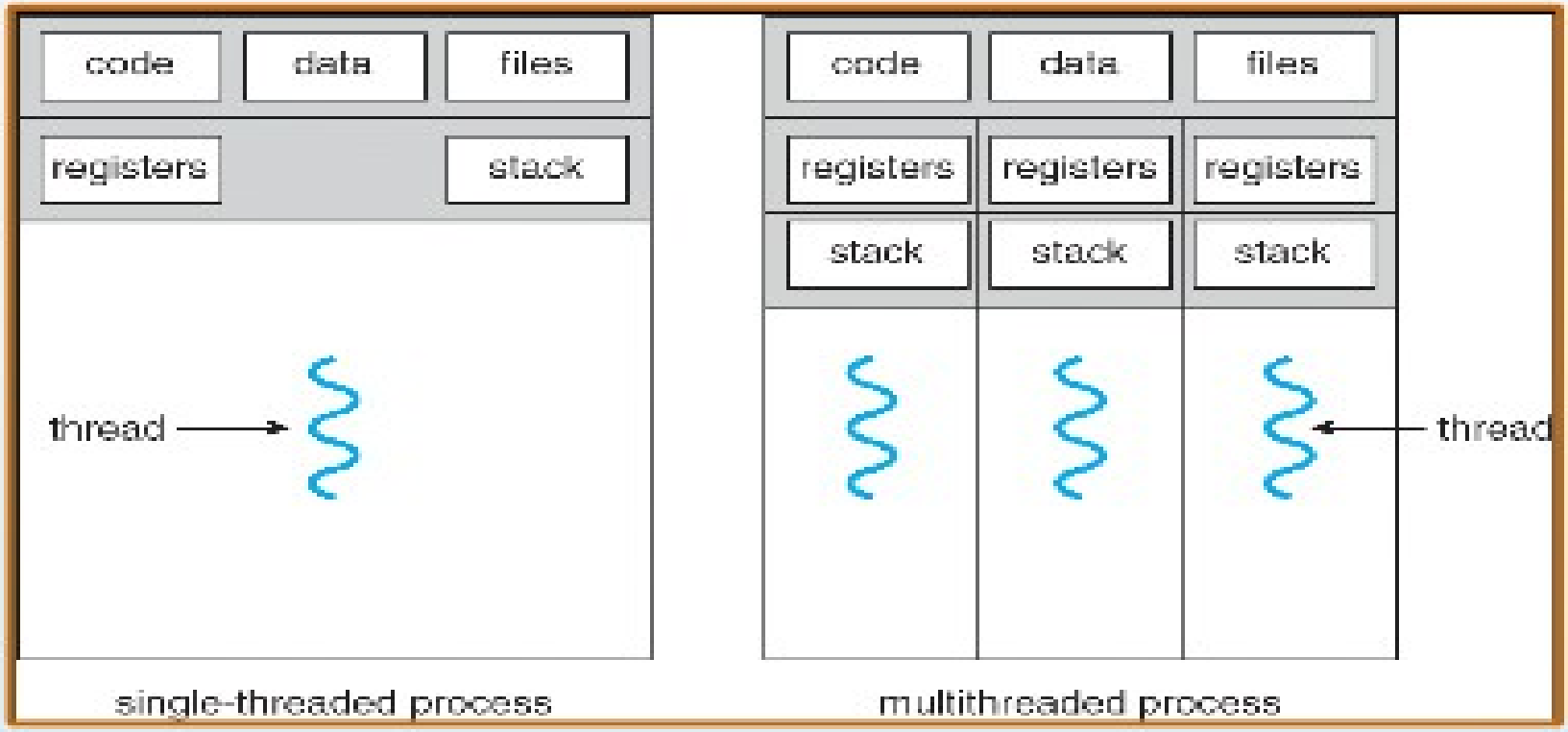
PTHREADS

BY
MALLESH MEEROLLA

What is Thread?

- ‘Light Weight Process’
- Sequence of control within a process
- Stream of instruction that can be scheduled as an independent unit.
- Exists within the process and uses or shares process resources.

