



Engineering excellence. **Sourced.**

POSIX Threads

Introduction

Arcent
A TCS COMPANY

Subhamita chatterjee

Overview

- Introduction to Threads
- Thread memory model
- Process vs Threads
- Thread states
- Thread benefits
- User mode thread and kernel mode thread
- What are POSIX Threads?
- Why use POSIX Threads?
- Pthread APIs
- Installing and building pthread
- Deadlock , race, starvation

Introduction to Threads

- A Thread is an independent stream of instructions that can be scheduled to run as such by the OS.
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently by the OS.
- A Thread exists within a process and uses the process resources.
- Threads only duplicate the essential resources it needs to be independently schedulable.

Contd.

- A thread will die if the parent process dies.
- A thread is “lightweight” because most of the overhead has already been accomplished through the creation of the process.
- Thread context is saved when not running.
- Has some per-thread static storage for local variables
- Has access to the memory and resources of its process
all threads of a process share this

Thread memory model

Multiple threads can be associated with a process

- Each thread has its *own* logical control flow (sequence of PC values)
- Each thread has its own thread id (TID)
- Each thread *shares* the same code, data, and kernel context

Thread 1 (main thread)

stack 1

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

Kernel context:

VM structures
Descriptor table
brk pointer

Thread 2 (peer thread)

stack 2

Thread 2 context:
Data registers
Condition codes
SP2
PC2

Threads Memory Model cont.

- Each thread runs in the context of a process
- Each thread has its own separate thread context
 - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers
- All threads share remaining process context
 - Code, data, heap, and shared library segments of process virtual address space
 - Open files and handlers

Process vs Threads

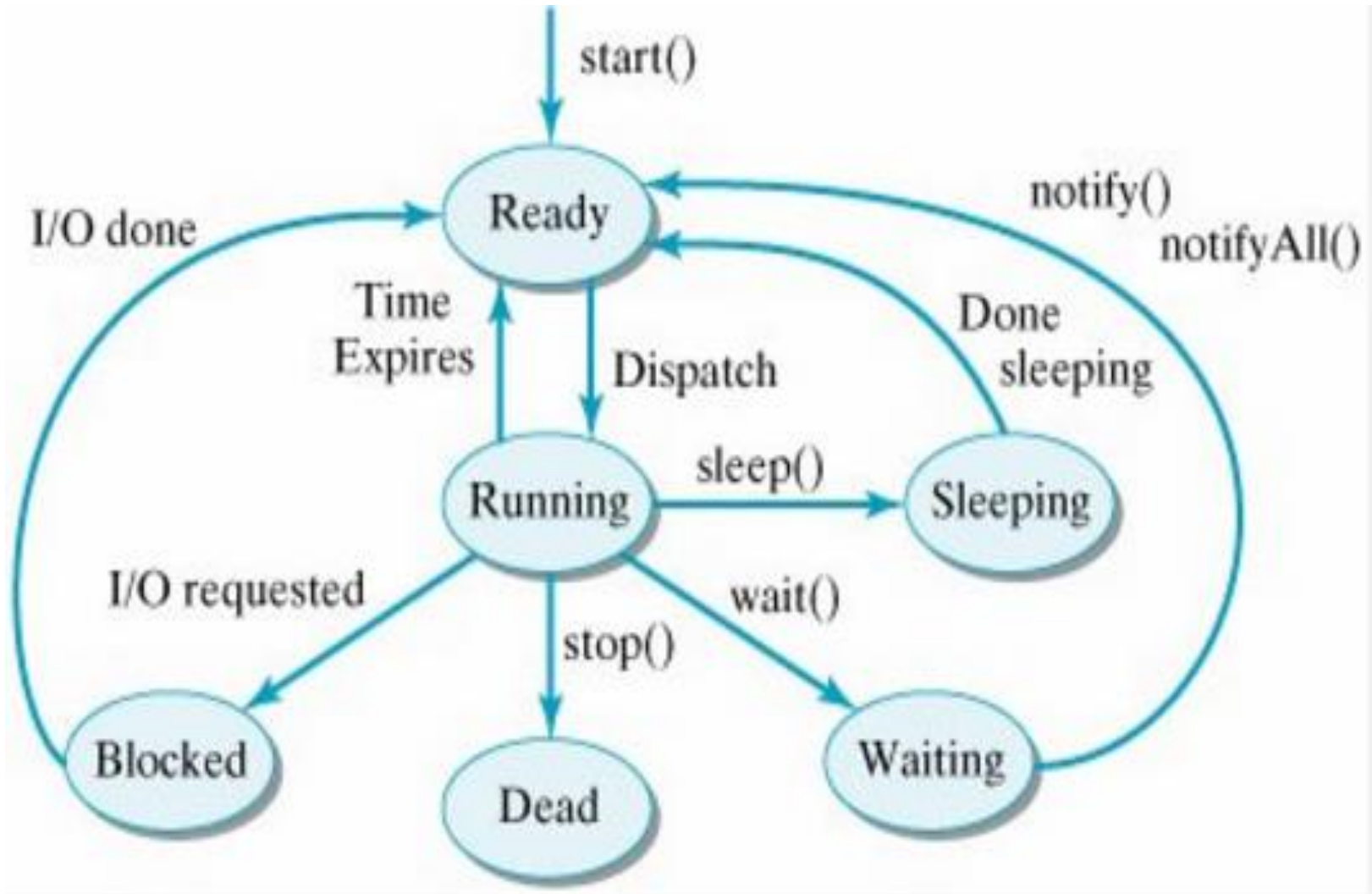
Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.

