



# POSIX THREADS & IPC SYNC

**Creation & Termination:** `pthread_create/pthread_join`

**Mutex:** `pthread_mutex_init/lock/unlock/trylock, destroy`

**Thread & Mutex attr:** `pthread_attr_*/pthread_mutex_attr`

**Mutex and Deadlock Avoidance**

**Barriers:** `pthread_barrier_init/wait/destroy`

**Semaphores:** `sem_open, sem_close, sem_unlink (named)`

`sem_init, sem_wait, sem_post (all)`

`sem_close, sem_destroy (unnamed)`

**Cond. Var.:** `pthread_cond_init/wait/signal/broadcast/destroy`

# INTRODUCTION

A thread defines a single execution stream within a process

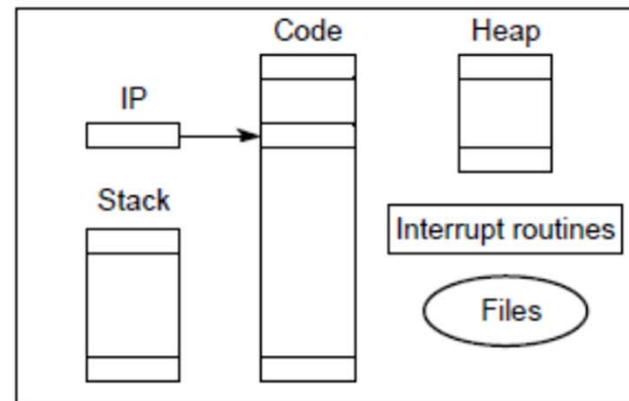
In this chapter, we examine

- How to manage multiple threads of control (or simply threads) to perform multiple tasks within a single process
- All threads within a single process can access the same process components, such as file descriptors and memory

# COMPARE PROCESSES & POSIX THREADS

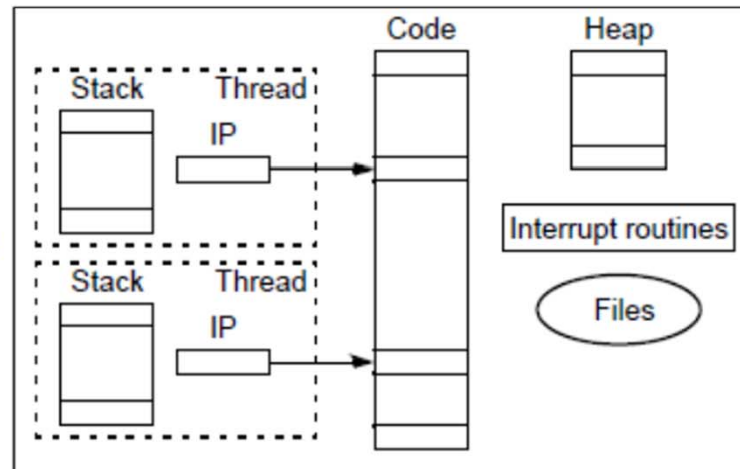
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



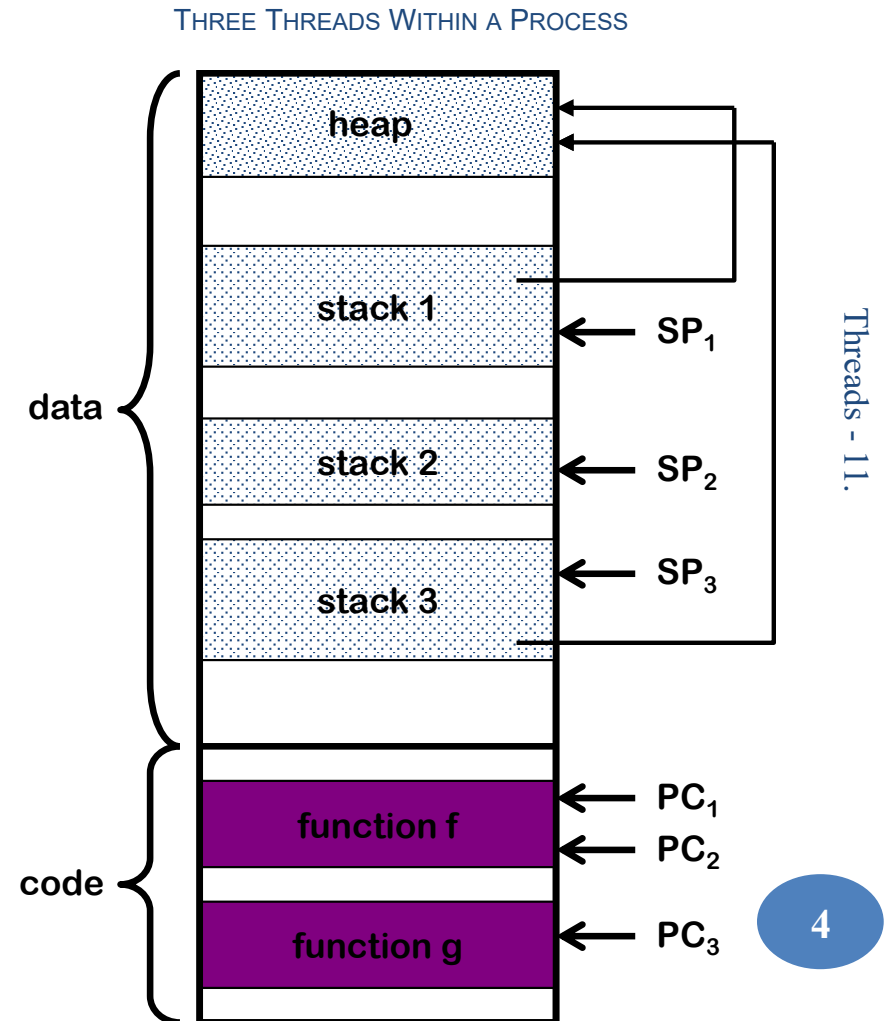
# THREADS VS PROCESSES

Different than process

- No initial system calls, like shared memory, sockets etc
- Simpler/faster communication & context switch (lightweight)

