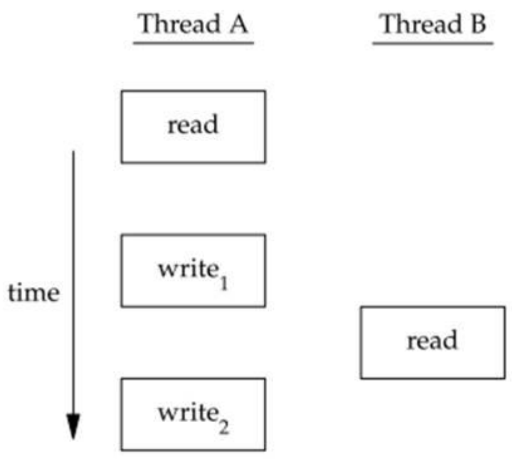
THREAD SYNCHRONIZATION

Why Synchronization

- When multiple threads of control share the same memory, we need to make sure that each thread sees a consistent view of its data.
- Consistency problem arises when one thread modifies a variable that is being shared by another thread.
- If each thread uses variables that other threads don't read or modify, no consistency problems will exist.

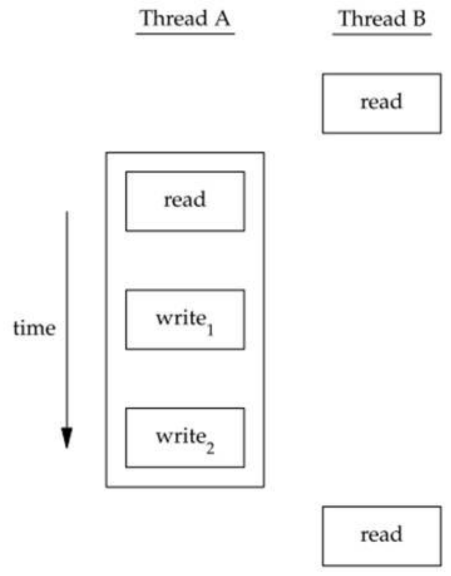
Interleaved memory cycles with two threads



Locks to the rescue

- To solve this problem, the threads have to use a lock that will allow only one thread to access the variable at a time.
- If it wants to read the variable, thread B acquires a lock.
- Similarly, when thread A updates the variable, it acquires the same lock. Thus thread B will be unable to read the variable until thread A releases the lock.

Two threads synchronizing memory access



Synchronization with Semaphores

- A **semaphore** is a variable that is used for controlling access, by multiple processes, to a common resource in a parallel programming or a multi user environment.
- There are two types of semaphore:
  - Binary Semaphore: takes value of 0 or 1.
  - Counting Semaphore: Wider range of values.

  - For synchronizing we will be using binary semaphores.

Semaphore Functions

- The semaphore functions do not start with pthread_, as most thread-specific functions do, but with sem_.
- A semaphore is created with the sem_init function, which is declared as follows:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Next Set of functions

- The next pair of functions controls the value of the semaphore and is declared as follows:

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

- These both take a pointer to the semaphore object initialized by a call to sem_init.
- The sem_post function atomically increases the value of the semaphore by 1.
- The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first.

Example

- Suppose we are creating an application in which one thread accepts input from the user and another thread counts the number of characters entered.
- Now in such a scenario, the two threads will be accessing a common resource, the character array:
  - char work_area[WORK_SIZE]
- Now since one thread is modifying this array and another thread is reading at the same

Initializing

- sem_t bin_sem; //A semaphore
- Initializing the semaphore:

```
res = sem_init(&bin_sem, 0, 0);
if (res != 0) {
perror("Semaphore initialization failed");
exit(EXIT_FAILURE);
}
```