Thread States: Life Cycle of a Thread

Thread states

- Ready state (runnable state)
- Running state
- Dead/Terminated state
- Blocked state
- Waiting state
- Sleeping state

Thread States: Life Cycle of a Thread

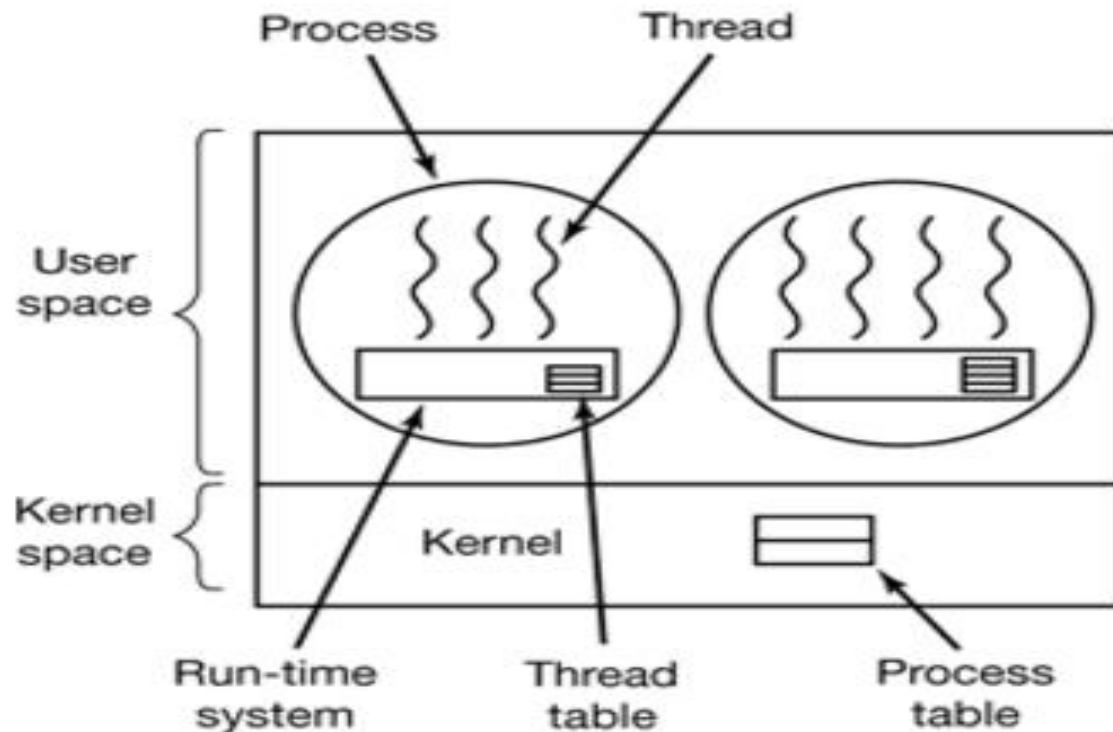


Thread benefits

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel. No IPC mechanisms required

User mode threads

User mode programs can access *user threads* through a thread library. User threads are part of a portable programming model. User threads are mapped to kernel threads by the threads library, in an implementation dependent manner. The threads library uses a proprietary interface to handle kernel threads. Kernel threads cannot be accessed from the user mode environment, except through the threads library.

