Lecture Notes for Lecture 11 of CS 5600 (Computer Systems) for the Fall 2019 session at the Northeastern University Silicon Valley Campus.

Threads and Concurrency

Philip Gust,
Clinical Instructor
Department of Computer Science

Acknowledgements:
These slide highlight content from suggested readings.

Lecture 10 Review

- In this lecture, we explored the low-level C language *unbuffered I/O* functions that enables writing and reading information directly to and from files. Each call function is a call to the operating system.
- We also learned about a higher-level access mechanism known as *buffered I/O* that buffers information before writing or reading it to or from the file system to improve I/O efficiency.
- Next we looked at block-oriented methods of writing and reading information to and from the disk as blocks rather than streams, which enables dealing with structs and arrays.
- Finally, we will saw how to access information in specific parts of a file using random-access I/O functions.

Threads and Concurrency

- Up until now, programs we have written performed just a single task at a time and exit when the task was complete.
- Today's lecture will present a model of computing where a program can perform multiple tasks at the same time, and coordinate the actions of these tasks.
- Tasks can begin and end during the execution of a program as required, and a task that completes does not result in the program exiting.
- The mechanism we will study that accomplishes this is known as a *thread*, and each task is called a *thread of execution*. A program that operates in this way is called multi-threaded.
- In this lecture we will study how threads work, how they can be used, and how to implement them in C and C++.

Threads and Concurrency

Processes

- When a program is run, the operating system creates a process that executes the program.
- A process is like a virtual computer with its own address space, executable code, open file handles, environment variables, and other resources.
- Each process is started with a single thread of execution, often called the *main thread*. and can create additional threads from any of its threads.

Threads and Concurrency

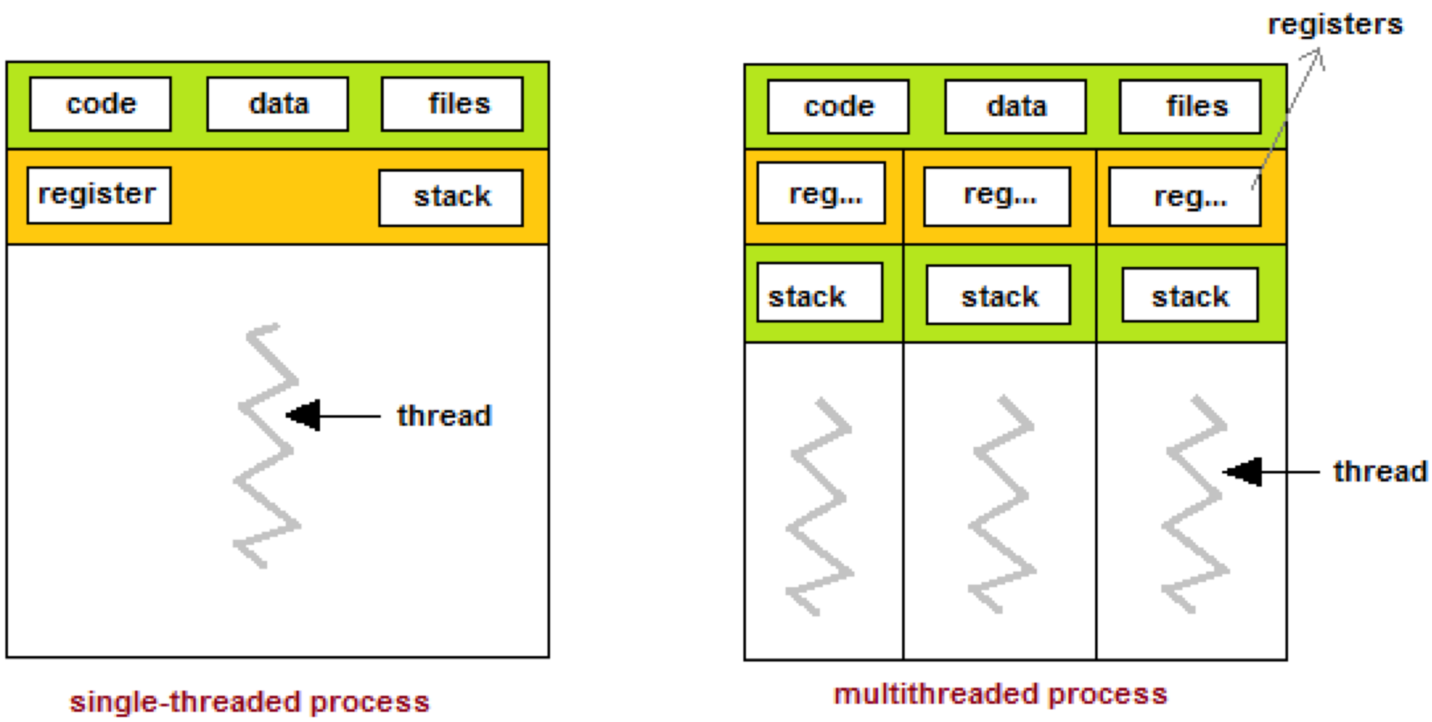
Processes

- All threads of a process share its virtual address space and system resources.
- Each thread maintains exception handlers, a scheduling priority, thread local storage, a thread ID, and structures the system uses to save the thread context until it is scheduled.
- The thread context includes the thread's machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.

Threads and Concurrency

Processes

- Here is a diagram of single- and multi-threaded processes.



