



# **POSIX THREADS & MUTEX/BARRIER/SEMAPHORES**

1



# POSIX THREADS & MUTEX

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

**Sync with Mutex:** `pthread_mutex_init/lock/unlock`

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

# BOOK

## ◦ Read List

- POSIX thread creation, termination and synchronization via mutex in Chapter 11.1 to 11.6.1 (pages 383-401) of main book
- Thread attributes (e.g., `pthread_attr_init/destroy`) in Chapter 12.3 (pages 426-427) of main book
- Mutex attributes (`pthread_mutex_attr`) in Chapter 12.4.1 (pages 430 - 431) of main book

# INTRODUCTION

A thread defines a single execution stream within a process

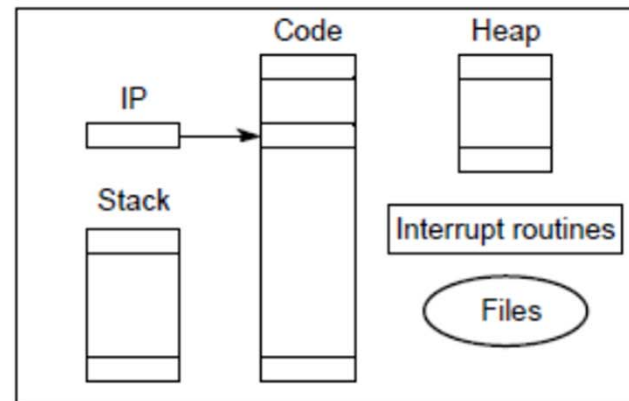
In this chapter, we examine

- 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

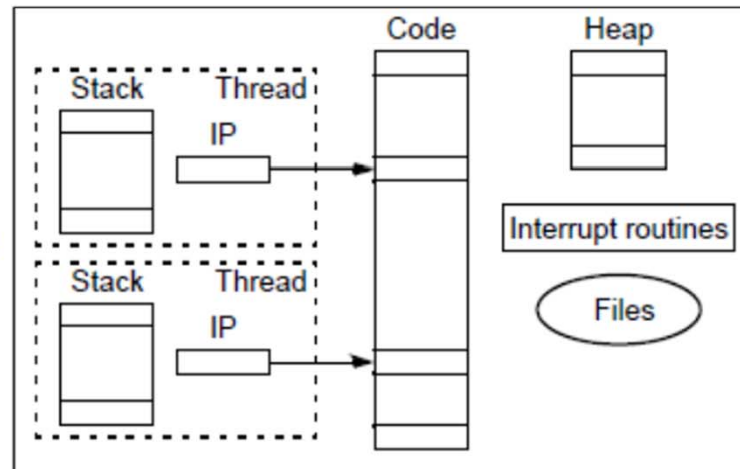
“heavyweight” process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