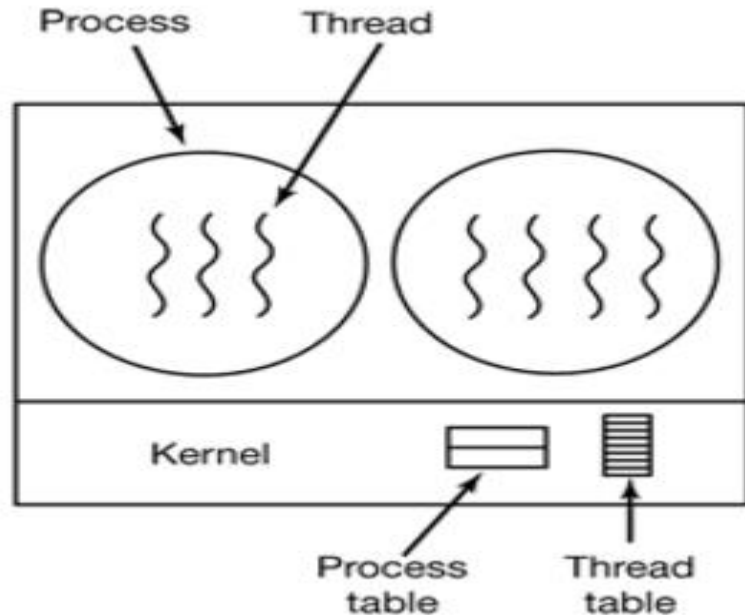


User mode threads contd.

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges
- Scheduling is application specific
- Created by thread library such as C-threads(Mach) or pthreads(POSIX).

Kernel mode thread

A kernel thread is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs in user mode environment when executing user functions or library calls; it switches to kernel mode environment when executing system calls.