Threads and Concurrency

Types of Threads

- There are two types of threads :
  - *Kernel threads* are implementation-dependent kernel resources that are entirely managed by the operating system scheduler.
  - *User threads* have a standard interface, operate entirely within a process, and are used to handle multiple flows of control within a running program.

Threads and Concurrency

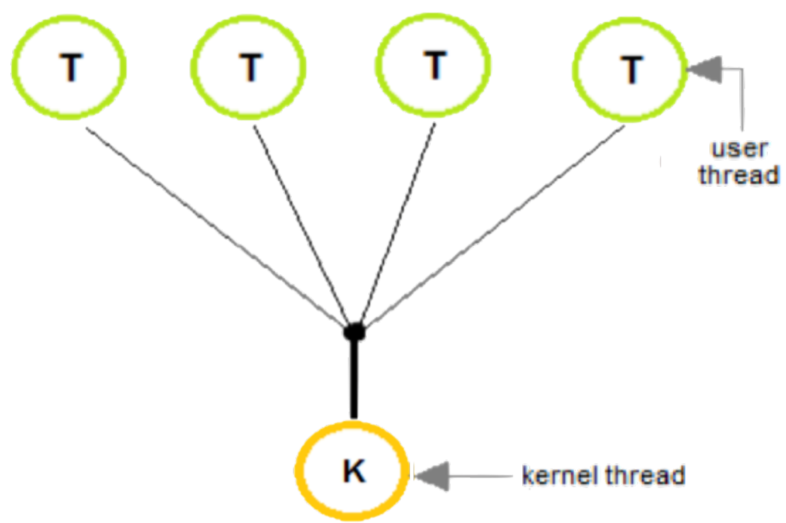
Types of Threads

- When a process needs to run a thread, it can manage the thread of execution itself, or “borrow” a kernel thread to run the process thread.
- Processes can have many threads, but kernel threads are limited, so user threads share kernel threads using one of the following strategies:
  - Many-to-One Model
  - One-to-One Model
  - Many-to-Many Model

Threads and Concurrency

Types of Threads

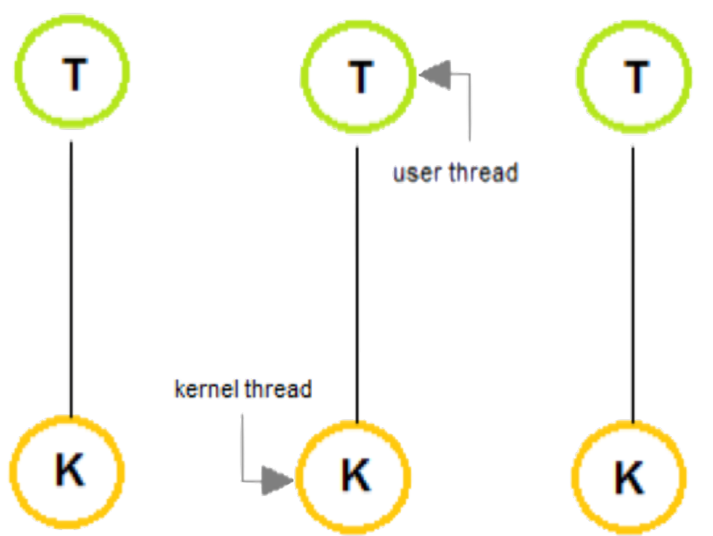
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by a thread library entirely in user space, which is efficient in nature because no operating system calls are required.



Threads and Concurrency

Types of Threads

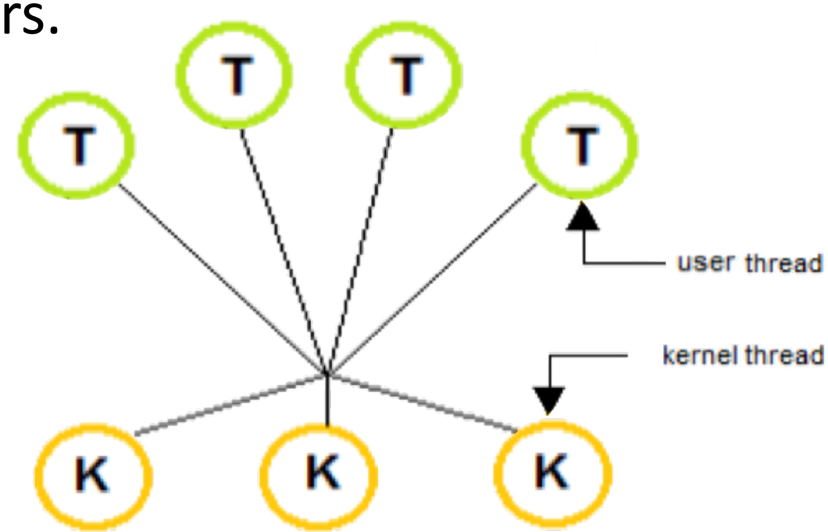
- A one-to-one model creates separate kernel thread for each and every user thread.
- Linux, and Windows from 95 to XP, implement this model. Most implementations limit how many threads can be created.



Threads and Concurrency

Types of Threads

- Multiplexes user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of threads. Kernel system calls do not block the entire process. Processes can be split across multiple processors.



Threads and Concurrency

Preemptive vs. Cooperative Threads

- Cooperative threads are also called non-preemptive threads. Cooperative threads have exclusive use of the CPU until they give it up. The scheduler then picks another thread to run.
- Preemptive threads can voluntarily give up the CPU but when they do not, the scheduler can take control from them and start another thread.
- Cooperative threads have greater efficiency (on single-core machines, at least) and easier handling of concurrency: it only exists when you yield control, so locking isn't required.
- Preemptive threads are more fault tolerance: failing to yield does not block other threads. They also work better on multi-core machines, and, you don't have to constantly yield.

