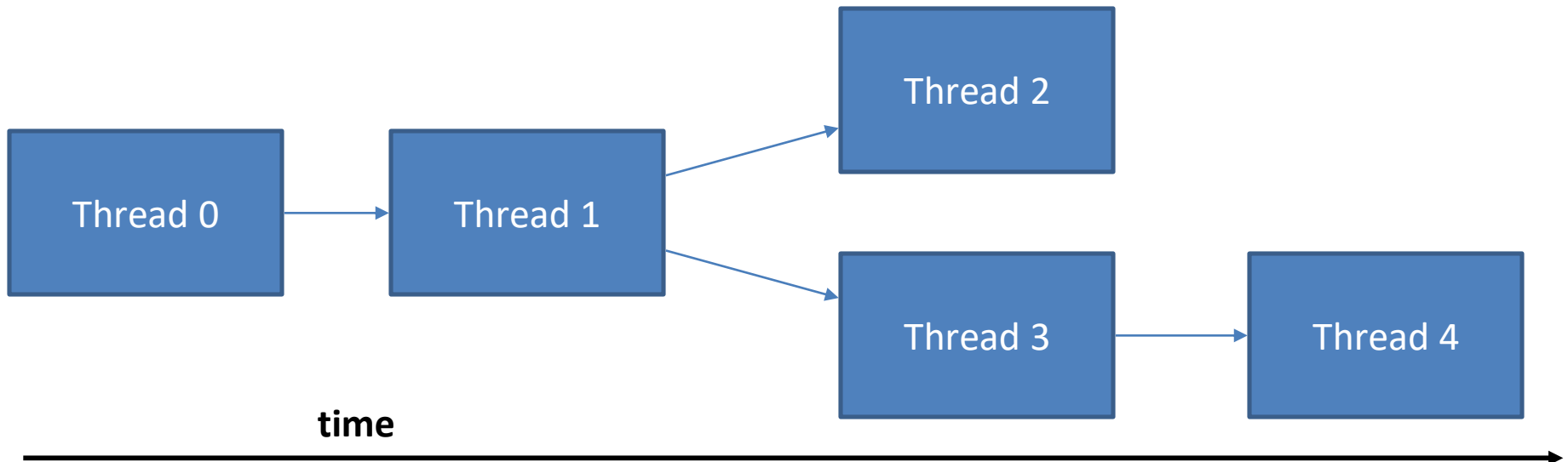


# Thread

- A thread is a sequence of related instructions executed independently of other instruction sequences.
- A thread can create another threads.
- Each thread maintains its current state of execution.
- Three types of thread
  - ✦ User level
  - ✦ Kernel level
  - ✦ Hardware level

# Thread

- Threads share same **address space** but have their own **private** stacks.
- Thread states: ready, running, waiting (blocked), or terminated.



# POSIX Thread (Pthread)

- Low-level threading libraries
- Native threading interface for Linux
- C language programming types and procedure calls implemented with a `pthread.h` header.
- It assumes shared memory.
- To compile with GNU compiler:
  - ✦ `gcc/g++ <progrname> -lpthread`
  - ✦ `gcc/g++ -pthread <progrname>`

# Pthread Management

- Routines:
  - `pthread_create (thread, attr, start_routine, arg)`
  - `pthread_join (threadid, status)`
  - `pthread_exit (status)`
  - `pthread_cancel (thread)`
  - `pthread_attr_init (attr)`
  - `pthread_attr_destroy (attr)`

# Pthread Management

- `pthread_create (thread, attr, start_routine, arg)` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- `pthread_create` arguments:
  - `thread`: An unique identifier for the new thread returned by the subroutine.
  - `attr`: can be used to set thread attributes. NULL: default, specified only at thread creation time.
  - `start_routine`: the C routine that the thread will execute once it is created.
  - `arg`: A single argument that may be passed to `start_routine`.

# Pthread Management

- `pthread_join(threadid, status)` accomplishes synchronization between threads.
- `pthread_join(threadid, status)` subroutine blocks the calling thread until the specified `threadid` thread terminates.
- The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`.

# Pthread Management

- `pthread_exit (status)` allows the programmer to specify an optional termination `status` parameter. If any file is open inside the thread, it will remain after the thread termination.
- A thread may be terminated:
  - The thread returns normally from its starting routine.
  - The thread makes a call to the `pthread_exit` subroutine.
  - The thread is canceled by another thread via the `pthread_cancel` routine.
  - The entire process is terminated due to a call to `exit()` .
  - If `main()` finishes first.

# Pthread Management


- `pthread_cancel (thread)` cancels the specified thread.
- `pthread_attr_init (thread)` allows to initialize the attribute thread.
- `pthread_attr_destroy (thread)` allows to free library resources used by the attribute.



```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    int tid;
    tid = (int) threadid;
    printf("Hello World! It's me, thread: %d!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    for(int t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
    return 0;
}
```



By having main() explicitly call 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.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
    int tid;
    tid = (int) threadid;
    printf("Hello World! It's me, thread: %d!\n", tid);
}
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    for(int t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    for (int i=0; i<NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
    return 0;
}
```



main() will block until all the threads[i] threads terminate.

# Mutual Exclusion (Mutex)

- One of the primary means of implementing **thread synchronization** and for protecting shared data when multiple writes occur.
- A mutex acts like a **lock** protecting access to a shared data resource.
- Only one thread can lock (or own) a mutex variable at any given time.

# Mutual Exclusion (Mutex)

- Routines:
  - `pthread_mutex_init (mutex, attr)`
  - `pthread_mutex_destroy (mutex)`
  - `pthread_mutex_lock (mutex)`
  - `pthread_mutex_unlock (mutex)`

# Mutual Exclusion (Mutex)

- `pthread_mutex_init (mutex, attr)` initiates the `mutex` and permits setting mutex object attributes `attr`.
- `pthread_mutex_destroy (mutex)` should be used to free a `mutex` object which is no longer needed.

# Mutual Exclusion (Mutex)

- `pthread_mutex_lock (mutex)` routine is used by a thread to acquire a lock on the specified `mutex` variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_unlock (mutex)` will unlock a mutex if called by the owning thread. An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
int sum_value;
void *doSum (void *arg) {
    int sum = (int) arg;
    while(true){
        pthread_mutex_lock(&mutex);
        sum_value += sum;
        printf("After sum in thread %d = %d\n", sum, sum_value);
        pthread_mutex_unlock (&mutex);
        Sleep(5);}
}
int main (int argc, char *argv[]) {
    pthread_t thread1, thread2;
    int t1 = 1, t2 = 2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&thread1, NULL, doSum, (void *)t1);
    pthread_create(&thread2, NULL, doSum, (void *)t2);
    pthread_join (thread1, NULL);
    pthread_join (thread2, NULL);
    return 0;
}
```

# 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 have `sem_` prefixes



# Semaphores

- Routines:
  - `sem_init (sem, pshared, value)`
  - `sem_destroy (sem)`
  - `sem_wait (sem)`
  - `sem_post (sem)`

# Semaphores

- `sem_init (sem, pshared, 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
- `sem_destroy (sem)` frees the resources allocated to the semaphore `sem` and this routine is usually called after `pthread_join()`.

# Semaphores

- `sem_wait (sem)` atomically decreases the value of a semaphore `sem` by 1, if it is negative, the calling process blocks
  - one of the blocked processes wakes up when another process calls `sem_post`.
- `sem_post (sem)` atomically increases the value of a semaphore `sem` by 1.