Similar to process

- Each process/thread can be independently scheduled to a CPU
  - Both viewed by Linux kernel as `task_struct`



# COMPARE PROCESSES & POSIX THREADS

A thread consists of the info to represent an execution context, including

- a thread ID that identifies the thread within a process,
- a set of register values,
- a stack,
- a scheduling priority and policy,
- a signal mask,
- an errno variable, and
- thread-specific data (Section 12.6)

Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the file descriptors

## PTHREAD ID

- Every thread has a unique thread ID (TID) that identifies the thread in the kernel
- The TID is of `pthread_t` type
- A thread can obtain its own TID by calling the `pthread_self` function

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

```
pthread_t pthread_self(void);
```

Returns: the thread ID of the calling thread

# PTHREAD CREATION

Threads are created by calling `pthread_create`

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void *),
                  void *restrict arg);

Returns: 0 if OK, error number on failure
```

## PTHREAD CREATE - ARGUMENTS

- `pthread_t *restrict tidp`: Thread ID pointer of the newly created POSIX thread, when `pthread_create` returns successfully
- `const pthread_attr_t *restrict attr`: used to customize thread attributes: concurrency level, scheduling policy/parameters, detach state, stack address/size, stack guard size. Setting this to NULL creates a thread with default attributes
- `void *(*start_rtn)(void *)`: The newly created thread runs `start_rtn` function.
- `void *restrict arg`: passed as single argument to `start_rtn()` function (typeless pointer). If you need to pass more than one argument, store them in a struct and pass the address of the struct to `arg` (casting to `void *`)

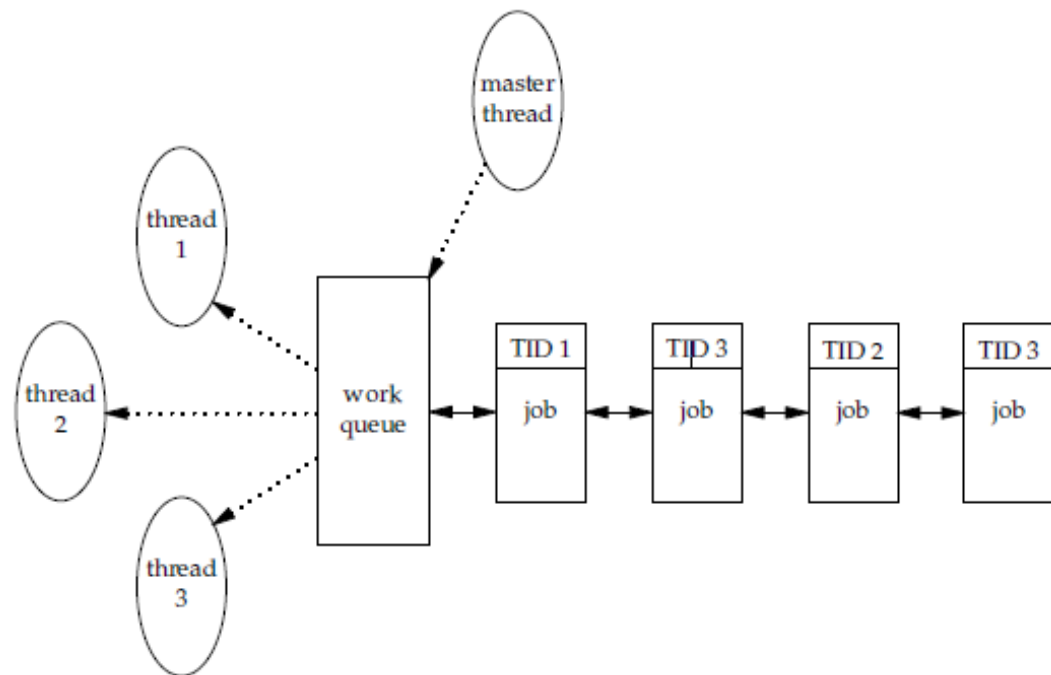


# PTHREAD\_CREATE

When a thread is created, there is no guarantee it will run first

The newly created thread

- has access to the process global address space
- inherits the environment and signal mask (the set of pending signals is cleared)



# PTHREAD TERMINATE

A pthread can exit in three ways.

1. The thread can return from the start routine `start_rtn`. The return value is the thread's exit code
2. The thread can be canceled by another thread in the same process

```
#include <pthread.h>
int pthread_cancel(pthread_t tidp);
Returns: 0 if OK, error number on failure
```

3. The thread can call `pthread_exit`

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
Returns: 0 if OK, error number on failure
```

- `void *rval_ptr`: a typeless pointer (i.e. constant, or pointer to struct), available to the process (or other threads) by calling `pthread_join`

# PTHREAD TERMINATE

The `pthread_join()` function waits for the thread specified by `thread`, to terminate.

```
#include <pthread.h>
int pthread_join(pthread_t tidp, void **rval_ptr);
           Returns: 0 if OK, error number on failure
```

# PTHREAD CREATE & TERMINATE – EXAMPLE 1

```
...
void *addtoX(void *);
int main() {
    pthread_t tid1, tid2, tid3;
    int x[3]={11, 12, 13};

    // create 3 threads
    pthread_create(&tid1, NULL, addToX, (void *) (intptr_t) x[0]);
    pthread_create(&tid2, NULL, addToX, (void *) (intptr_t) x[1]);
    pthread_create(&tid3, NULL, addToX, (void *) (intptr_t) x[2]);

    // wait for all 3 threads to terminate
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);
}
void *addToX(void *arg) {
    int y = (intptr_t) arg; // *((int*) arg);
    printf("Thread argument is :%d\n", y);
    return 0;
}
```

