# COP 4338 Class 17

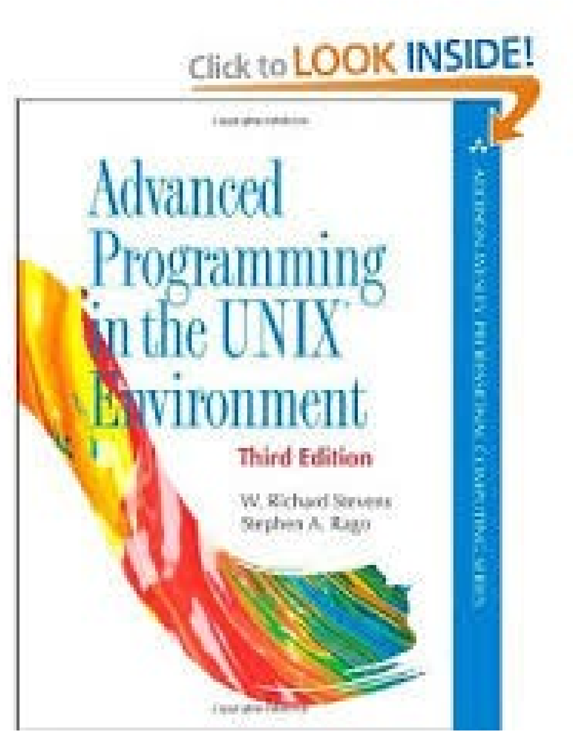
**Last Class:**

**Ch 8: Process Control**: (check slides, example, and book)

**Today's Class:**

**Continue Process Control and Threads!**

Note: Second homework , worth 15% Due March 9th



# Chapter 11. Threads

# COP-4520

Taken from: Introduction to Parallel Computing

By: Dr. Liu

Modified by:
Leonardo Bobadilla

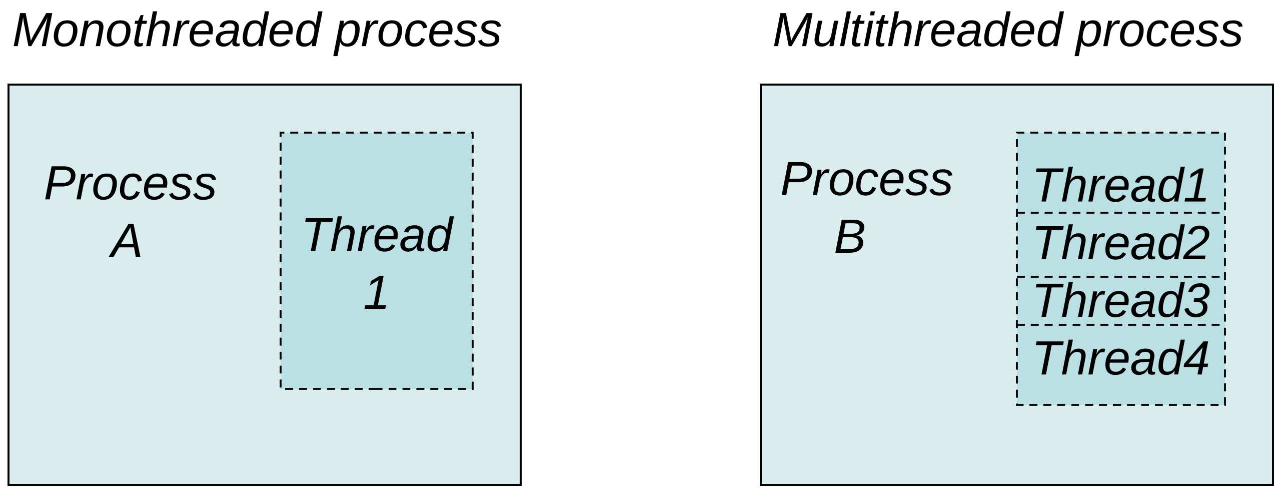
## A good resource!