Threads and Concurrency

Managed vs. Daemon Threads

- A thread is said to be *managed* if the state of the thread can be directly controlled either by the thread that created it or from any other thread in the process.
- Managed threads run ”in the foreground” and perform tasks in coordination with other threads in the process.
- A thread is said to be a *daemon* if the thread is *detached* or runs independently of other threads, and is not controlled by any other thread in the process.
- Daemon threads run “in the background” and once started, perform tasks that require no further external control.

Threads and Concurrency

Benefits of Multi-Threading

- Benefits of multi-threaded programs include:
  - More responsiveness because users can interact with a the program while processing also takes place
  - Better resource utilization because different parts of a program can utilize different resources concurrently.
  - Allows different threads of a program to be distributed over multiple processors.
  - Context switching is smooth because switching between threads in a process is more efficient than switching between processes.

Threads and Concurrency

Examples of Threaded Systems

- Threads play a critical role in a number of both large and small scale systems, including:
  - Producer/consumer and queueing systems
  - Data-centric systems like databases
  - Web browsers and web servers
  - Distributed and peer-to-peer systems
  - Cell phone operating systems like iOS & Android

Threads and Concurrency

User Thread Libraries

- Thread libraries provides programmers with API for creating and managing of threads.
- May be implemented either in user space or in kernel space.
  - The user space involves API functions implemented solely within user space, with no kernel support.
  - The kernel space involves system calls, and requires a kernel with thread library support.

Threads and Concurrency

User Thread Libraries

- We will study POSIX *pthreads* available on most Unix-based systems.
- Pthreads defines a set of C programming language types, functions and constants. It is implemented with a pthread.h header and a thread library.

Threads and Concurrency

User Thread Libraries

- There are around 100 pthread procedures, all prefixed by pthread_. They are categorized into four groups:
  - Thread management - creating, joining threads etc.
  - Mutexes
  - Condition variables
  - Synchronization between threads using read/write locks and barriers
- The POSIX semaphore API works with POSIX threads but is part of the POSIX.1b Real-time extensions (IEEE Std 1003.1b-1993) standard. Semaphore procedures are prefixed by sem_.

Threads and Concurrency

Example: C Posix Thread Library

```
/** data shared by the thread */
int sum;

/**
 * This function reads a non-negative value parameter
 * and runs a thread that sets global sum to the sum of
 * numbers between 1 and that value.
 */
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, ”main(): usage: %s max_value\n", argv[0]);
        return EXIT_FAILURE;
    }
    int maxval = atoi(argv[1]);
    if (maxval < 0) {
        fprintf(stderr, ”main(): %d must be >= 0\n", maxval);
        return EXIT_FAILURE;
    }
```

Threads and Concurrency

Example: C Posix Thread Library

```
// initialize thread attributes
pthread_attr_t attr;    // set of thread attributes
pthread_attr_init(&attr);

// create the thread
pthread_t tid;
printf("main(): running thread to compute sum from 1 to %d\n", maxval);
int status = pthread_create(&tid, &attr, runner, &maxval);
if (status != 0) {
      fprintf(stderr, "thread not created: error=%d", status);
      return EXIT_FAILURE;
}
printf("main(): created thread %lu: waiting for it to exit\n", (unsigned long)tid);
```

Threads and Concurrency

Example: C Posix Thread Library

```
    // wait for thread to exit
    void *sumptr;
    status = pthread_join(tid, &sumptr);  // result also returned here
    if (status != 0) {
        fprintf(stderr, "thread exited with status %d\n", status);
        return EXIT_FAILURE;
    }

    // report sum
    printf("main(): thread %lu exited normally\n", (unsigned long)tid);
    printf(”main(): global sum = %d\n", sum);
    if (sumptr != NULL) {
        printf(”main(): thread returned sum = %d\n", *(int*)sumptr);
    }
    return EXIT_SUCCESS;
}
```

Threads and Concurrency

Example: C Posix Thread Library

```
/**
 * Thread begins control in this function, which computes the sum
 * of the integers between 1 and the value of the input parameter.
 * @param param the integer pointed to by param
 * @return pointer to result value (unused)
 */
void *runner(void *param) {
	sum = 0;  // initialize global

	int upper = *(int*)param;
	for (int i = 1; i < upper; i++) {
		sum += i;
	}

	pthread_t tid = pthread_self();
	printf("runner() [thread %lu]: sum is %d\n", (unsigned long)tid, sum);

	// return pointer to sum that pthread_join can retrieve
	pthread_exit(&sum);  // or return pointer to sum
}
```