**Compile :** gcc name.c - lpthread

**Run:** ./a.out

**The program gives us:**

Thread argument is :12

Thread argument is :13

Thread argument is :11

# PTHREAD CREATE & TERMINATE

Multiple examples in :

➤ yolinux :

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

## PTHREAD SYNCHRONIZATION (SAME FOR PROCESS)

When multiple threads share memory, we must make sure that each thread has a consistent view of its data

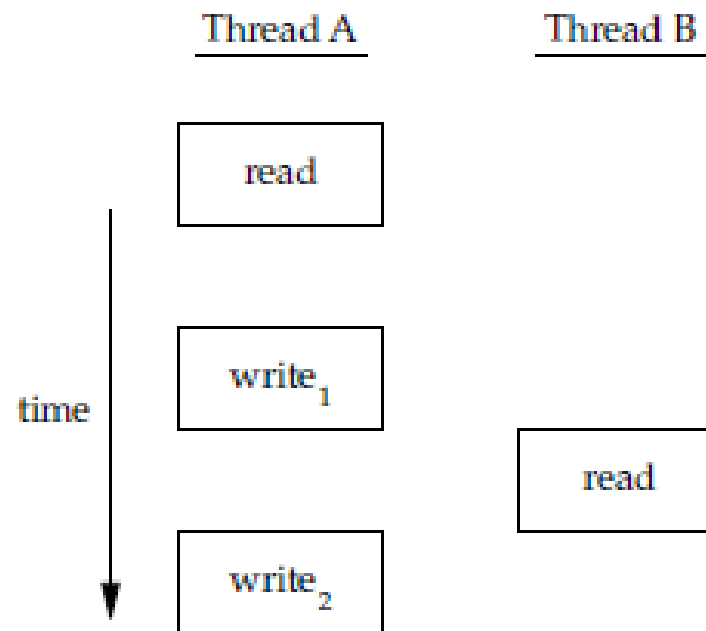
If each thread uses variables that other threads don't read or modify, no consistency problems exist. Similarly, if a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time.

However, when **one thread modifies a variable that other threads want to read or modify, we must synchronize the threads.** This ensures that they don't use an invalid value when accessing the variable.

## EXAMPLE - PTHREADS ACCESS SAME VARIABLE

Let thread A read the variable and then write a new value to it, and assume write operation takes two memory cycles. If thread B reads the same variable between the two write cycles, it will read an inconsistent value.

**Interleaved memory  
cycles with two threads**

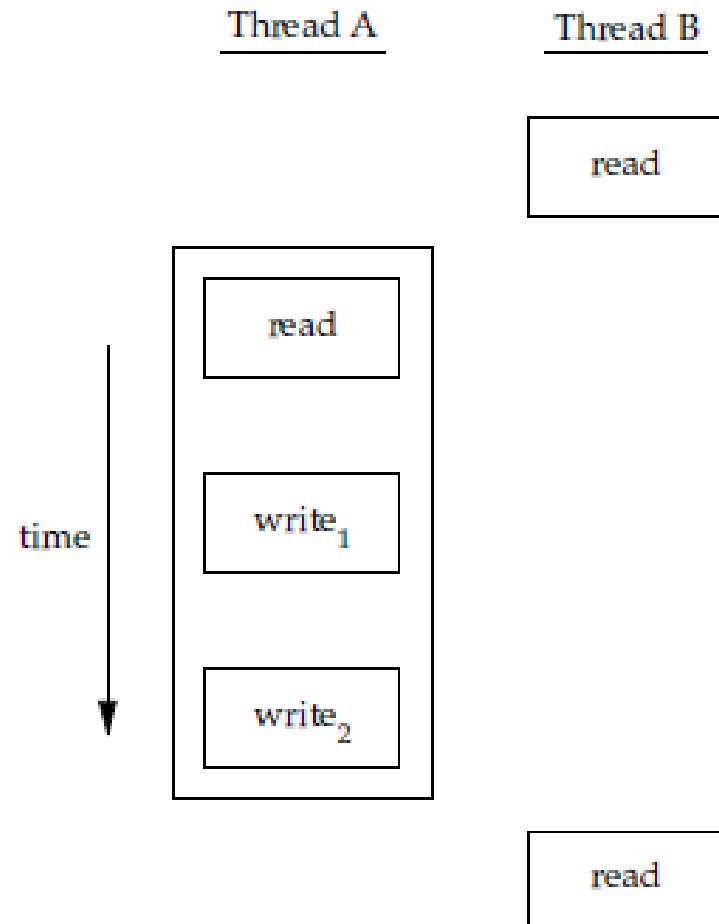


## EXAMPLE – MULTIPLE PTHREADS ACCESS A VARIABLE

To solve the problem with read-modify-write,

- if it's not important which thread changes the variable first, threads use a lock that allows only one thread to access the variable at a time
  - For example, thread A acquires the lock before updating the variable, and thus thread B is unable to read the variable until thread A releases the lock
- If it's important who updates first, we use a barrier to temporally separate accesses (see later)

(Note: this is similar for processes)