- I strongly encourage you to follow this tutorial:

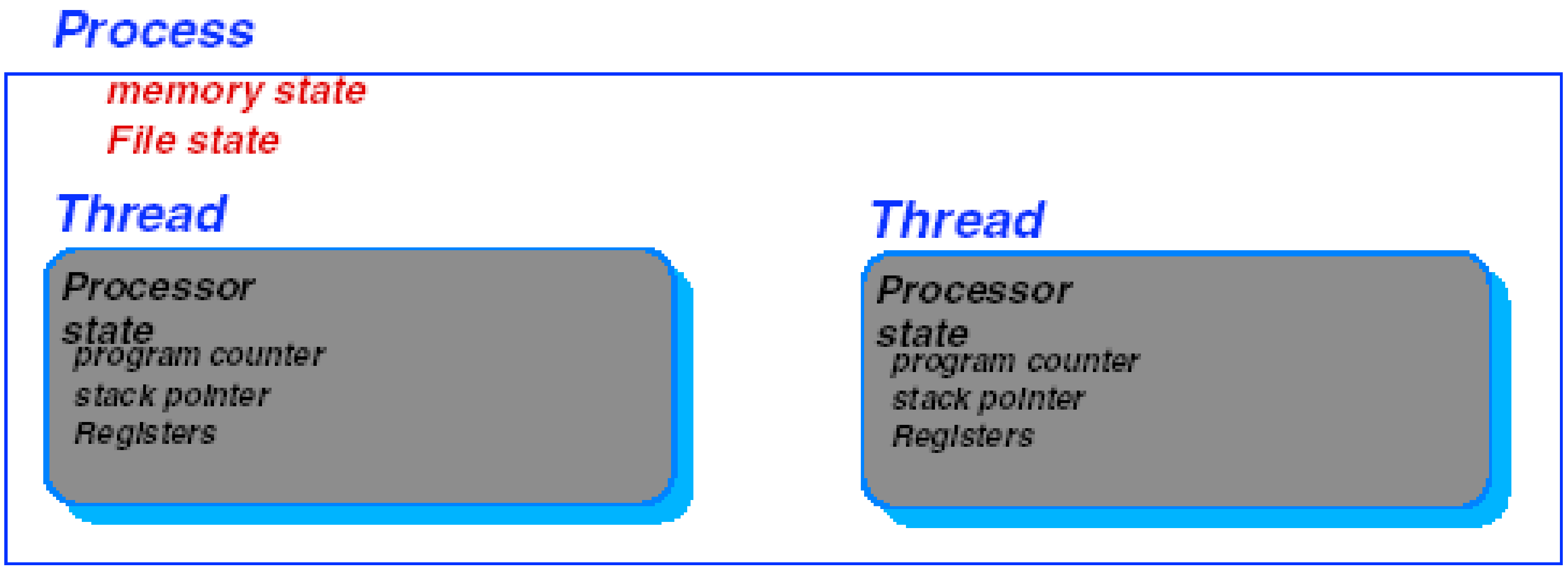
https://computing.llnl.gov/tutorials/pthreads/

## What Are threads?

- A thread:
  - Is an independent flow of control.
  - Operates within a process with other threads.



# Threads Versus Processes



- Threads vs. Processes:
  - Threads use and exist within the process resources
  - A thread maintains its own stack and registers, scheduling properties, set of pending and blocked signals.
  - By default, threads’ data are shared, processes’ data are private.
- Secondary Threads vs. Initial Threads
  - An initial thread is created automatically when a process is created.
  - Secondary threads are peers.

## Why Threads?

- To realize potential program performance gains:
  - On a uniprocessor, multi-threaded processes hide latency for memory access, I/O, and communication.
  - On a multiprocessor system, a process with multiple threads provides potential parallelism.
- Benefits of multithreaded programming (rather than using multiple processes):
  - Compared to the cost of creating and managing a process, a thread can be created and managed with much less operating system overhead.
  - All threads within a process share the same address space. Inter-thread communication is more efficient and than inter-process communication.

## Threads Versus Processes

Process

fork is expensive (time & memory)