Creating The thread

```
res = pthread_create(&a_thread, NULL, thread_function,
  NULL);
if (res != 0) {
perror(“Thread creation failed”);
exit(EXIT_FAILURE);
}
void *thread_function(void *arg) {
sem_wait(&bin_sem);
while(strncmp(“end”, work_area, 3) != 0) {
printf(“You input %d characters\n”, strlen(work_area) -1);
sem_wait(&bin_sem);
}
```

Code for the main Thread

```
printf(“Input some text. Enter ‘end’ to finish\n”);
while(strncmp(“end”, work_area, 3) != 0) {
fgets(work_area, WORK_SIZE, stdin);
sem_post(&bin_sem);
}
printf(“\nWaiting for thread to finish...\n”);
res = pthread_join(a_thread, &thread_result);
if (res != 0) {
perror(“Thread join failed”);
exit(EXIT_FAILURE);
}
printf(“Thread joined\n”);
```

Closing the semaphore

- sem_destroy(&bin_sem);
- exit(EXIT_SUCCESS);

How It Works

- When we initialize the semaphore, we set its value to 0. Thus, when the thread's function starts, the call to sem_wait blocks and waits for the semaphore to become nonzero.
- In the main thread, we wait until we have some text and then increment the semaphore with sem_post, which immediately allows the other thread to return from its sem_wait and start executing.
- Once it has counted the characters, it again calls sem_wait and is blocked until the main thread again calls sem_post to increment the

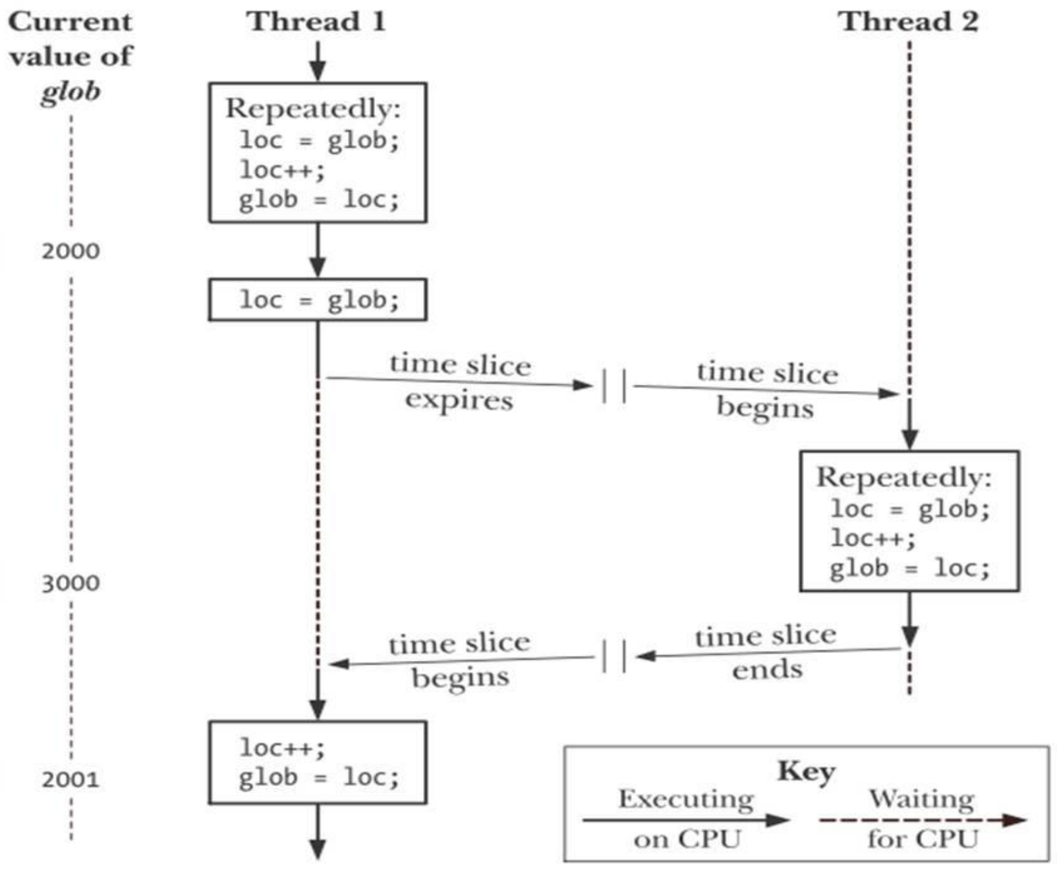
Protecting Accesses to Shared Variables: Mutexes