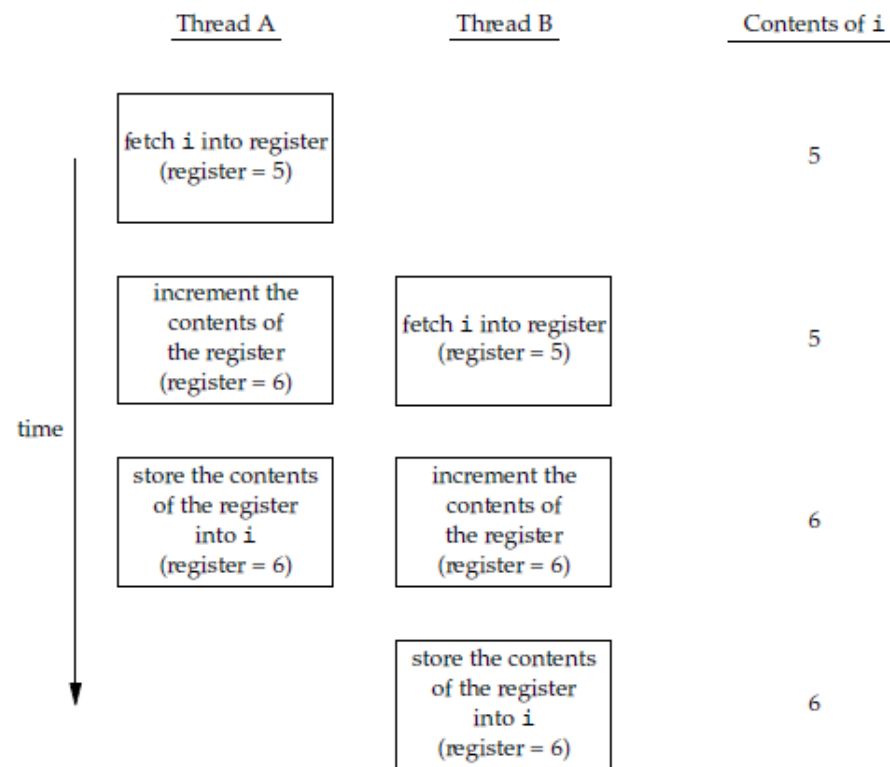


## EXAMPLE – MULTIPLE PTHREADS ACCESS A VARIABLE

We must synchronize **two or more threads that modify the same variable at the same time**

Consider the case in which we increment a variable (next figure). The increment operation is usually broken down into three steps.

1. Read the memory location into a register
2. Increment the value in the register
3. Write the new value back to the memory location



Two unsynchronized threads incrementing the same variable

## MUTEX: LOCK & UNLOCK PRIMITIVES

Lock-based programming protects data in a critical section by ensuring access by one thread at a time!

A mutex is a construct that allows

- **locking a mutex before accessing a shared resource, and**
- **unlocking the mutex when done**

Any thread locking a mutex is **blocked** until the mutex is released

Upon release of the lock, all blocked threads are made runnable, and the first one to run locks the mutex atomically and proceeds, while others block (sleeping)...

## STATIC MUTEX DEFINITION & INITIALIZATION

- A mutex variable is represented by the `pthread_mutex_t` data type
- Before we can use a mutex variable, we first initialize it by setting it to constant `PTHREAD_MUTEX_INITIALIZER` or calling `pthread_mutex_init` function

Note: The second type of initialization must always be used if lock is dynamically allocated

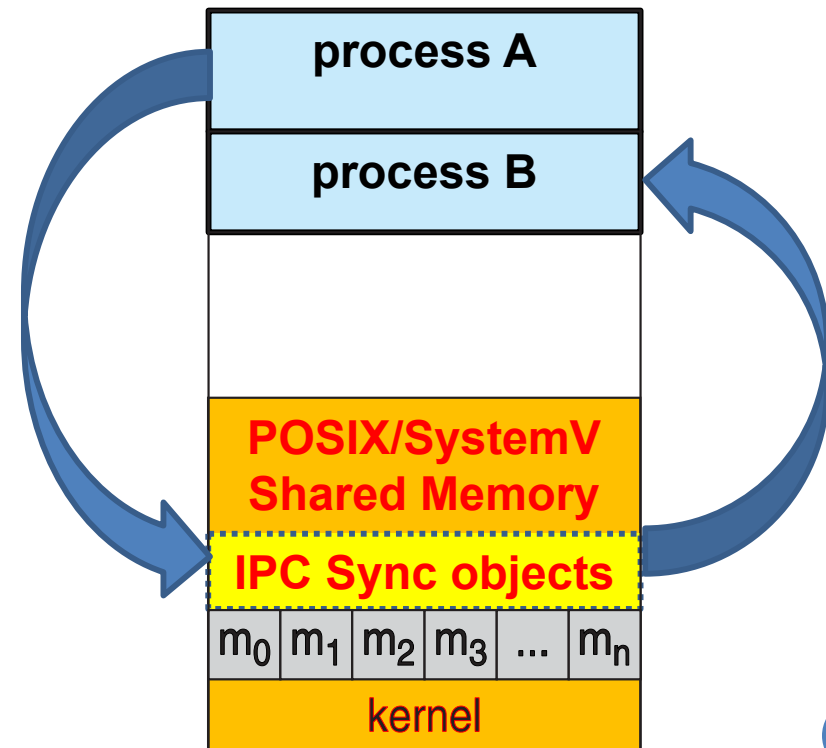
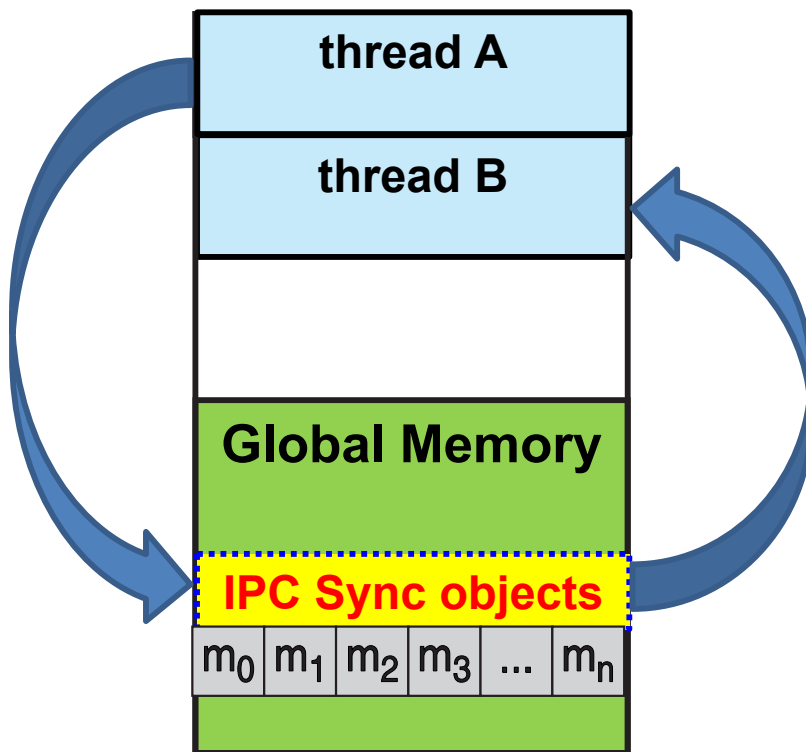
## DYNAMIC MUTEX – MUTEX\_INIT & \_DESTROY

```
#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
```