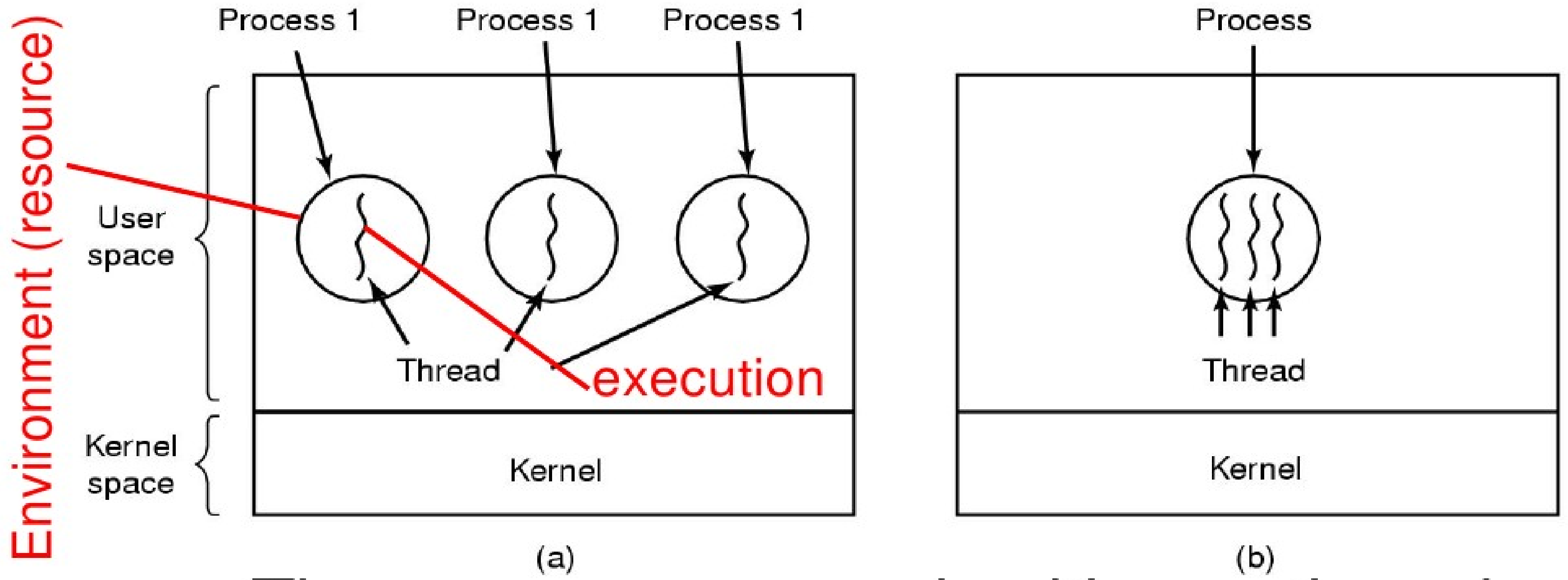
Thread

-Lightweight process

-Shared data space

-Does not require lots of memory or startup time

# Threads Versus Processes



a) Three processes each with one thread
b) One process with three threads

## What are POSIX threads (pthreads)?

- Historically, hardware vendors have implemented their own proprietary versions of threads. Not portable.

  pthread is a standardized thread programming interface specified by the IEEE POSIX (portable operating systems interface) in 1995.
- pthreads are defined as a set of C language programming types and procedure calls, implemented with a header file (**pthread.h**) and a library **(libpthread.a**).

## Outline

- Thread basics: creation and termination.
- Synchronization primitives:
  - Mutual exclusion.
  - Conditional variables.
- Thread cancellation.
- Composite synchronization constructs:
  - Read-write locks.
  - Barriers.

## Thread Creation

- Creating a thread:

```
int pthread_create(
    pthread_t *tid,                          /* thread id */
    const pthread_attr_t *attr,              /* thread attributes */
    void *(*start_rountine) (void *),        /* pointer to function */
    void *arg                                /* argument for function */
  );
```

- Returns the ID of the new thread via the *tid* argument.
- The *attr* parameter is used to set thread attributes, *NULL* as default.
- The *start_routine* is the C routine that the thread will start executing once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void. You can use a data structure if you want to pass multiple arguments.

threadid.c threadexample1.c 14-threads.c simplechild1.c simplechild2.c simplechild3.c

## Thread Creation Arguments

Study hello_arg1.c

Study hello_arg2.c

## Thread Termination

- There are many ways to terminate a thread:
  - The thread returns from its starting routine;
  - The thread calls **pthread_exit** ();
  - The thread is canceled by another thread via **pthread_cancel**();
  - The thread receives a signal that terminates it;
  - The entire process is terminated.

## Thread Termination and Join

**void pthread_exit (void *status) ;**

- This function is used by a thread  to terminate.  The return value is passed as a pointer.

**int pthread_join (pthread_t tid, void** status);**

- The pthread_join() subroutine blocks the calling thread until the specified thread *tid* terminates.
- The function returns 0 on success, and negative on failure.
- The returned value is a pointer returned by reference.  If you do not care about the return value, you can pass NULL for the second argument.