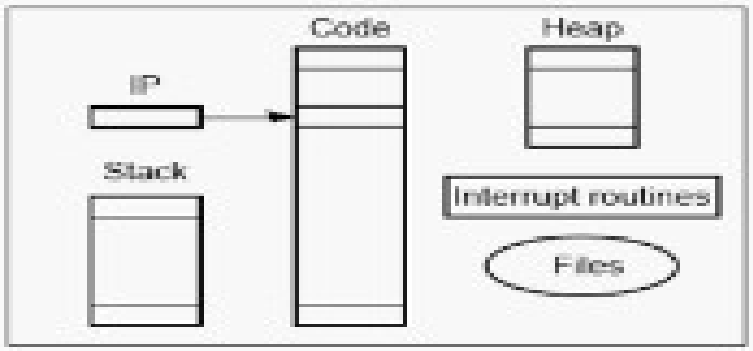
Single and Multithreaded Processes



Contd....

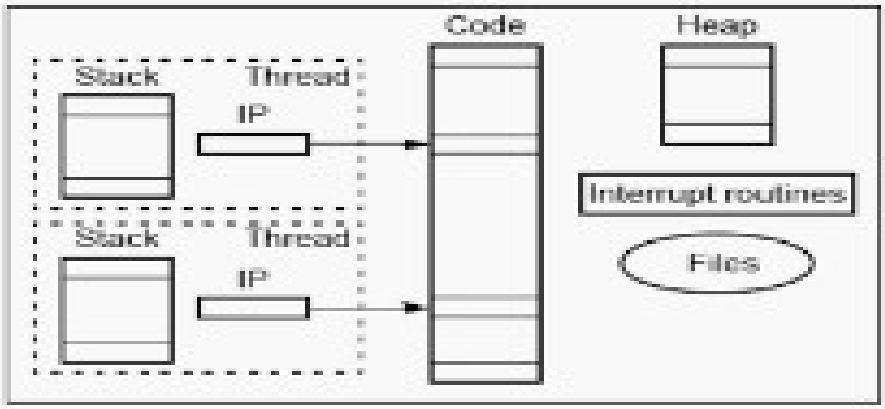
Threads vs Processes

"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.



(a) Process

Threads - shares the same memory space and global variables between routines.