Threads and Concurrency

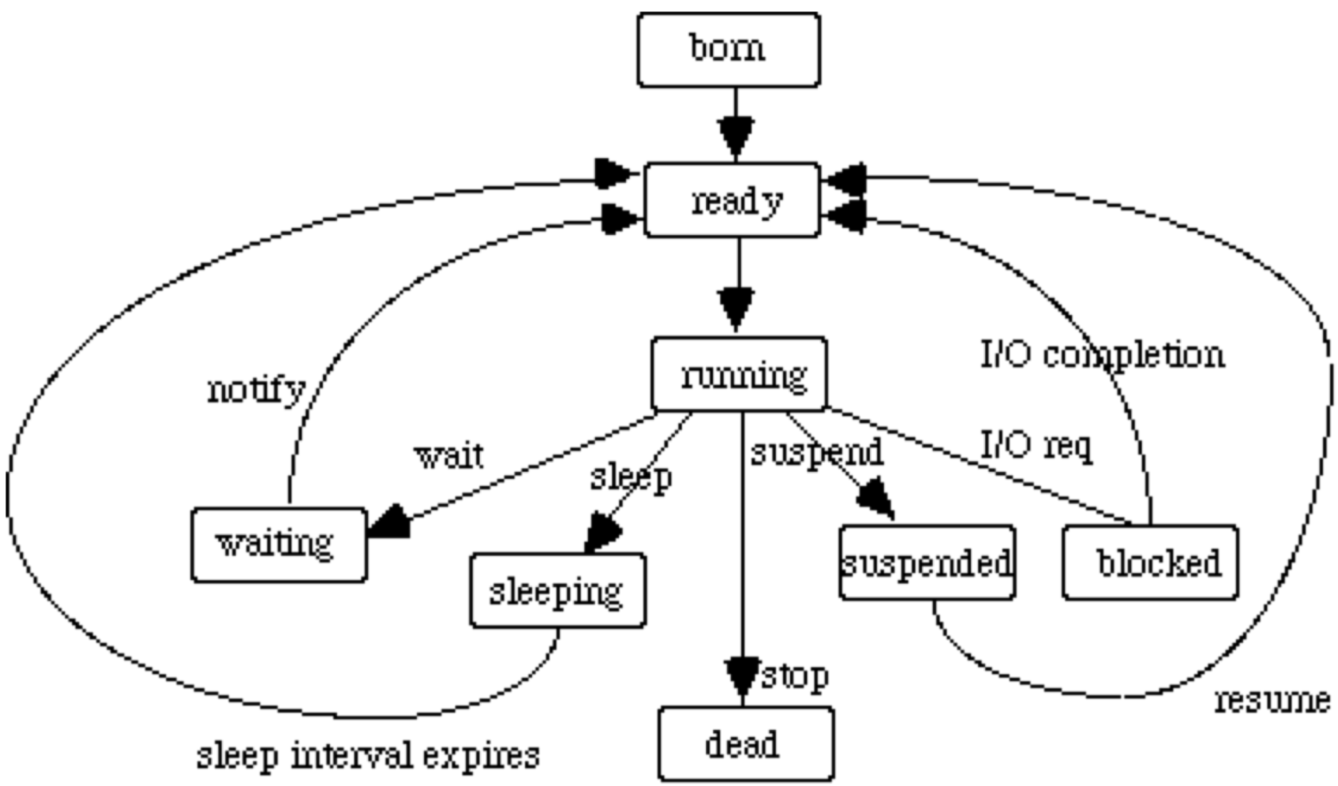
Lifecycle of a Thread

- Threads go through a number of transitions from one state to another as they are created, initialized, run, complete, and are destroyed.
- As threads to from one state to another they follow a regular set of paths that are know as *state transitions*.
- The set of states and the state transitions they follow can be represented in a graph know as a *state transition diagram*.

Threads and Concurrency

Lifecycle of a Thread

- Here is a state transition diagram for a thread's lifecycle.



Threads and Concurrency

Lifecycle of a Thread

- A thread runs until it is swapped out for another thread (going back to the ready state) or until one of the following happens:
  - If it runs to completion or a "stop" message is sent to it, then it moves to a "dead" state.
  - If it issues an I/O request, it moves to a blocked state until the I/O request is completed, and then moves to the "ready" state again.
  - If a "sleep(n)" is called on it, then it waits for at least n milliseconds, and then returns to the ready state. Used when want fixed delay time.

Threads and Concurrency

Lifecycle of a Thread

- If a "suspend" message is sent to it, then it stays in the "suspended" state until a resume message is sent to it.
- If a wait message is sent to it, then it goes into the "waiting" state where it waits in a queue.
- The first item in the wait queue goes into the "ready" state on a call to notify by another thread associated with the object.
- A call of notifyAll puts all of the threads on the wait queue in the "ready" state. Used when don't know who is waiting to proceed.

Threads and Concurrency

Lifecycle of a Thread

- Most thread systems allow a program to ask a thread if it is still alive and whether it can be joined.
- It is hard to suspend a thread if it is busy.
  - As a result, often have a method that sets a flag that can then be checked each time through the loop to see if want to suspend.
  - The thread can then suspend itself, though it will be "resumed" from outside.

Threads and Concurrency

Concurrency and Synchronization

- So far, we have seen a simple case of multi-threading where a main thread creates and runs a second thread while the main thread waits for it to complete.
- In this case, there is little need for the two threads to coordinate their execution, or guard against one thread writing data that interferes with the other one using it.
- In real-world multi-threaded programs threads must gain exclusive access to data at times, while cooperating to produce data for other threads to read.
- We will next consider the problem of concurrency and synchronization, and the tools for implementing them.

Threads and Concurrency

Concurrency and Synchronization

- *Synchronization* refers to one of two distinct but related concepts: synchronization of tasks and of data.
    - *Task synchronization* refers to multiple tasks coordinating based on some shared state, to ensure that the state of one task matches with the state of another one.
    - *Resource synchronization* refers to multiple tasks coordinating on some shared resource to ensure orderly and consistent access to the resource.