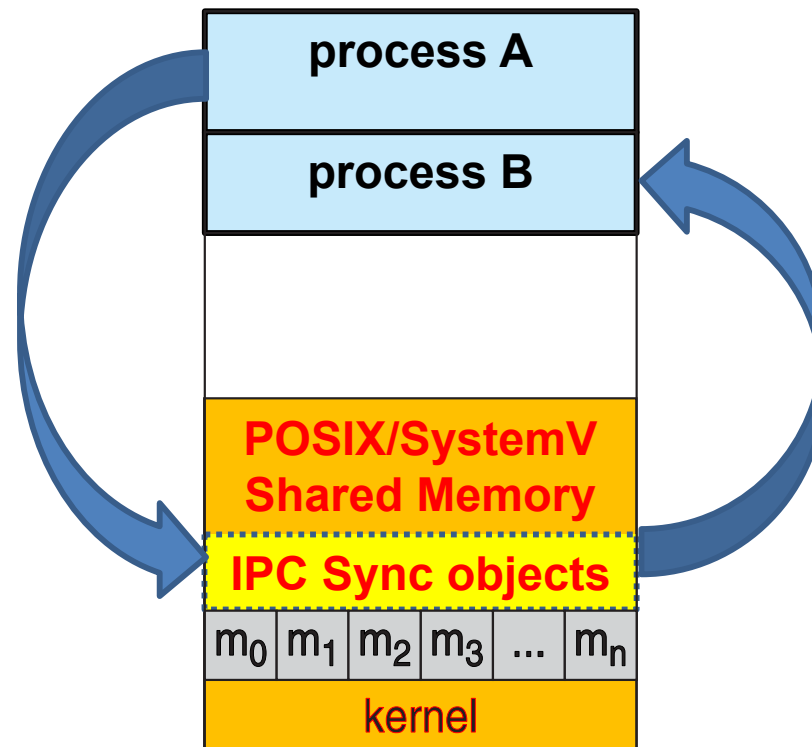
To initialize a mutex with the default attributes, we set `attr` to `NULL`. We discuss mutex attributes (fairness, processes/threads) later

If we allocate the mutex dynamically (e.g., by calling `malloc`), then we need to call `pthread_mutex_destroy` before freeing the memory

# HOW THREADS/PROCESSES SYNC?



# HOW PROCESSES SYNC VIA IPC?



# MUTEX ATTRIBUTES

Different functions to use

- `pthread_mutexattr_init()` : initialize mutex attributes
- `pthread_mutexattr_setpshared()` : set scope of mutex
- `int pthread_mutex_init()` : initialize a mutex object
- ... `pthread_mutex_lock(...)` / `unlock(...)` ...
- `pthread_mutexattr_destroy()` : destroy mutex attributes
- `int pthread_mutex_destroy()` : destroy mutex object
- **free** IPC objects (`pthread_mutexattr_t` & `pthread_mutex_t`)

# PTHREAD\_MUTEXATTR\_INIT & \_DESTROY

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

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Returns: 0 if OK, -1 on error

- *pthread\_mutexattr\_t \*attr*: specifies the mutex attributes

Functions `pthread_mutexattr_init` (and `pthread_mutexattr_destroy`) initialize to default (or deinitialize) the `pthread_mutexattr_t` structure



# MUTEX ATTRIBUTES – SETPShared & GETPShared

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

- *const pthread\_mutexattr\_t \* restrict attr*: specifies an object of *pthread\_mutexattr\_t* type
- *int \*restrict pshared*: set *attr* to
  - PTHREAD\_PROCESS\_SHARED: **sync threads in this and other processes**
  - PTHREAD\_PROCESS\_PRIVATE: **sync threads in this process only (default)**

# DYNAMIC MUTEX – INITIALIZATION & DESTROY

```
#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
```

- *pthread\_mutex\_t \*restrict mutex*: specifies the address of *pthread\_attr\_t* data type
- *const pthread\_mutexattr\_t \*restrict attr*: specifies to initialize a mutex with *attr* attributes (may also be NULL for default attributes). See the book on how to set other mutex attributes, e.g., fairness

If we allocate a mutex dynamically (e.g. by calling `malloc`), then we need to call `pthread_mutex_destroy` to destroy (deinitialize) the object

## MUTEX – LOCK/UNLOCK OPERATIONS

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_unlock(pthread_mutex_t *mutex);
    All return: 0 if OK, error number on failure
```

# TIMING MEASUREMENTS & PIN A PTHREAD TO CPU

Use `clock_gettime` to evaluate total execution time for different X