- Problem Statement:
  - This program creates two threads, each of which executes the same function.
  - The function executes a loop that repeatedly increments a global variable, glob, by copying glob into the local variable loc, incrementing loc, and copying loc back to glob.

The Program

```
#include <pthread.h>
static int glob = 0;
static void *                /* Loop 'arg' times incrementing
'glob' */
threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;
   for (j = 0; j < loops; j++) {
      loc = glob;
      loc++;
      glob = loc;
   }
```

```
int main(int argc, char *argv[])
{
   pthread_t t1, t2;
   int loops =1000, s;
   s = pthread_create(&t1, NULL, threadFunc, &loops);
   if (s != 0)
      printf( "pthread_create");
   s = pthread_create(&t2, NULL, threadFunc, &loops);
   if (s != 0)
      printf( "pthread_create");
   s = pthread_join(t1, NULL);
   if (s != 0)
      printf(s, "pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      printf(s, "pthread_join");
   printf("glob = %d\n", glob);
   exit(EXIT_SUCCESS);
```



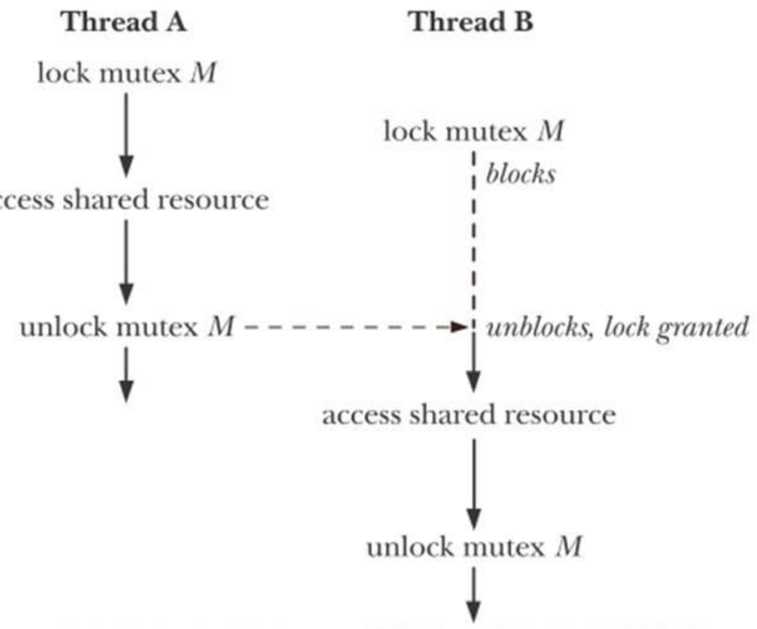
Mutexes

- We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces.
- A mutex is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done.
- While it is set, any other thread that tries to set it will block until we release it.
- If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock.

The Protocol

- Each thread employs the following protocol for accessing a resource:
  - lock the mutex for the shared resource;
  - access the shared resource; and
  - unlock the mutex.