(b) Threads

Process vs Threads

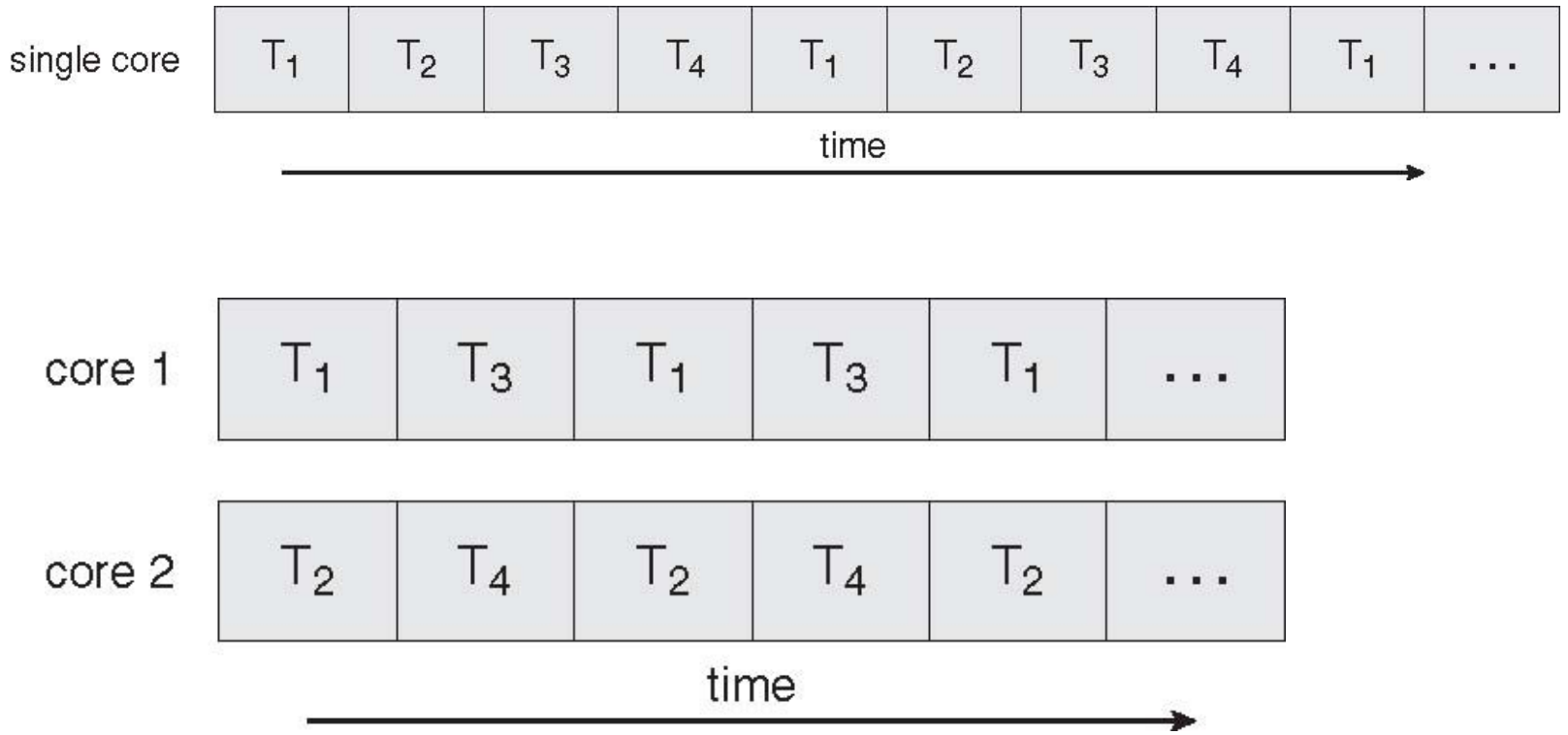
Advantages of Threads

- The overhead for creating a thread is significantly less than that for creating a process (~ 2 milliseconds for threads)
- switching between threads requires the OS to do much less work than switching between processes.
- multitasking, i.e., one process serves multiple clients.

Drawback of Threads

- Writing multithreaded programs require more careful thought
- More difficult to debug than single threaded programs
- For single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many CPU cycles to be had)

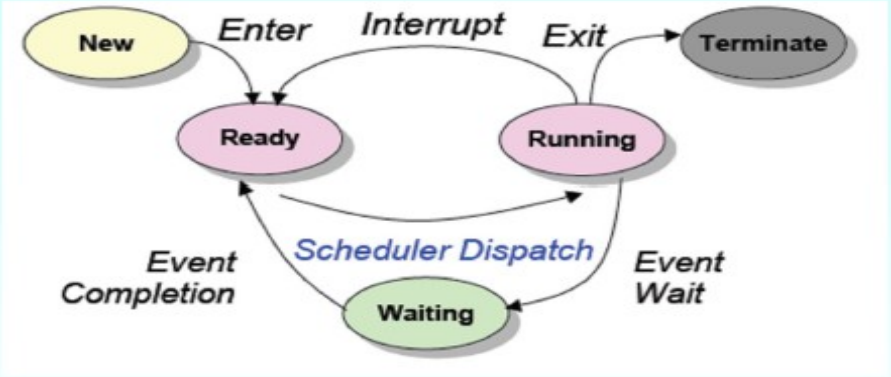
Parallel execution: Single-core V/s Multicore System



System overview of Threads……

State Diagram for a Thread

- Threads Creation
- Four Stages of Thread
  - Life Cycle
    - Ready
    - Running
    - Waiting (blocked)
    - Termination



Thread lifecycle

What is Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads

Pthreads

- POSIX threads or Pthreads is a portable threading library which provides consistent programming interface across multiple operating systems.
- It is set of C language programming types and procedure calls , implemented with pthread.h file and a thread library.
- Set of threading interfaces developed by IEEE committee in charge of specifying a portable OS Interface.
- Library that has standardized functions for using threads across different platform.

Pthread API's

Simple Pthread program using pthreads APIs:

- Include thread library
  - #include <pthread.h>
- Create a thread(s) and assign sub programs in main program
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
- Join threads (main thread wait for all threads to complete)
  - int pthread_join(pthread_t thread, void **retval);
- Exit threads
  - void pthread_exit(void *retval);

Thread Creation

- **pthread_create**
  - When a new thread is created it runs concurrently with the creating process.
  - When creating a thread you indicate which function the thread should execute.
  - **First argument:** Each thread is identified by a threadID of type pthread_t . Pointer to a pthread_t variable , in which threadID is stored.
  - **second argument:** Pointer to thread attribute object. If NULL is passed thread will be created with default thread attributes.
  - **third argument:** pointer to the thread function. Ordinary function pointer of type void* (*) (void*).
  - **Fourth argument:** Last argument value of type void*, passed as argument to the thread function.
  - Returns 0 for succes