```
#include <time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

<https://gist.github.com/pfigure/9ce8a2c0b14a2542acd7>

Use `pthread_setaffinity_np` to balance threads across all CPUs. Function **pin\_cpu(N)** from thread function to fix execution to CPU<sub>N</sub>

```
void pin_cpu(int cpu) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu, &cpuset);
    if (pthread_setaffinity_np(pthread_self(), \
        sizeof(cpu_set_t), &cpuset) < 0)
        err(1, "failed to set affinity");
}
```

## DEADLOCK – MULTIPLE MUTEXES

A protocol deadlock may occur in a program with multiple mutexes if the following situation occurs:

- **one thread, holding mutex A, blocks while trying to lock a second mutex B, while**
- **another thread holding mutex B, attempts to lock the first mutex A**

Neither thread can proceed, since each one requires a resource that is held by the other. Notice that this dependency cycle may involve one or more threads



## DEADLOCK AVOIDANCE – CONTROLLING LOCK ORDER

A deadlock can only occur if one thread attempts to lock mutexes in the opposite order from another thread.

Deadlocks can be avoided by carefully **controlling the order in which mutexes are locked**

For example, assume that you have two mutexes A and B, that you need to lock at the same time. If all threads lock mutex A before mutex B, no deadlock can occur (deadlock with other resources is still possible).



## DEADLOCK AVOIDANCE

Sometimes, it is difficult to apply lock order.

Alternatively, one can use `pthread_mutex_trylock` (nonblocking call) to **avoid deadlock**.

If `pthread_mutex_trylock` is successful, then the process can proceed. Otherwise, the process can release the locks held, and try again later.

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
    All return: 0 if OK, error number on failure
```



# BERNSTEIN CONDITIONS - THEORY

Set of conditions sufficient to determine whether two processes can be executed simultaneously. Given:

$I_i$  is the set of memory locations read (input) by process  $P_i$ .

$O_j$  is the set of memory locations written (output) by process  $P_j$ .

For two processes  $P_1$  and  $P_2$  to be executed simultaneously, inputs to process  $P_1$  must not be part of outputs of  $P_2$ , and inputs of  $P_2$  must not be part of outputs of  $P_1$ ; i.e.,

$$I_1 \cap O_2 = \phi$$

$$I_2 \cap O_1 = \phi$$

where  $\phi$  is an empty set. Set of outputs of each process must also be different; i.e.,

$$O_1 \cap O_2 = \phi$$

If the three conditions are all satisfied, the two processes can be executed concurrently.



# BERNSTEIN CONDITIONS – EXAMPLE

Suppose the two statements are (in C)

`a = x + y;`

`b = x + z;`

We have

$$I_1 = (x, y)$$

$$O_1 = (a)$$

$$I_2 = (x, z)$$

$$O_2 = (b)$$

and the conditions

$$I_1 \cap O_2 = \phi$$

$$I_2 \cap O_1 = \phi$$

$$O_1 \cap O_2 = \phi$$

are satisfied. Hence, the statements `a = x + y` and `b = x + z` can be executed simultaneously.

## BERNSTEIN CONDITIONS – LOCK OR BARRIER?

○ P1 :  $x = x + y$  (RMW)

○ P2 :  $x = x + z$  (RMW)

○ P1 :  $x = x + y$

○ P2 :  $y = z + w$

○ P1 :  $x = x + y$

○ P2 :  $y = z + x$



## BARRIER

Barrier is a synchronization mechanism that *allows each thread to wait until all cooperating threads have reached a certain point in their code, and then continue executing from there.*

Notice that `pthread_join` function acts as a barrier to allow one thread to wait until another thread exits.

Barrier allows an arbitrary number of threads to wait until all of the threads have completed processing, but the threads don't have to exit. They can continue working after all threads have reached the barrier



# PTHREAD\_BARRIER\_INIT & \_DESTROY

`pthread_barrier_init` initializes, and `pthread_barrier_destroy` clears a barrier until next initialization

```
#include <pthread.h>

int pthread_barrier_init( pthread_barrier_t *restrict barrier,
                        const pthread_barrierattr_t *restrict attr,
                        unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);

Both return: 0 if OK, error number on failure
```

- `pthread_barrier_t *restrict barrier`: A pthread barrier is represented by the barrier data type `pthread_barrier_t`
- `const pthread_barrierattr_t *restrict attr`: barrier attribute are set to `PTHREAD_PROCESS_SHARED/DYNAMIC` for use with process/thread (default threads)
- `unsigned int count`: number of threads that must reach the barrier before all threads are allowed to continue

# PTHREAD\_BARRIER\_WAIT

We use the `pthread_barrier_wait` function to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up

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

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Returns: 0 or PTHREAD\_BARRIER\_SERIAL\_THREAD if OK, error number on failure



# POSIX NAMED VS. UNNAMED SEMAPHORES

- Named semaphores are used by unrelated processes/threads (e.g., written by different engineers) by passing the same name to `sem_open()`
- Unnamed semaphores (lacking a name) must exist in a pre-existing, agreed upon memory location (shared memory for processes, and global memory or heap for threads of a single process). Thus, code in parent, child, or threads already knows the address of the semaphore.



# POSIX NAMED SEMAPHORE - SEM\_OPEN

A named semaphore is identified by a name `/somename`. The `sem_open()` function **creates and possibly initializes** a new named semaphore or opens an existing one.