Threads and Concurrency

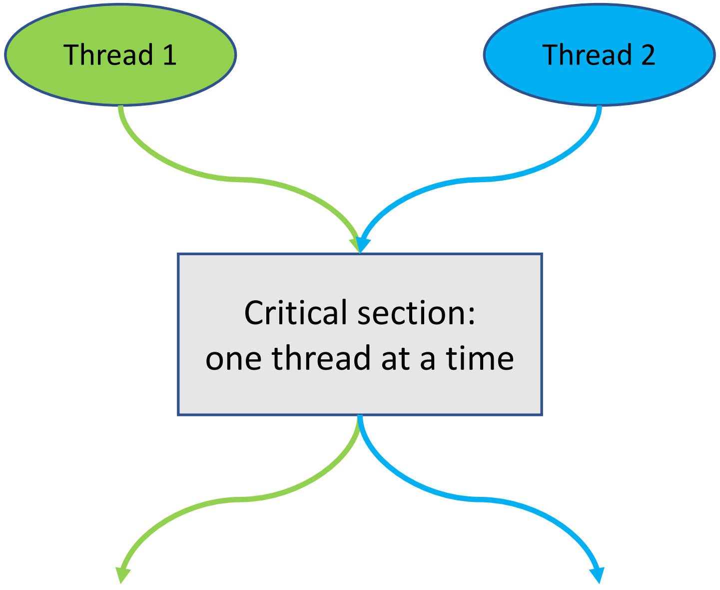
Concurrency and Synchronization

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

Threads and Concurrency

Concurrency and Synchronization

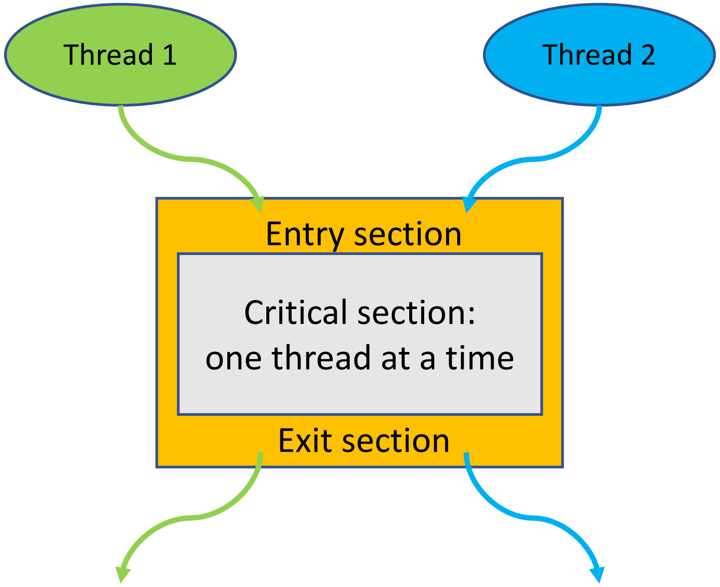
- Task synchronization: multiple threads should not be in a critical section of code at the same time.



Threads and Concurrency

Concurrency and Synchronization

- Task synchronization: multiple threads should not be in a critical section of code at the same time.



Threads and Concurrency

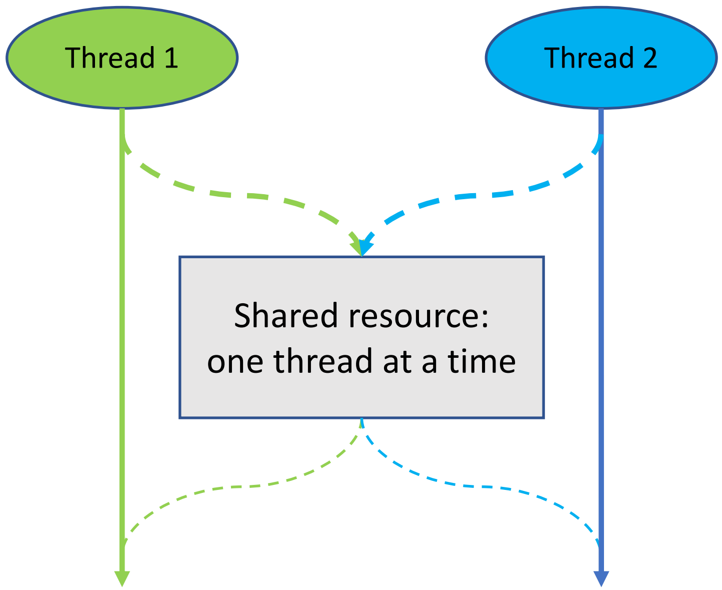
Concurrency and Synchronization

- Resource synchronization is a mechanism that ensures two or more concurrent threads do not simultaneously access a shared resource at the same time
- When one thread gains access to a shared resource, the other thread should wait until the first thread finishes before accessing it.
- A thread’s access to a shared resource is controlled by using synchronization techniques.

Threads and Concurrency

Concurrency and Synchronization

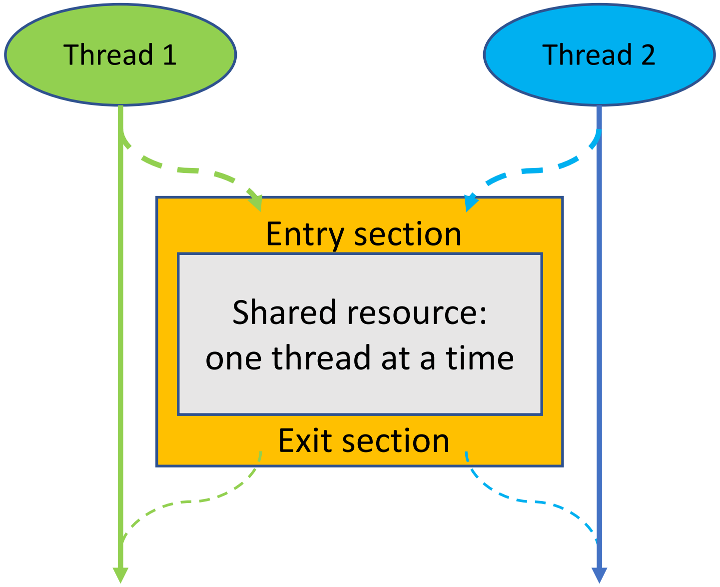
- Resource synchronization: multiple threads should not access a critical resource at the same time.