Thread exit and Thread join and etc

- pthread_join
  - **First argument:** The ThreadID of the thread to wait for
  - **second argument:** Pointer to the void* variable that will receive the finished thread's return value.
  - Failure to join threads memory and other resource leaks until the process ends
  - Returns 0 for success
- pthread_exit
  - Thread's return value.
  - A thread can explicitly exit by using this API.
- pthread_cancel
  - One thread can request that another exit with pthread_cancel
  - int pthread_cancel(pthread_t thread);
- pthread_self
  - Returns the thread identifier for the calling thread

•Thread Attributes

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- pthread_attr_init and pthread_attr_destroy are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling parameters
  - Scheduling contention scope

•Thread Attributes

- **detach state attribute:**
  - int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
    - detached – main thread continues working without waiting for the daughter threads to terminate
    - joinable – main thread waits for the daughter threads to terminate before continuing further

•Thread Attributes

- **Contention scope attribute:**

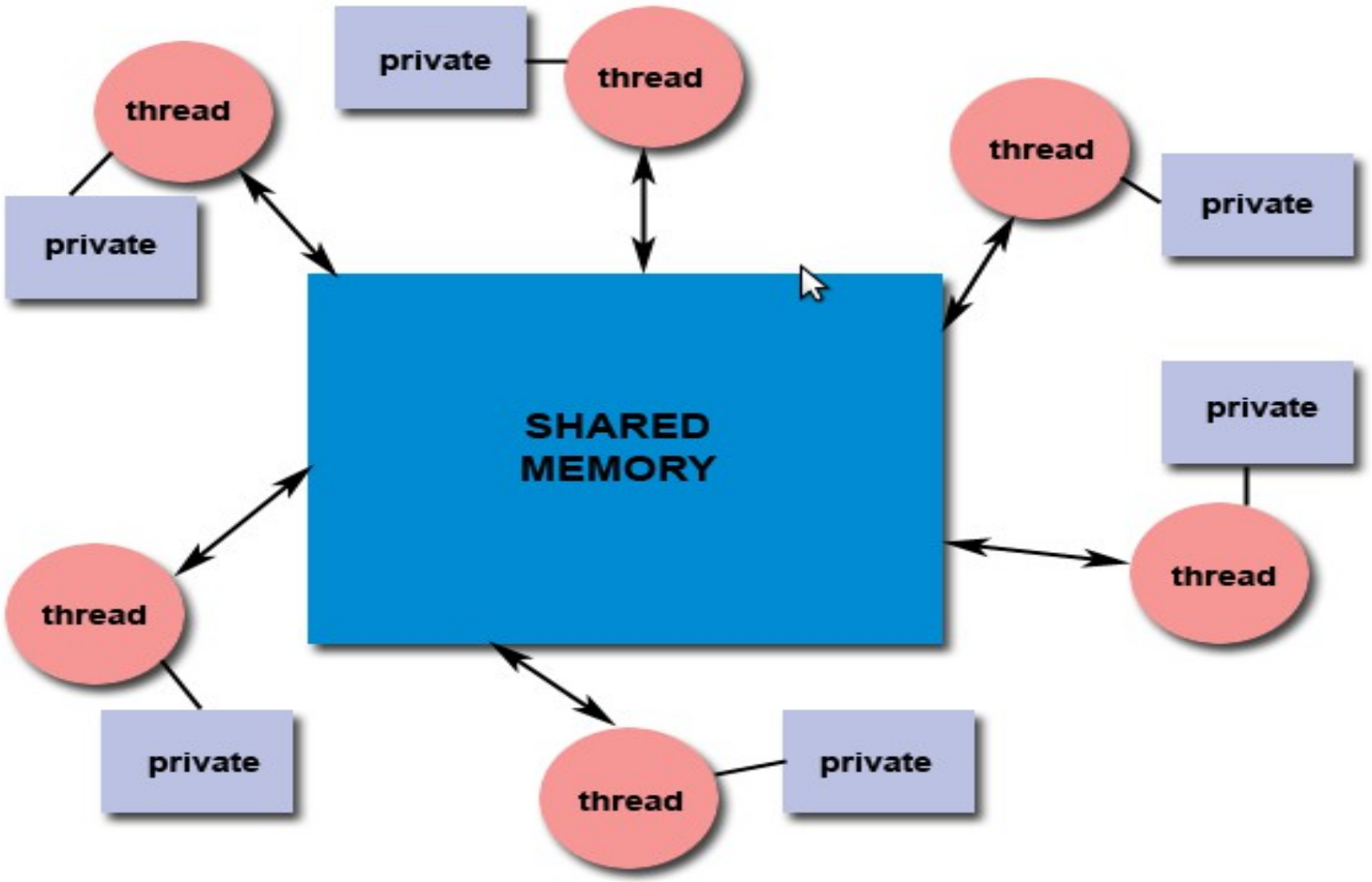
  - int pthread_attr_setscope(pthread_attr_t *attr, int *scope);