## Hello World!

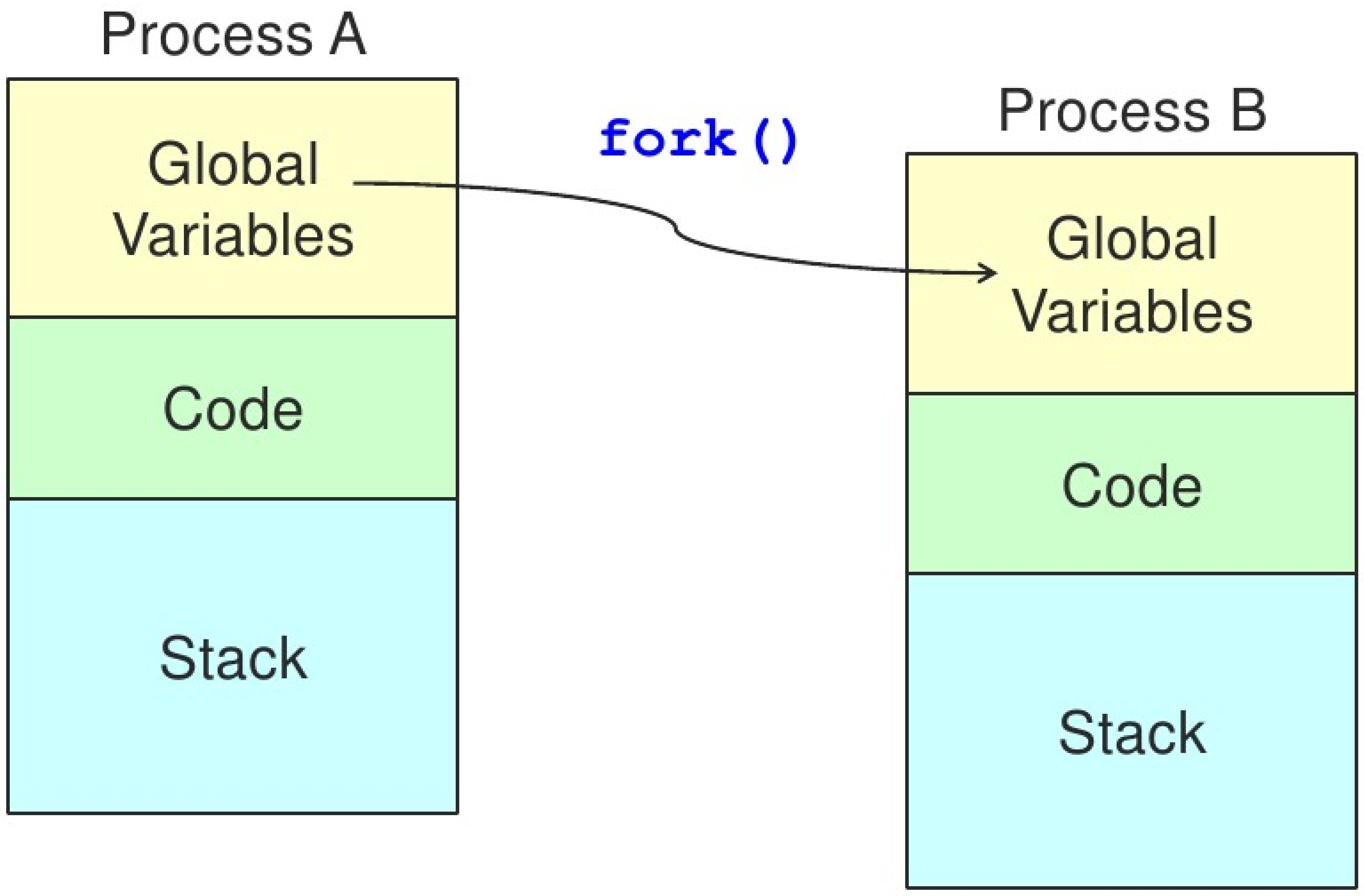
```
#include <pthread.h>
#include <stdio.h>

void *PrintHello(void * id) {
   printf(“Thread%d: Hello World!\n", (int)id);
   pthread_exit(NULL); /* not necessary */
}

int main (int argc, char *argv[]) {
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
   pthread_join(thread0, NULL);
   pthread_join(thread1, NULL);
   return 0;
}
```