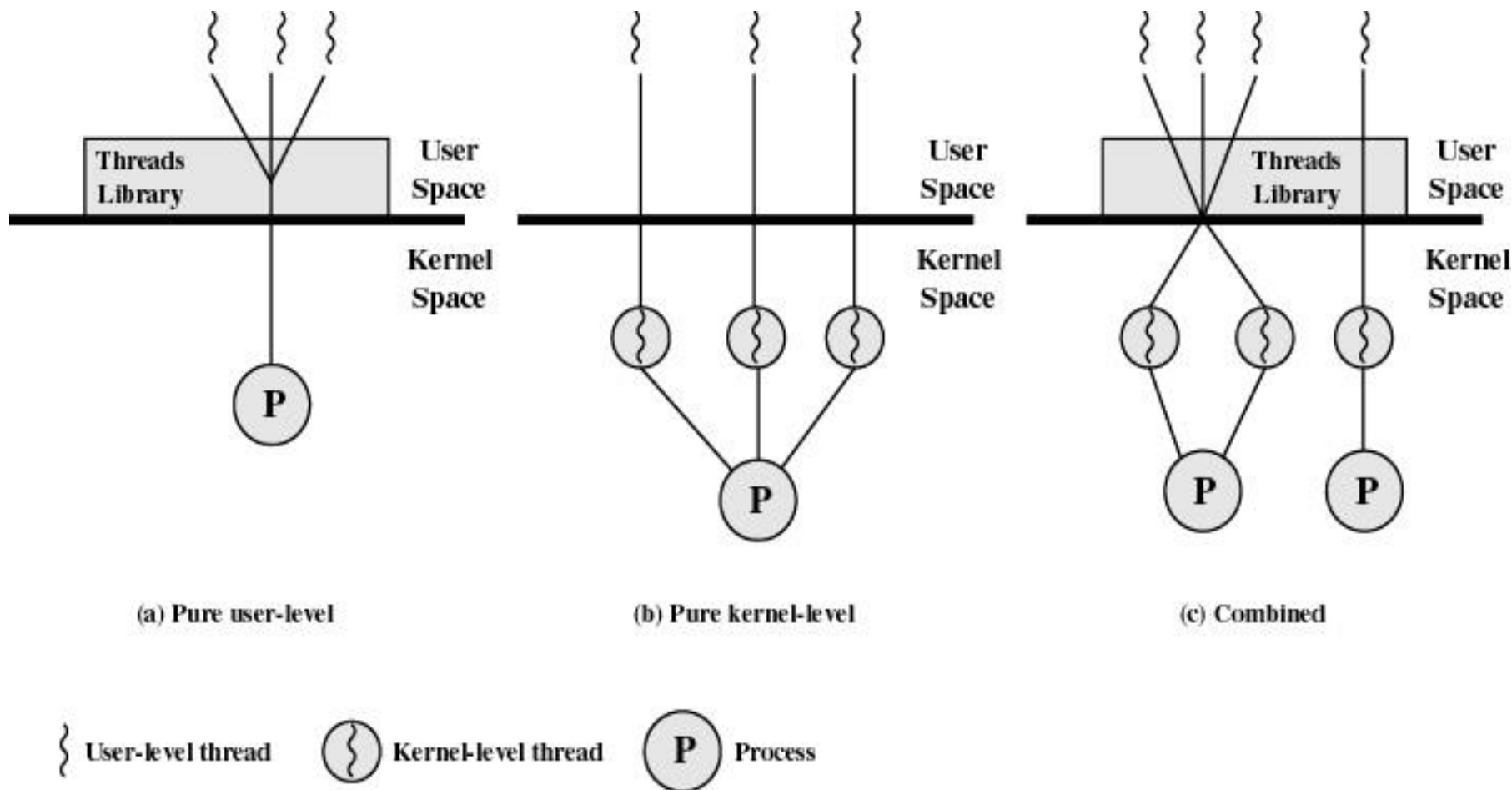


Figure 4.6 User-Level and Kernel-Level Threads

What are POSIX Threads (PThreads) ?

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are **Pthreads**.
- Pthreads are C language programming types defined in the pthread.h header/include file.
- There are around 100 Pthreads procedures, all prefixed "pthread_" and they can be categorized into four groups:
 1. Thread management - creating, joining threads etc.
 2. Mutexes
 3. Conditional Variables
 4. Synchronization between threads using read/write locks and barriers
- The POSIX semaphore API works with POSIX threads but is not part of threads standard, having been defined in the *POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)* standard. Consequently, the semaphore procedures are prefixed by "sem_" instead of "pthread_".

Why use POSIX Threads?

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks that can execute concurrently.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways.

Thread Application Example

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
- Asynchronous event handling: tasks that service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism.

Posix Threads (Pthreads) Interface

- Creating and reaping threads

`pthread_create, pthread_join`

- Determining your thread ID

`pthread_self`

- Terminating threads

`pthread_cancel, pthread_exit`

`exit` [terminates all threads], `return` [terminates current thread]

- Synchronizing access to shared variables

`pthread_mutex_init, pthread_mutex_[un]lock`

`pthread_cond_init, pthread_cond_[timed]wait`

Installing and building pthread

The pthread.h (POSIX Thread) should be available by default with GCC

If you are going to compile a C program with pthread.h in LINUX using GCC or G++ you will have to use **-lpthread** option after the compile command.

```
gcc -lpthread -o output_file test.c
```

after that ./output_file provides output for program and here program test.c is the pthread program you have implemented.

Creating threads

Always include pthread library:

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

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

This creates a new thread of control that calls the function start_routine.

It returns a zero if the creation is successful, and thread id in tp (first parameter). attr is to modify the attributes of the new thread. If it is NULL default attributes are used.

The arg is passing arguments to the thread function.

Using threads

1. `#include <pthread.h>` at the top of your header.
2. Declare a variable of type `pthread_t`
3. Define a function to be executed by the thread.
4. Create the thread using `pthread_create`
5. Make sure creation is successful by checking the return value.
6. Pass any arguments needed through 'arg (packing and unpacking arg list necessary.)