    - System scope – threads are mapped one-to-one on the OS's kernel threads (kernel threads are entities that scheduled onto processors by the OS)
    - Process scope – threads share a kernel thread with other process scoped threads

Single thread program vs multi thread program

```c
// example single threaded program

#include<stdio.h>

// function
void  printmsg(char *msg)
{
        printf("%s",msg);
        return;

}

int main()
{

printmsg("helloworld\n");   //subprogram1

printmsg("byeworld\n");     //subprogram2

return 0;

}
```

```c
//example multi threaded program

#include<pthread.h>   // to use apis provided by Pthreads library
#include<stdio.h>

// function
void  * printmsg(char *msg)
{
        printf("%s",msg);
        return;
}

int main()
{

//declare 2 child threads
pthread_t pthread1,pthread2;

/* To initializes
the thread,
it's attributes,
the address of the routine the thread has to start executing,
the parameters for that routine.
*/
pthread_create(&pthread1,NULL,printmsg,(void*)"helloworld");
pthread_create(&pthread2,NULL,printmsg,(void*)"byeworld");

//Just as pthread_create() splits our single thread into two threads, pthread_join()
merges two threads into a single thread.
pthread_join(pthread1,NULL);
pthread_join(pthread2,NULL);

// terminate thread
pthread_exit("thank u");

printf("\n");

return 0;
}
```