```
#include <fcntl.h>      // For O_* constants
#include <sys/stat.h>    // For mode constants
#include <semaphore.h>   // Link with -pthread
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int val);
                Return: 0 if OK, -1 on error
```

- `O_CREAT` specifies that a new semaphore must be created if it does not exist. Its owner/group is set to the effective uid/gid of the calling process. If `O_CREAT` | `O_EXCL` is specified, then an error is returned if a semaphore with the given name already exists
- If the semaphore is created, then
  - mode specifies r/w permissions as in `shm_open()`, see `<sys/stat.h>`.
  - value specifies the initial value for the new semaphore

## POSIX NAMED SEMAPHORES – SEM\_UNLINK

Once all processes close a previously open named semaphore, we can discard it by calling the `sem_unlink` function

```
#include <semaphore.h>
int sem_unlink(const char* name);
```

Returns: 0 if OK, -1 on error



## POSIX UNNAMED SEMAPHORES – SEM\_CLOSE

When we are done using an unnamed semaphore, we can discard it by calling the `sem_close()`. This frees semaphore resources allocated to the calling process.

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

Returns: 0 if OK, -1 on error

# POSIX UNNAMED SEMAPHORES (MEMORY-BASED)

An unnamed semaphore does not have a name.

- Instead, the semaphore is placed in a region of memory that is shared between multiple threads (global variable) or processes (shared memory region), either System V or POSIX shared memory
- Before being used, an unnamed semaphore must be initialized using `sem_init()`
- It can then be operated using `sem_post()` and `sem_wait()`.
- When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be closed using `sem_close()` and destroyed using `sem_destroy()`
- `sem_unlink()` removes a named semaphore when last process stops using it

# POSIX UNNAMED SEMAPHORES

To initialize an unnamed semaphore, we call `sem_init`

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int val);
Returns: 0 if OK, -1 on error
```

`int pshared`: indicates if we plan to use the semaphore with processes, or with threads in the same process (essentially an in-process semaphore). In the former case, we set it to a nonzero value

`unsigned int val`: specifies the initial value of the semaphore



# SEMAPHORE OPERATIONS – NAMED & UNNAMED

- `sem_wait` checks if semaphore is greater than zero, and if so, it decrements it and returns immediately. Otherwise, the function blocks until the semaphore is positive, or a signal interrupt occurs

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

Both return: 0 if OK, -1 on error

- `sem_post` increments the semaphore. As a result, some other process blocked on this semaphore (calling `sem_wait`) is unblocked to continue execution

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

Returns: 0 if OK, -1 on error



# POSIX SEMAPHORES

When we not using the semaphore anymore, we can discard it from virtual address space of the process by calling the `sem_destroy` and destroy it using `sem_unlink`

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
                Returns: 0 if OK, -1 on error
```

```
#include <semaphore.h>
int sem_unlink(const char *name);
                Returns: 0 if OK, -1 on error
```



# CONDITION VARIABLES

- `#include <pthread.h>`
- The `pthread_cond_t` object has two main operations
  - Wait: `pthread_cond_wait(...)`
  - Signal: `pthread_cond_signal(...)`
- Used for event notification
  - Wake up a process when a particular condition occurs
- Implements a monitor along with a mutex



# CONDITION VARIABLES

## ○ Important functions

- `pthread_cond_wait(mutex)` causes the thread to suspend execution until some condition is true
- `pthread_cond_signal(mutex)` signals a condition, hence, one of the threads which have posted previously a wait on this condition variable (if any) is woken up and given access to the mutex
- `pthread_cond_broadcast(mutex)` broadcasts a condition, hence, all threads which have posted previously a wait on this condition variable (if any) are woken up and given access to the mutex

Possible data race: **if one thread signals the condition before another thread actually waits, then the signal is lost**

A condition variable is associated with a user-defined mutex to avoid deadlock during data race



## EXAMPLE

- Waiting for  $x==y$  condition

```
pthread_mutex_lock(&m);  
while (x != y)  
    pthread_cond_wait(&v, &m);  
    /* modify x or y if necessary */  
pthread_mutex_unlock(&m);
```

- Notifying the waiting thread that  $x$  has been incremented

```
pthread_mutex_lock(&m);  
x++;  
pthread_cond_signal(&v);  
pthread_mutex_unlock(&m);
```





# CREATING / DESTROYING CONDITION VARIABLES

## Creating a condition variable

- Static initialization

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Standard Initializer

```
int pthread_cond_init(pthread_cond_t *restrict cond,  
                      const pthread_condattr_t *restrict attr);
```

- Working with processes (default with threads)

```
int pthread_condattr_setpshared(pthread_condattr_t *attr, int  
pshared);
```

- Destroying a condition variable

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Returns 0 if successful, nonzero error code if unsuccessful



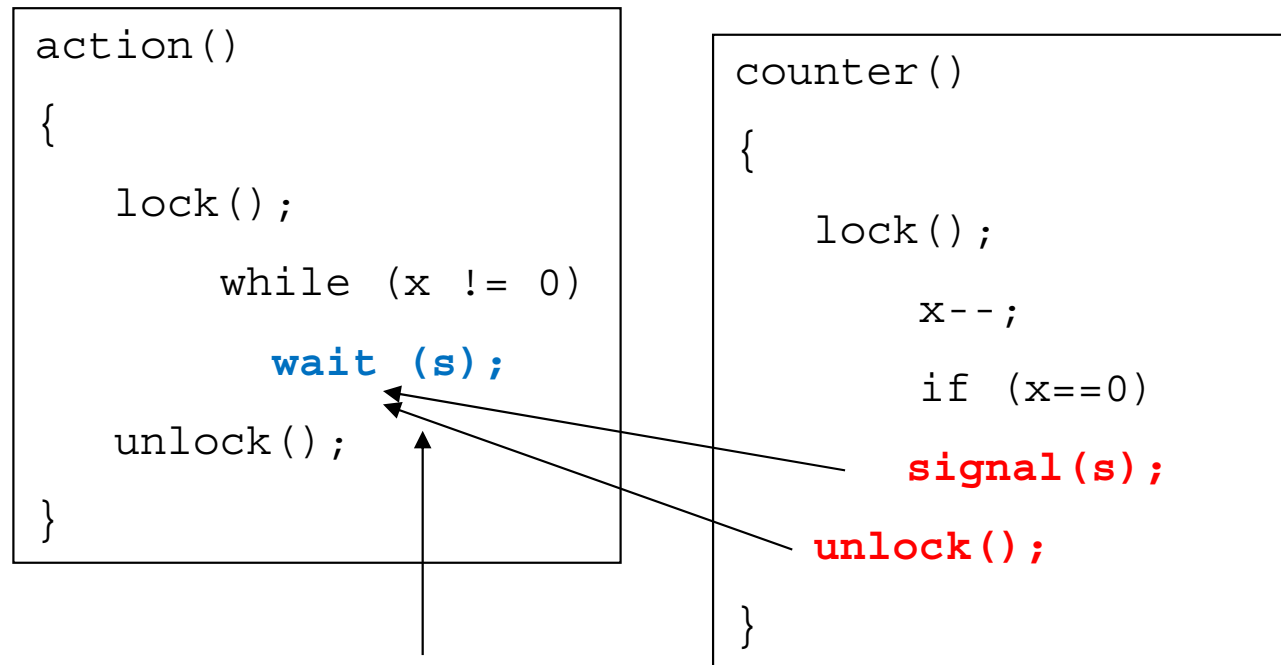
# WAITING ON CONDITION VARIABLES