Threads and Concurrency

Concurrency and Synchronization

- Resource synchronization: multiple threads should not access a critical resource at the same time.



Threads and Concurrency

Synchronization Techniques

- Semaphores (1960s)
  - Introduced by Dutch computer scientist Edsger Dijkstra
  - Mutual exclusion primitives of early operating systems.
- Monitors (1970s)
  - Proposed independently by British computer scientist Tony Hoare and by Danish-American computer scientist Per Brinch Hansen.
  - Provides both mutual exclusion and the ability to wait (block) for a certain condition to become true.

Threads and Concurrency

Synchronization Techniques: Semaphore

- A semaphore provides mutual exclusion for executing a block of code or for locking access to a shared resource.
- A semaphore is implemented as a non-negative integer variable with two atomic and isolated operations: Passieren()/down() and Vrijgeven()/up()
  - P()/down() if semaphore value > 0, decrements semaphore and proceeds; otherwise waits until it is possible to decrement.
  - V()/up() increments semaphore; if value > 0 another waiter on P() can proceed to lock semaphore.

Threads and Concurrency

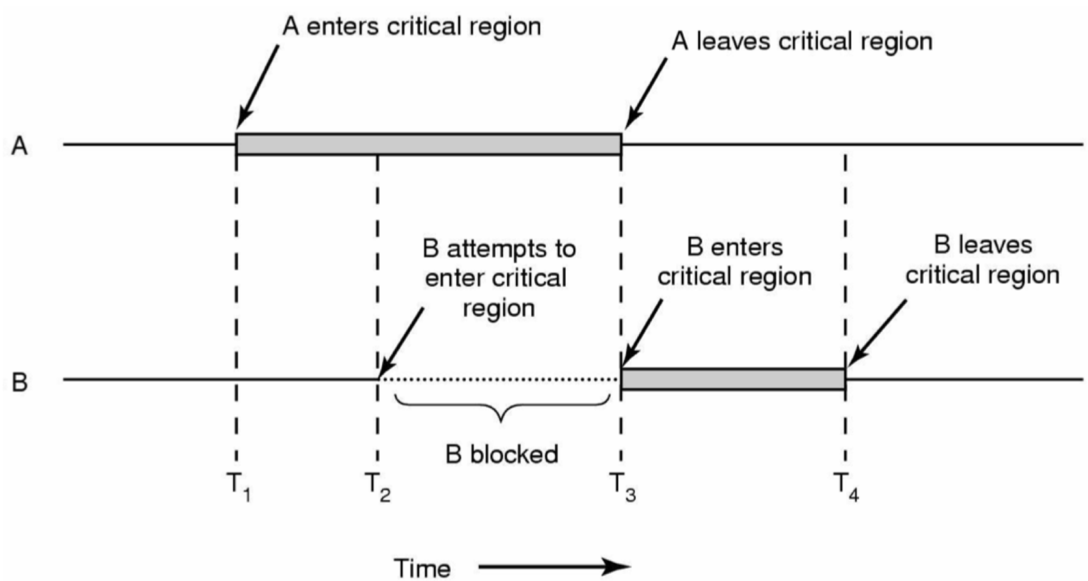
Synchronization Techniques: Semaphore

- Two types of semaphores
  - Binary semaphores: Can either be 0 or 1
  - General/Counting semaphores: any non-negative value.
  - Binary semaphores are as expressive as general semaphores (given one can implement the other)

Threads and Concurrency

Synchronization Techniques: Semaphore

- Semaphores are used both for mutual exclusion, and conditional synchronization



Mutual exclusion using critical regions

Threads and Concurrency

Synchronization Techniques: Semaphore in C

- C pthreads package includes a semaphore primitive that acts as locking mechanism for critical sections.
- **sem_t** is the type descriptor for a semaphore. It is a small integer, similar to a file descriptor for files.
- There are two kinds of semaphores:
  - Unnamed semaphores can be used only by threads belonging to the same process or by processes that share memory
  - Named semaphores has a actual name in the file system and can be shared by multiple unrelated processes.

Note: MacOS has replaced Posix unnamed semaphores with a proprietary mechanism. However, it is possible to substantially implement unnamed semaphores using named semaphores, which they still support. “semaphore.h” does this for MacOS

Threads and Concurrency

Synchronization Techniques: Semaphore in C

- Here is an example of using a semaphore to implement a producer and consumer. A *producer* and a *consumer* share a resource and must coordinate access to it.
- The consumer must wait for the producer to produce units of a resource before consuming. The producer must wait for the consumer to consume a resource before producing more.
- A semaphore is used to synchronize access to the shared resource to ensure orderly production and consumption.
- In this example, the shared resources is a message string that is written by the producer and read by the consumer.