Race condition

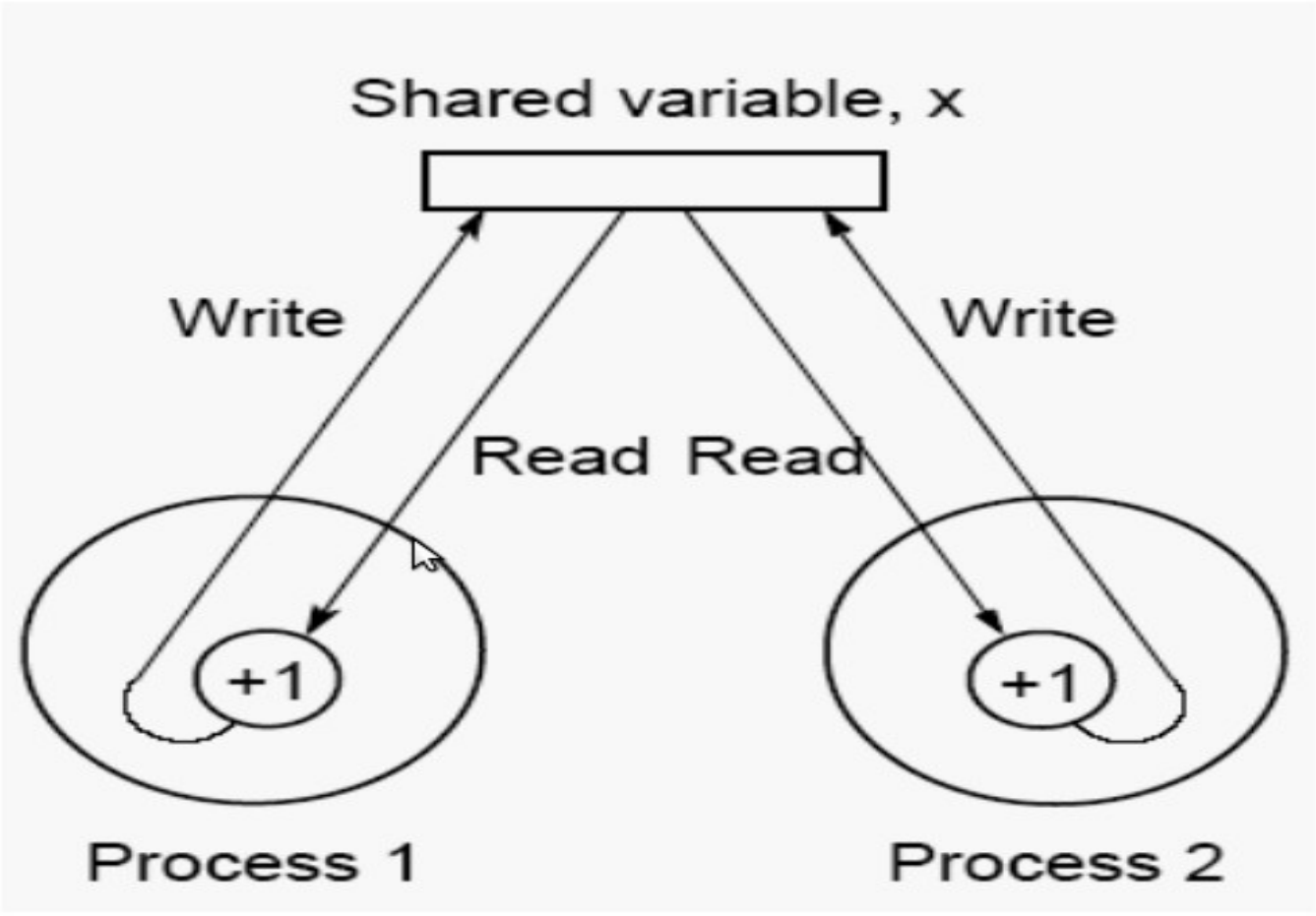
Accessing Shared Data

- The programmer must carefully control access to shared data.
- Consider two processes which both increment the variable X:

Instruction	Process 1	Process 2
x = x + 1;	read x	read x
	compute x + 1	compute x + 1
	write to x	write to x

Time

Race Condition



Race Condition

Critical Sections

- Critical sections provide a way to make sure only one process accesses a resource at a time.
- The critical section is the code that accesses the resource. They must be arranged such that only one section can be executed at a time.
- This mechanism is known as mutual exclusion.

Race Condition

Thread Synchronization Mechanisms

- **Mutual Exclusion (mutex):**
  - Guard against multiple threads modifying the same shared data simultaneously
  - Provides locking/unlocking critical code sections where shared data is modified
  - Each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

Mutex

- **pthread_mutex_init():**
  - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
  - initializes the mutex and sets its attributes
- **pthread_mutex_destroy():**
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);
  - destroy the mutex
- **pthread_mutex_lock():**
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - Locks a mutex.
  - If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- **pthread_mutex_unlock():**
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - Unlocks a Mutex.

Mutex

- **pthread_mutex_trylock():**
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - Tries to lock a Mutex. If the mutex object referenced by mutex is currently locked by any thread, the call returns immediately.

•Mutex

- A new data type named pthread_mutex_t is designated for mutexes
- A mutex is like a key (to access the code section) that is handed to only one thread at a time
- The attribute of a mutex can be controlled by using the pthread_mutex_init() function
- Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

•Counting Semaphores

- Permit a limited number of threads to execute a section of the code
- Similar to mutexes
- Should include the semaphore.h header file
- Semaphore functions do not have pthread_ prefixes; instead, they have sem_ prefixes

•Counting Semaphores

- Basic Semaphore Functions:
  - **creating a semaphore:**
  - int sem_init(sem_t *sem, int pshared, unsigned int value);
    - initializes a semaphore object pointed to by sem
    - pshared is a sharing option; a value of *0* means the semaphore is local to the calling process
    - gives an initial value value to the semaphore

  - **terminating a semaphore:**
  - int sem_destroy(sem_t *sem);
    - frees the resources allocated to the semaphore sem
    - usually called after pthread_join()
    - an error will occur if a semaphore is destroyed for which a thread is waiting

• Counting Semaphores

- **semaphore control:**
  - int sem_post(sem_t *sem);
  - int sem_wait(sem_t *sem);

  - sem_post *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

  - sem_wait *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

• Reference

- https://computing.llnl.gov/tutorials/pthreads/