```
int pthread_cond_wait(pthread_cond_t *restrict cond,  
                      pthread_mutex_t *restrict mutex);
```

- Called with a mutex lock held
- Internals
  - Causes the thread to release the mutex
  - Sleeps until signaled
  - Reacquires the lock when woken up
- Variation: `pthread_cond_timedwait`



# CONDITIONAL WAITING



Both must occur before wait () returns

## EXERCISE – RELATIVE PROGRESS RATE OF THREADS

1. Write a program which creates two threads which enter a loop that prints its id (1 or 2). However, threads must synchronize, so that the first thread is always executed twice before the other. Notice that the only valid sequence of execution is:

1, 1, 2, 1, 1, 2, ...

2. Rewrite your program, the first thread is always executed twice in each round of three trials. Notice that now there are more valid sequences:

1, 1, 2, ... OR

1, 2, 1, ... OR

2, 1, 1, ...

For a given number of runs which program is faster and why?



## EXERCISE – SLEEPING BARBER (N CHAIRS)

Write a program that models barber/customer threads and operations

- A barbershop consists of a waiting room with N chairs and a barber room with one barber chair
- If there are no customers to be served  $\Rightarrow$  barber goes to **rest**
- If a customer enters and all chairs are occupied  $\Rightarrow$  customer **leaves**
- If the barber is busy but chairs are available  $\Rightarrow$  customer **sits** in one of the free chairs
- If the barber is asleep  $\Rightarrow$  customer wakes the barber to **have haircut**



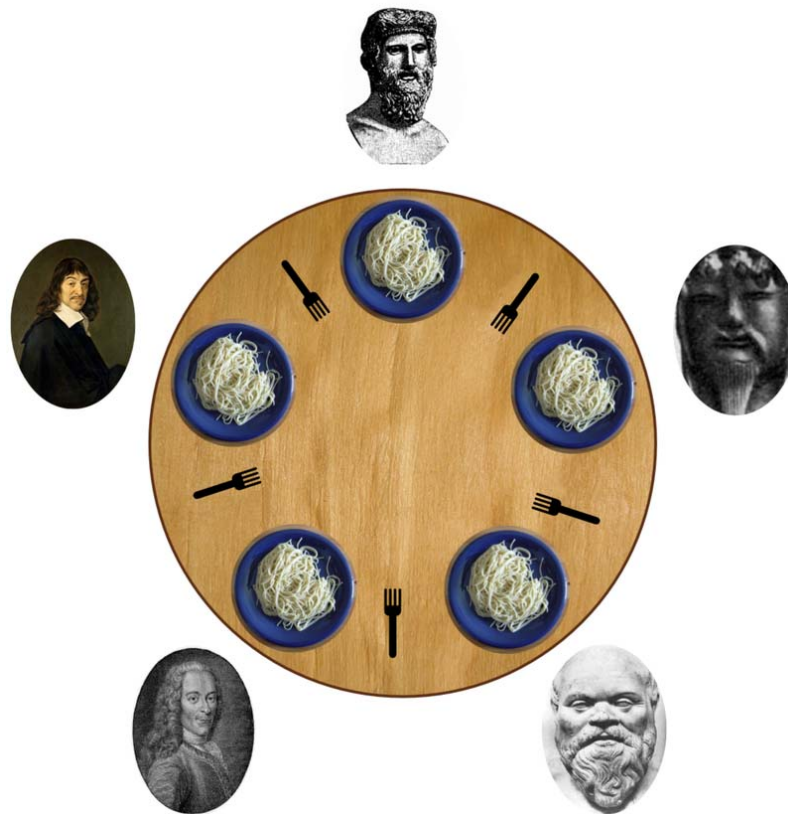
## EXERCISE – M SLEEPING BARBERS (N CHAIRS)

Write a program that models barber/customer threads and operations

- A barbershop consists of a waiting room with N chairs and a barber room with M barber chairs
- If there are no customers to be served  $\Rightarrow$  all barbers go to **rest**
- If a customer enters and all chairs are occupied  $\Rightarrow$  customer **leaves**
- If all barbers are busy but chairs are available  $\Rightarrow$  customer **sits** in one of the free chairs
- If all barbers are asleep  $\Rightarrow$  customer wakes a barber to **have haircut**



# EXERCISE - DINING PHILOSOPHERS PROBLEM



[https://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](https://en.wikipedia.org/wiki/Dining_philosophers_problem)