See helloworldthreads.c pthreadjoinex.c

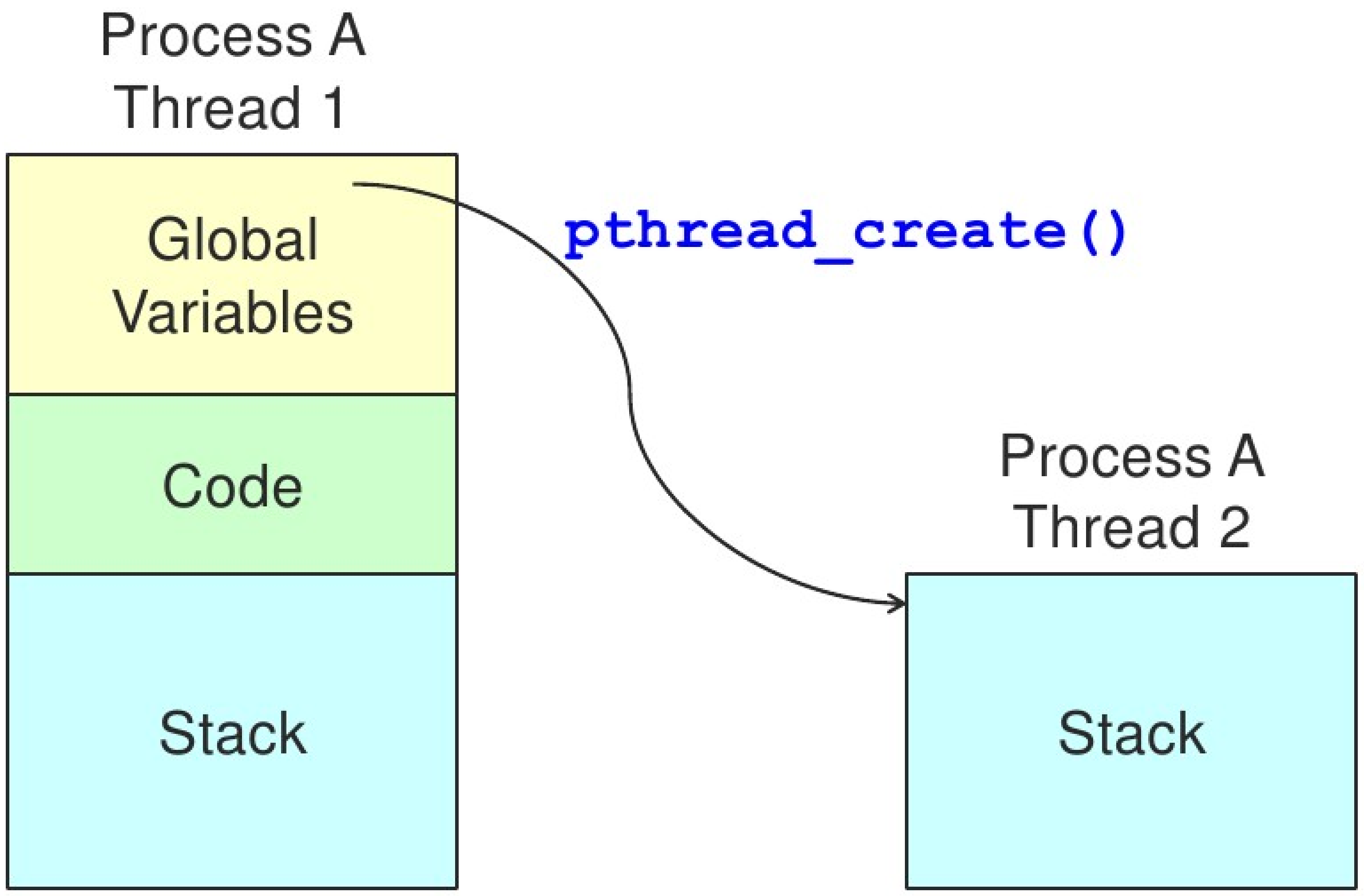
## Thread vs Process creation

- `fork()` clones the process
  - Two separate processes with independent destinies
  - Independent memory space for each process
- `pthread_create()`
  - Start from a function
  - Share memory

# Thread vs Process creation



# Thread vs Process creation



## Important Points

A thread is the lightest unit of work that can be scheduled to run on the processor