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include "semaphore.h”  // see note

/** mutual exclusion semaphore */
sem_t semaphore;

/** shared string resource */
char message[128] = "";  // no message
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
/** Creates consumer thread, runs producer, and reports results. */
int main(void) {
    pthread_t consumer_tid;  // thread for consumer

    // initialize semaphore
    If (sem_init(&semaphore, 0, 1) != 0) {
        perror("sem_init");
        return EXIT_FAILURE;
    }
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
    // create and start consumer thread
    sem_wait(&semaphore);  // lock critical section
        fprintf(stderr, "main: creating consumer\n");
        pthread_create(&consumer_tid, NULL, consumer, NULL);
        sleep(2);  // to slow things down
    sem_post(&semaphore);  // unlock critical section

    //produce messages
    const char* messages[] = {"first", "second", "third", "quit"};
    producer(4, messages);

    // wait for consumer and report count
    char *msgcount;
    pthread_join(consumer_tid, &msgcount);
    fprintf(stderr, "main: finished with msgcount %u\n", (unsigned)(msgcount – (char*)0));
}
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
/** This function produces message strings. */
void producer(int nmessages, const char *messages[]) {
    unsigned msgindex = 0;
    fprintf(stderr, "producer: beginning\n");

    while (msgindex < nmessages) {
        fprintf(stderr, "producer: waiting to lock semaphore\n");
        sem_wait(&semaphore);  // lock critical section
            fprintf(stderr, "producer: locked semaphore\n");
            if (strlen(message) == 0) {  // message consumed
                fprintf(stderr, "producer: producing message '%s'\n", messages[msgindex]);
                strcpy(message, messages[msgindex++]);
            } else {  // message not yet consumed
                fprintf(stderr, "producer: waiting to produce next message\n");
            }
            fprintf(stderr, "producer: unlocking semaphore\n");
        sem_post(&semaphore);  // unlock critical section
        sleep(3);  // sleep to give consumer time to consume message
    }
    fprintf(stderr, "producer: ending\n");
}
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
/** This function consumes message strings. */
void *consumer(void *arg) {
    unsigned int msgcount = 0;
    bool quit = false;

    fprintf(stderr, "    consumer: thread beginning\n");

    do {
        fprintf(stderr, "    consumer: waiting to lock semaphore\n");
        // If producer has monitor, thread waits; when monitor unlocks, thread continues
        sem_wait(&semaphore);  // lock critical section
            fprintf(stderr, "    consumer: locked semaphore\n");
            if (strlen(message) > 0) {  // message available
                // report and truncate message
                fprintf(stderr, "    consumer: consuming message '%s'\n", message);
                quit = (strcmp(message, "quit") == 0);
                strcpy(message,"");
                msgcount++;
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

```
            } else {  // no message yet
                fprintf(stderr, "    consumer: waiting for next message to consume\n");
            }
            fprintf(stderr, "    consumer: unlocking semaphore\n");
        sem_post(&semaphore);  // unlock critical section
        sleep(1); // sleep to give producer time to produce message
    } while (!quit);

    fprintf(stderr, "    consumer: thread finishing\n");
    return ((char*)0 + msgcount);  // pointer encoded count
}
```

Threads and Concurrency

Synchronization Techniques: Semaphore in C

- One down-side to semaphores is that if there is no message to consume, the consumer must ”busy-wait” until the next message is available
- Similarly, if the previous message has not yet been consumed, the producer must “busy-wait” until the previous message was consumed.
- Is there an alternative?