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *howdy(void *vargp);

int main() {
    pthread_t tid;

    Pthread_create(&tid, NULL, howdy, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *howdy(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

*Thread attributes
(usually NULL)*

*Thread arguments
(void *p)*

*return value
(void **p)*

Joining threads

Prototype : `int pthread_join(pthread_t thread, void **retval);`

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.

A call to this function makes a thread wait for another thread whose thread id is specified by *thread* in the above prototype.
When the thread specified by *thread* exits its completion status is stored and returned in *retval*

Thread termination

Implicit : Simply returning from the function executed by the thread terminates the thread. In this case thread's completion status is set to the return value.

Explicit : Use `pthread_exit`.

Prototype: `void pthread_exit(void *retval);`

The `pthread_exit()` function terminates the calling thread and returns a value via *retval*

The `pthread_exit()` routine is called after a thread has completed its work and it no longer is required to exist.

If the main program finishes before the thread(s) do, the other threads will continue to execute if a `pthread_exit()` method exists.

The `pthread_exit()` method does not close files; any files opened inside the thread will remain open, so cleanup must be kept in mind.

Contd.

Prototype: `int pthread_cancel(pthread_t thread);`

`pthread_cancel` - send a cancellation request to a thread

The **`pthread_cancel()`** function sends a cancellation request to the thread *thread*. Whether and when the target thread reacts to the cancellation request depends on thread attributes.

Threading problems

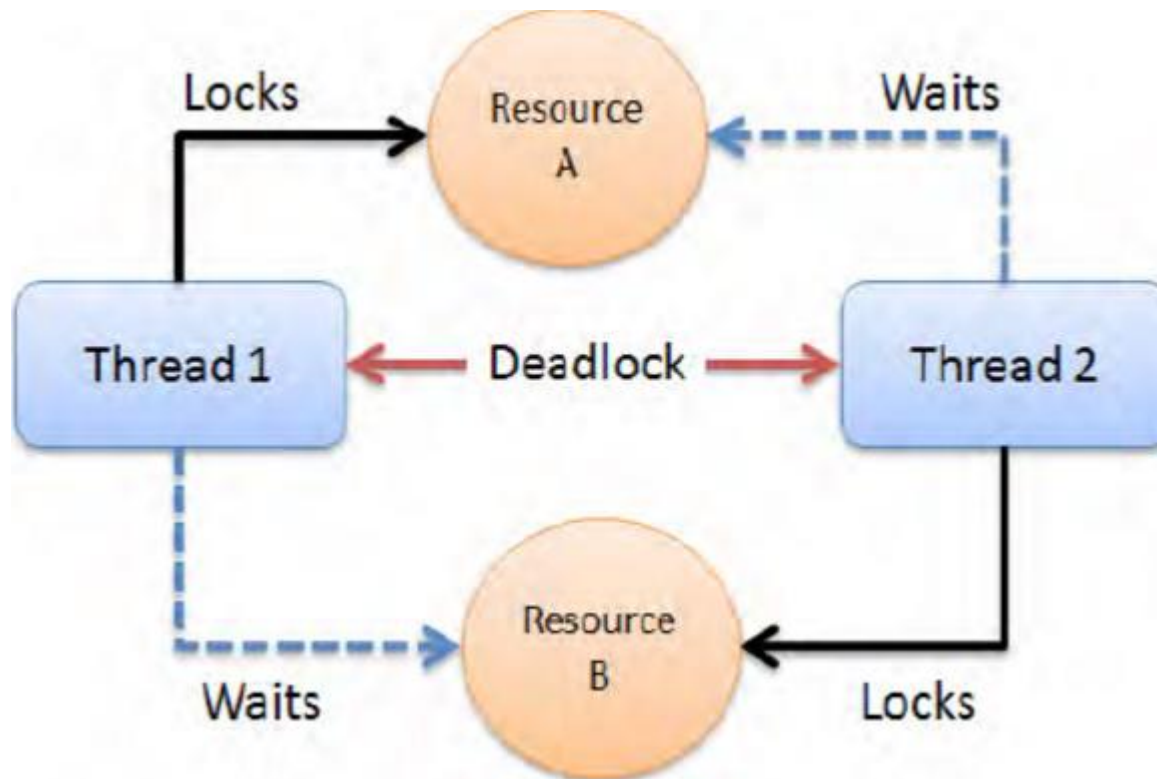
What can go wrong when you write thread code and how can you guard against them?