When creating a thread you

- Indicate which function the thread should execute

When a new thread is created

- It runs concurrently with the creating thread
- It shares common data space

## Why threads over processes?

Creating a new process can be expensive

Time

-A call into the operating system is needed

-Context-switching involves the operating system

Memory

-The entire process must be replicated

The cost of inter-process communication and synchronization of shared data

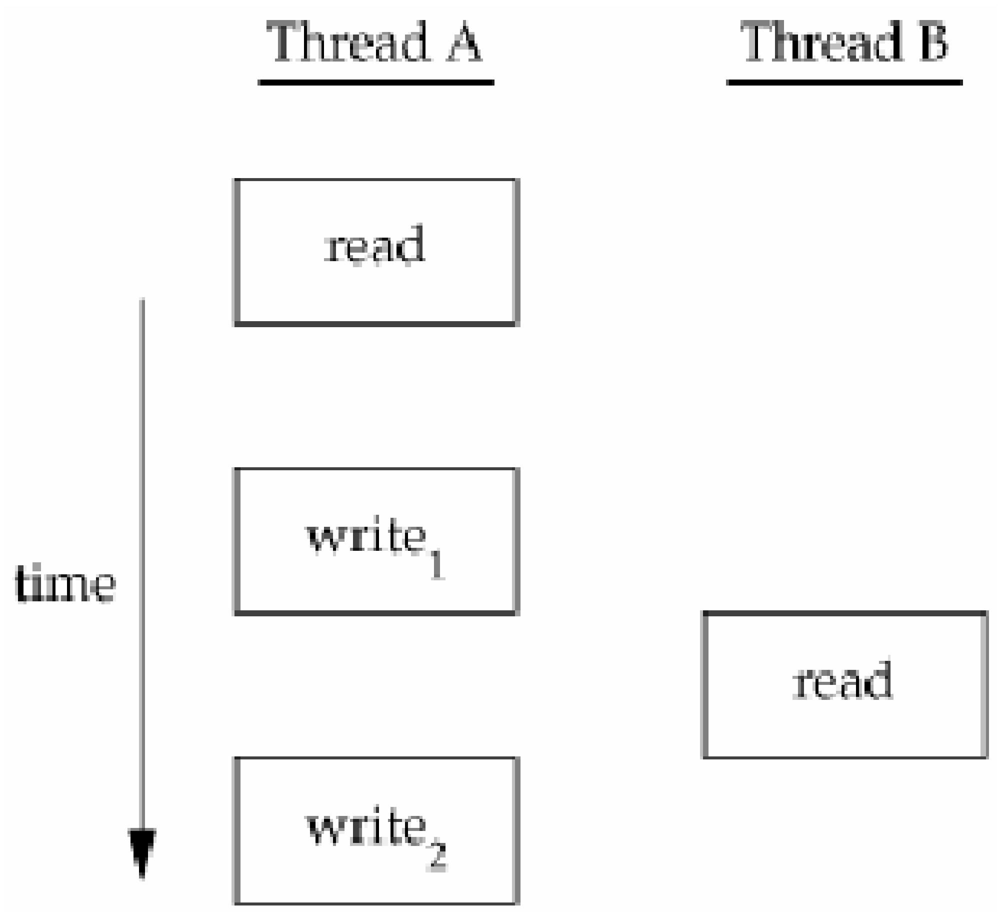
-May involve calls into the operation system kernel