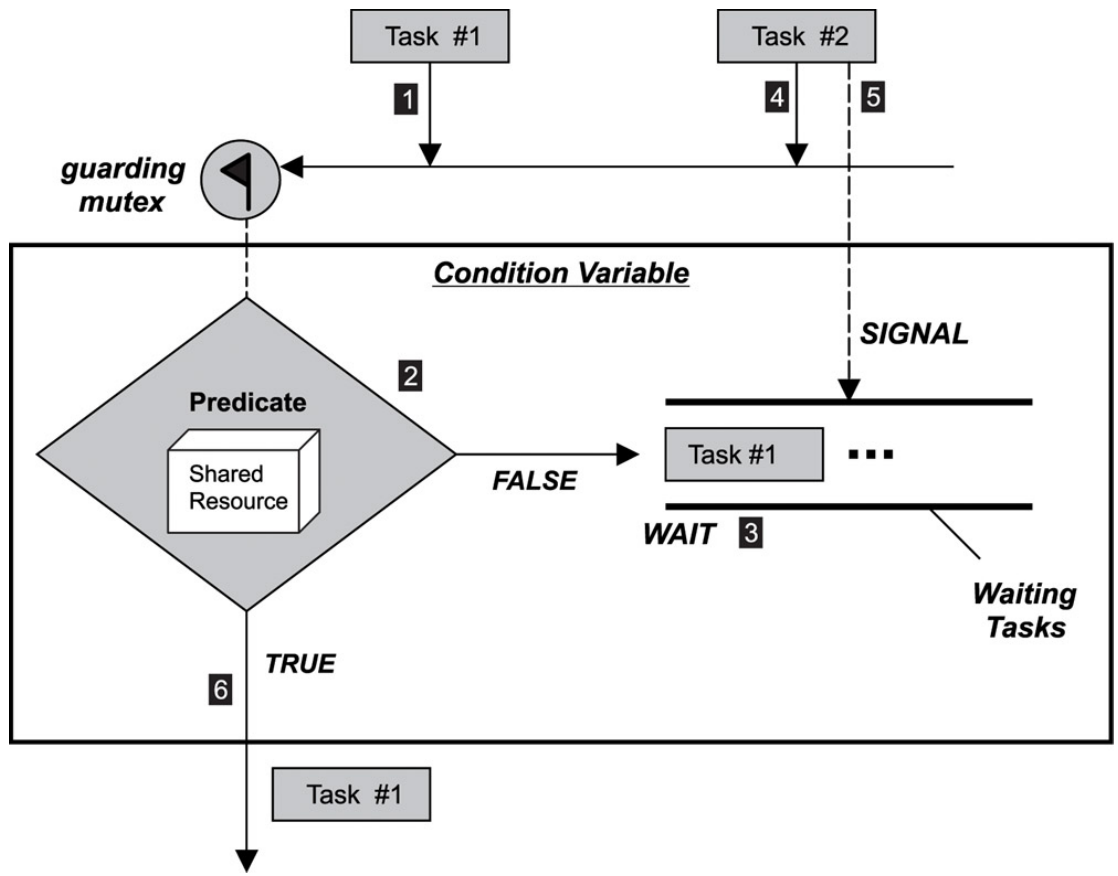
Threads and Concurrency

Synchronization Techniques: Monitor

- A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.
- A monitor consists of a *mutex* (lock) object and condition variables. A condition variable is basically a container of threads that are waiting for a certain condition.
- Monitors also have a mechanism for signaling other threads that their condition has been met.

Threads and Concurrency

Synchronization Techniques: Monitor



Threads and Concurrency

Synchronization Techniques: Monitor

- Monitors provide a mechanism for a thread to give up access temporarily to the critical section.
- The thread waits for some condition to be met, before regaining exclusive access and resuming its task.
- Monitors separate the concerns of mutual exclusion and conditional synchronization

Threads and Concurrency

Synchronization Techniques: Monitor

- Basic idea:
  - Permit access to shared resources only within a critical section
  - Maintains conditional invariant of shared resources
- General program structure
  - Entry section
    - Lock before entering critical section
    - Wait if invariant does not yet hold until it does
    - Key point: synchronization may involve wait
  - Critical section
    - Execute code that depends on invariant
  - Exit section
    - Unlock when leaving the critical section

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>

/** Monitor for message string.  */
pthread_mutex_t message_lock = PTHREAD_MUTEX_INITIALIZER;

/** Condition for message state changes.*/
pthread_cond_t message_cond = PTHREAD_COND_INITIALIZER;

/** shared string resource */
char message[128] = "";  // no message
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
int main(void) {
    pthread_t consumer_tid;  // thread for consumer

    // create and start consumer thread
    pthread_mutex_lock(&message_lock); // lock message monitor to block thread
        fprintf(stderr, "main: creating consumer\n");
        pthread_create(&consumer_tid, NULL, consumer, NULL);
        sleep(2);  // to slow things down
    pthread_mutex_unlock(&message_lock);  // unlock monitor
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
    // produce messages
    const char* messages[] = {"first", "second", "third", "quit"};
    producer(4, messages);

    // wait for consumer and report count
    void *msgcount;
    pthread_join(consumer_tid, &msgcount);
    fprintf(stderr, "main: finished with msgcount %u\n", (unsigned)(msgcount – (char*)0));
}
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
/** This function produces message strings.  */
void producer(int nmessages, const char *messages[]) {
    unsigned msgindex = 0;

    fprintf(stderr, "producer: beginning\n");
    pthread_mutex_lock(&message_lock); // lock message monitor
        while (msgindex < nmessages) {
            // wait for consumer
            if (strlen(message) != 0) {
                fprintf(stderr, "producer: waiting to send next message\n");
                do {
                    pthread_cond_wait(&message_cond, &message_lock);
                } while (strlen(message) != 0);
            }
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
            // send next message
            fprintf(stderr, "producer: sending message '%s'\n", messages[msgindex]);
            strcpy(message, messages[msgindex++]);

            pthread_cond_signal(&message_cond); // signal message available
            sleep(2);  // only to slow things down for demo purposes
        }
    pthread_mutex_unlock(&message_lock);  // unlock message monitor

    fprintf(stderr, "producer: ending\n");
}
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
/** This thread function consumes message srings */
void *consumer(void *arg) {
    unsigned int msgcount = 0;
    fprintf(stderr, "    consumer: thread beginning\n");
    bool quit = false;

    // If producer has monitor, thread waits; when producer unlocks, thread continues
    fprintf(stderr, "    consumer: waiting for monitor lock\n");
    pthread_mutex_lock(&message_lock);  // lock message monitor
        do {
            fprintf(stderr, "    consumer: monitor locked\n");
            if (strlen(message) == 0) {
                fprintf(stderr, "    consumer: waiting to receive next message\n");
                do {
                    pthread_cond_wait(&message_cond, &message_lock);
                } while (strlen(message) == 0);
            }
```

Threads and Concurrency

Synchronization Techniques: Monitor in C

- Here is a version of producer/consumer using a monitor.

```
            fprintf(stderr, "    consumer: received message is '%s'\n", message);
            quit = (strcmp(message, "quit") == 0);
            strcpy(message,"");

            pthread_cond_signal(&message_cond);  // signal message received
            msgcount++;
        } while (!quit);
        fprintf(stderr, "    consumer: unlocking monitor\n");
    pthread_mutex_unlock(&message_lock); // unlock message monitor

    fprintf(stderr, "    consumer: thread finishing\n");
    return ((char*)0 + msgcount);  // pointer encoded count
}
```