Basically, the three threading problems are

1. **Deadlock**
2. **Races**
3. **Starvation**

Deadlock

The simplest deadlock condition is when there are two threads and thread1 can't progress until thread2 finishes, while thread2 can't progress until thread1 finishes. This is usually because both need the same two resources to progress. Various symmetry breaking algorithms can prevent this in the two thread or larger circle cases.

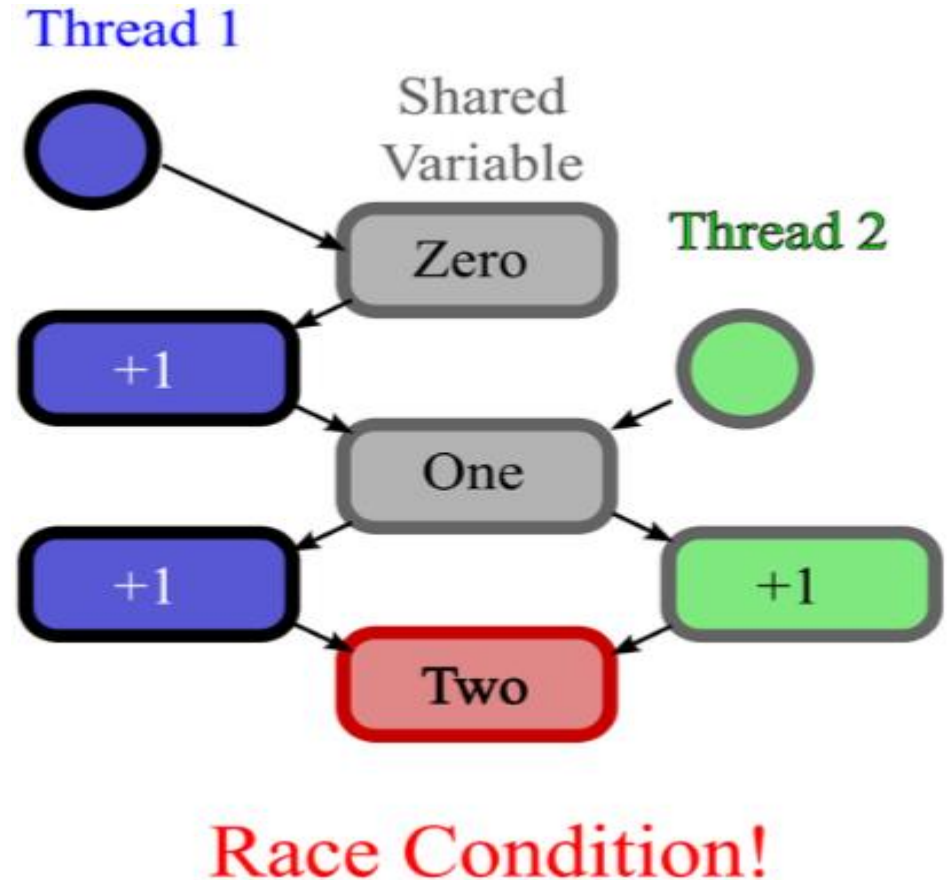


Races

Races happen when one thread changes the state of some resource when another thread is not expecting it.

Ex - Such as changing the contents of a memory location when another thread is part way through reading, or writing to that memory.

Solution – protect concurrent access to data using a **mutex** (Locking methods are the key here.)



Starvation


Starvation happens when a thread needs a resource to proceed, but can't get it. The resource is constantly tied up by other threads and the one that needs it can't get in. The scheduling algorithm is the problem when this happens. Look at algorithms that assure access.

Debugging threaded programs

printf is useful, but it takes time to execute – why is this potentially a problem when writing multithreaded programs?



GDB is pthreads-aware and supports inspecting the state of running threads.

A close-up photograph of four hands, two from a darker-skinned person and two from a lighter-skinned person, positioned to form a square frame. The hands are open, with fingers pointing towards the corners, creating a central negative space. The background is dark and out of focus.

We are the **source.**

The future starts here. With you. With us. With Aricent.

Aricent® Engineering excellence. **Sourced.**