Protecting the Critical section



Initializing the mutex

- A mutex variable is represented by the pthread_mutex_t data type.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex , const pthread_mutexattr_t
*restrict attr );
int pthread_mutex_destroy(pthread_mutex_t * mutex );
                                        Both return: 0 if OK, error number on failure
```

Locking and Unlocking a mutex

- To lock a mutex, we call pthread_mutex_lock . If the mutex is already locked, the calling thread will block until the mutex is unlocked. To unlock a mutex, we call pthread_mutex_unlock .

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex );
int pthread_mutex_trylock(pthread_mutex_t * mutex );
int pthread_mutex_unlock(pthread_mutex_t * mutex );
                                All return: 0 if OK, error number on failure
```

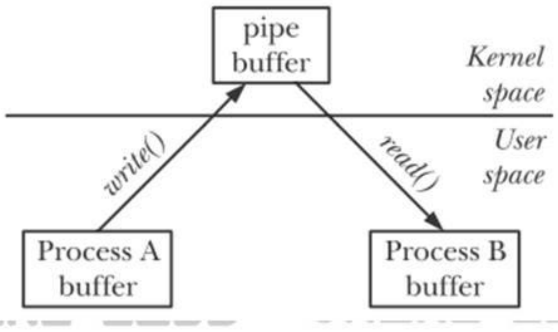
Inter-Process Communication

What is IPC

- A mechanism used by processes to communicate with each other.
- A Taxonomy of IPC Facilities:
  - Communication: These facilities are concerned with exchanging data between processes.
  - Synchronization: These facilities are concerned with synchronizing the actions of processes or threads.
  - Signals: Although signals are intended primarily for other purposes, they can be used as a synchronization technique in certain

Communication Facilities

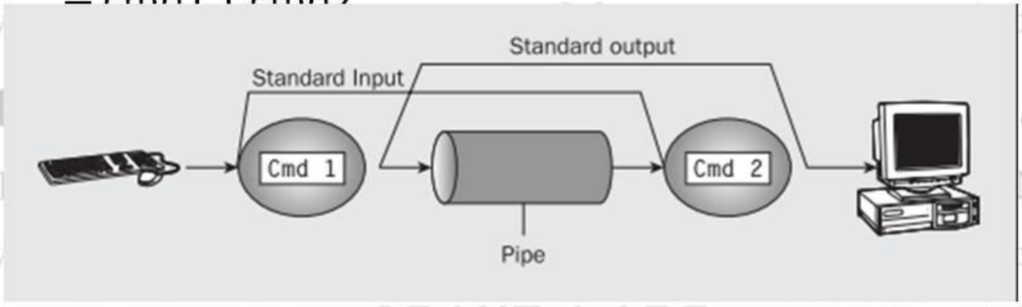
- We can break the communication facilities into two categories:
  - Data-transfer facilities: The key factor distinguishing these facilities is the notion of writing and reading



- Shared memory:
  - Shared memory allows processes to exchange information by placing it in a region of memory that is shared between the processes.
  - Because communication doesn't require system calls or data transfer between user memory and kernel memory, shared memory can provide very fast communication.

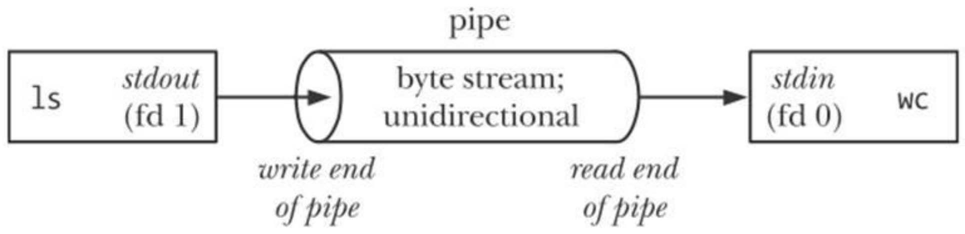
What Is a Pipe?

- We use the term pipe when we connect a data flow from one process to another. Generally we attach, or pipe, the output of one process to the input of another.
  - cmd1 | cmd2



Example

- Lets count the number of files in a directory:
  – ls | wc –l



Process Pipes

- Passing data between two programs can be done using the following functions:

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

  – The command string is the name of the program to run, together with any parameters.
  – open_modemust be either “r”or “w”.