-Threads can be created without replicating an entire process

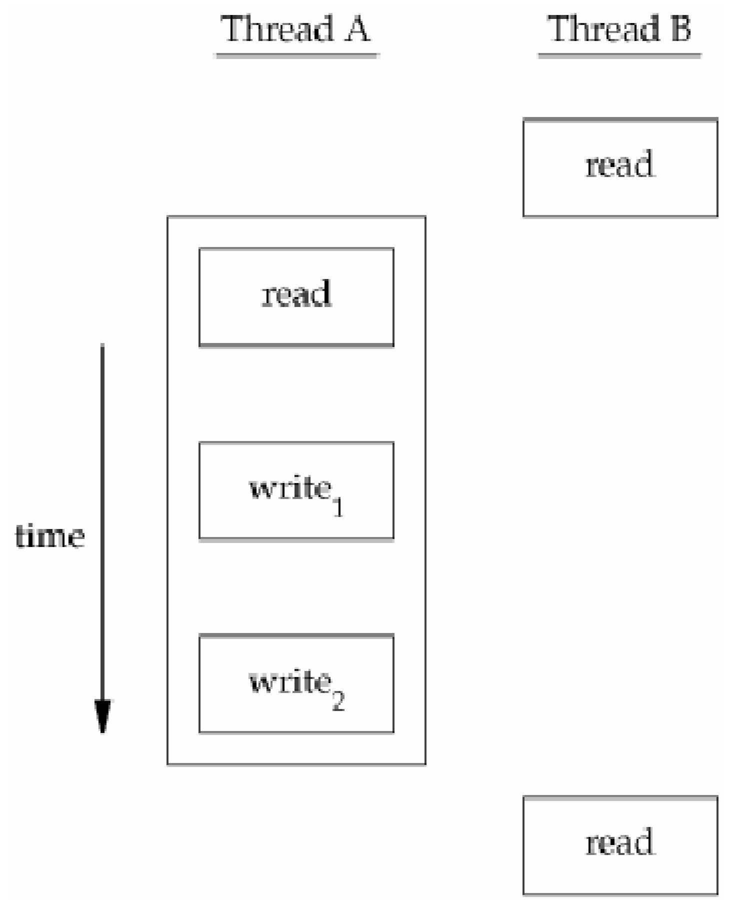
## Comparisson

| Property | Processes created with fork | Threads of a process | Ordinary function calls |
|---|---|---|---|
| variables | Get copies of all variables | Share global variables | Share global variables |
| IDs | Get new process IDs | Share the same process ID but have unique thread ID | Share the same process ID (and thread ID) |
| Data/control | Must communicate explicitly, e.g., use pipes or small integer return value | May communicate with return value or carefully shared variables | May communicate with return value or shared variables |
| Parallelism (one CPU) | Concurrent | Concurrent | Sequential |
| Parallelism (multiple CPUs) | May be executed simultaneously | Kernel threads may be executed simultaneously | Sequential |

## Thread Synchronization



# Thread Synchronization



## Mutual Exclusion

- For thread synchronization and protecting shared data when multiple writes occur.
- A **mutex** variable acts like a "lock”, protecting access to a shared data resource.
- Only one thread can obtain (or own) the lock at any given time. Thus, even if several threads try to lock a **mutex**, only one thread will be successful. No other thread can obtain the lock until the owning thread unlocks that mutex.
  - Threads must "take turns" accessing protected data.

## Creating / Destroying Mutexes

```
pthread_mutex_t mutex;
int pthread_mutex_init(pthread_mutex_t* mutex,
    pthread_mutexattr_t* attr);
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

- Mutex variables must be declared and initialized before they can be used.
- Usually, one specifies *attr*, the mutex object attributes, to be NULL to accept defaults.
- It’s a good habit to destroy the mutex after use.

# Locking / Unlocking Mutexes

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_trylock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- The pthread_mutex_lock() routine is used by a thread to acquire a lock on the specified *mutex* variable. The thread **blocks** if the mutex is already locked by another thread.
- pthread_mutex_trylock() will attempt to lock a mutex. However, if the mutex is already locked, the routine will **return immediately** with a "busy" error code.
- pthread_mutex_unlock() will unlock a mutex; must be called by the owning thread.

## A Typical Scenario

```
pthread_mutex_t mutex;

int main() {
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&t0, NULL, p, NULL);
  pthread_create(&t1, NULL, q, NULL);
  …
  pthread_join(t0, NULL);
  pthread_join(t1, NULL);
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

syncexample.c mutex1.c bank.c

```
void p(void* arg) {
   …
   pthread_mutex_lock(&mutex);
   /* access shared variable */
   pthread_mutex_unlock(&mutex);
   …
}
void q(void* arg) {
   …
   pthread_mutex_lock(&mutex);
   /* access shared variable */
   pthread_mutex_unlock(&mutex);
   …
}
```

# Deadlocks

- Deadlocks occur when a thread try to relock a mutex it already owns:

```
f( ) {
        pthread_mutex_lock ( & mutex);
        …
        g( );
        …
        pthread_mutex_unlock();
}
```

```
g( ) {
        pthread_mutex_lock ( & mutex);
        …
        pthread_mutex_unlock();
}
```

- Deadlocks may also occur when mutexes are locked in reverse order:

```
/* Thread A */
pthread_mutex_lock(&mutex1);   time 0
pthread_mutex_lock(&mutex2);   time 2
```

```
/* Thread B */
pthread_mutex_lock(&mutex2);   time1
pthread_mutex_lock(&mutex1);   time3
```

Notes: To avoid this, successive mutexes should be locked in the same order.

## Basic Rules on Using Mutexes

- Avoid deadlocks:
  - If it must use 2 locks simultaneously, be careful of the locking order.
- Improve performance:
  - Move all unnecessary code outside the critical section; the code inside a critical section is doomed to be sequential.
  - Accesses to shared data should be put together if they can be covered by a single lock/unlock.

## Condition Variables

- Condition variables allow threads to synchronize based upon the actual data value;
- Without condition variables, the thread has to check the condition continuously (i.e., polling). This can be very resource consuming;
- With condition variables, the thread sleeps and waits on the condition variable;
- A condition variable is always used in conjunction with a mutex.

## Condition Variables

### standard "waiter" idiom:

```
pthread_mutex_lock( &mutex );
while ( !condition )
        pthread_cond_wait( &cond, &mutex );
do_something();
pthread_mutex_unlock( &mutex );
```

This function atomically releases mutex and causes the “waiter” to block on the condition variable “cond”. Upon successfully return, the mutex has been locked and owned by the “waiter” again.

### standard "signaler" idiom:

```
pthread_mutex_lock( &mutex );
do_something();
// make condition TRUE
pthread_cond_signal( &cond );
pthread_mutex_unlock( &mutex );
```

If more than one thread is blocked on “cond”, the scheduling policy determines the order in which threads are unblocked. pthread_cond_broadcast ( ) unblocks all threads currently blocked on “cond”.

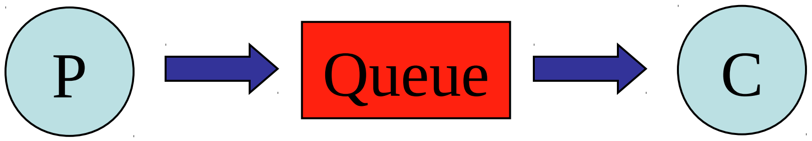
Notes: pthread_cond_wait( ) should be called while mutex is locked;
pthread_cond_signal( ) should be called after mutex is locked;
It is a logical error to call signal before calling wait.

## Condition Variables

Check condvar.c
cond1.c

## Classical Producer-Consumer Problem

- The producer produces data that are consumed by the consumer.



- The producer will only be able to produce a data while the queue has available space.
- The consumer will only be able to consume a data while the queue has data in it.

```
pthread_mutex_t mutex;                              /* declare mutex */
pthread_cond_t cond_empty, cond_full;               /* declare two conditional variables for different
    queue state */
int data_available = 0;                             /* declare and initialize global variables */

main() {
  pthread_mutex_init(&mutex, NULL);                 /* initialize the mutex */
  pthread_cond_init(&cond_empty, NULL);             /* initialize the condition variables */
  pthread_cond_init(&cond_full, NULL);
}

void producer(void* arg) {
  for(;;) {
    produce_data();                                 /* produce the data first, outside the critical section */
    pthread_mutex_lock(&mutex);                     /* will block until the lock is obtained */
    while(data_available)                           /* IMPORTANT: check whether there’s space to insert
    data */
      pthread_cond_wait(&cond_empty, &mutex);       /* if not, wait until there is space */
    insert_data_into_queue();                       /* insert the data into the queue, knowing there’s
    space */
    data_available = 1;                             /* set the condition that the data is now available for
    retrieve */
    pthread_cond_signal(&cond_full);                /* time to wake up the consumer */
    pthread_mutex_unlock(&mutex);                   /* get out of the critical section */
  }
}

void consumer(void* arg) {
  for(;;) {
    pthread_mutex_lock(&mutex);                     /* try to enter the critical section */
    while(!data_available)                          /* IMPORTANT: check whether there’s data in the
    queue */
      pthread_cond_wait(&cond_full, &mutex);        /* if not, wait until there’s data */
    retrieve_data_from_queue();                     /* retrieve the data from the queue, knowing there
    must be one */
    data_available = 0;                             /* set the condition that the queue is now empty */
    pthread_cond_signal(&cond_empty);               /* time to wake up the producer to produce more */
    pthread_mutex_unlock(&mutex);                   /* get out of the critical section */
    consume_data();                                 /* consume the data, OUTSIDE of the critical section
    */
```

## Example

run pc_1.c

## Thread Cancellation

- A thread may cancel itself or other threads.
- It is not guaranteed that the specified thread will receive or act on the cancellation.
- If cancellation is acted upon, it will terminate the thread as if **pthread_exit**() is called.

**int pthread_cancel(pthread_t thread);**

## Composite Synchronization Constructs

- POSIX threads API only provides a basic set of synchronization primitives: mutex, condition variables, etc.
- We can use these synchronization primitives to build higher level constructs.
- Read-Write Locks & Barriers (check that in the book)

## Composite Synchronization Constructs

- Suppose A, B, and C are NxN matrices, we are suppose to calculate C=AxB.
- Suppose we have p^2 number of threads (i.e., the number of threads is a square number) and we assume N is divisible by p.

## Composite Synchronization Constructs

- We can multiply A and B block by block. For explanation, let's assume p=4. That is, we have 16 threads. Matrix A can thus be divided into 4x4 submatrics, as follows:

  A00 A01 A02 A03

  A10 A11 A12 A13

  A20 A21 A22 A23

  A30 A31 A32 A33

## Composite Synchronization Constructs

- You can do the same for matrix B and matrix C.
- Now, to calculate matrix C from A and B. We can have each thread to be in charge of a submatrix of C. Let's assume it's the ith row and jth collumn (where 0<=i<=3, and 0<=j<=3 in this example): Cij = Ai0xB0j + Ai1xB1j + Ai2xB2j + Ai3xB3j

## References

The previous and following slides are mostly taken from: http://www.cs.stevens.edu/~jschauma/810D/

-