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

(b) Threads



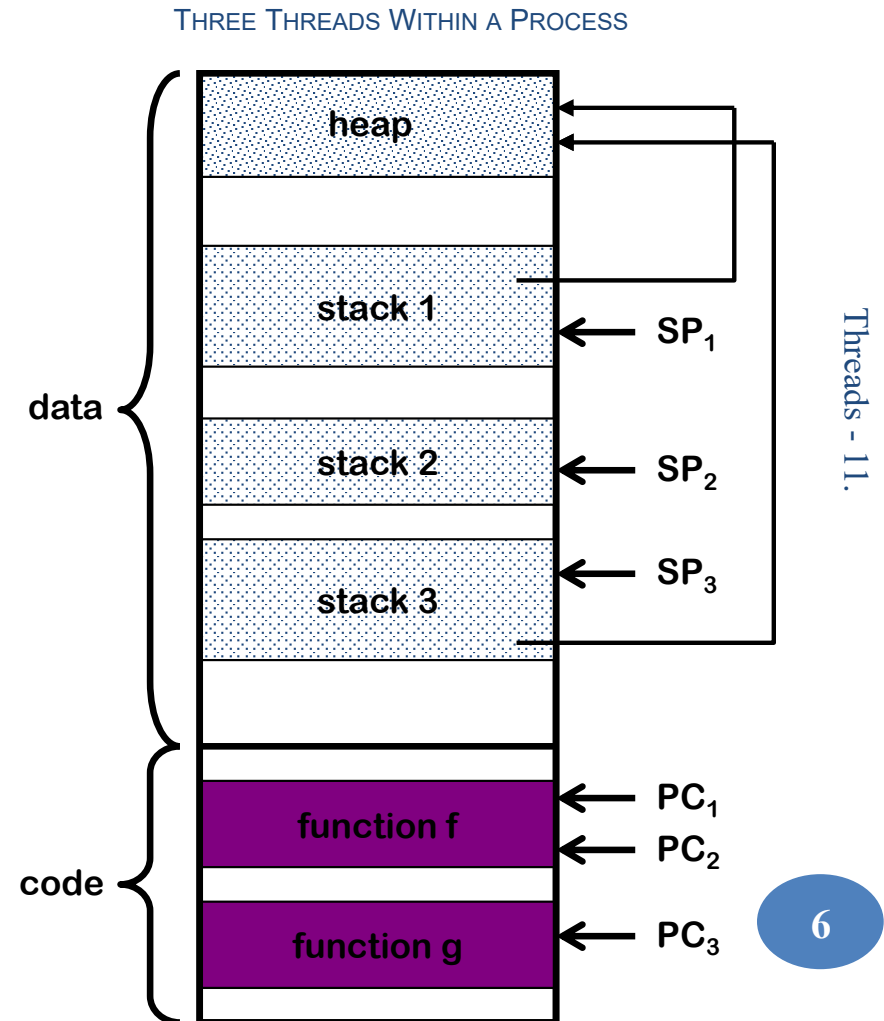
# THREADS VS PROCESSES

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

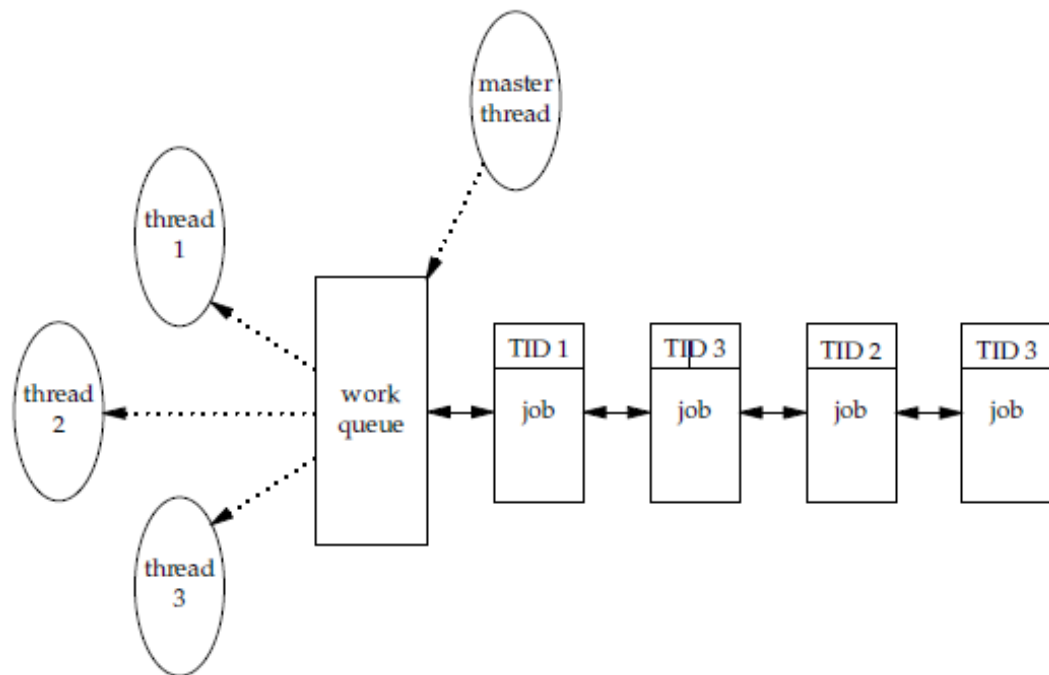
- `pthread_t *restrict tidp`: Thread ID pointer of the newly created POSIX thread, when `pthread_create` returns successfully
- `const pthread_attr_t *restrict attr`: used to customize thread attributes: concurrency level, scheduling policy/parameters, detach state, stack address/size, stack guard size. 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 :

- Book (Stevens & Rago): pages 389 -390
- 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.

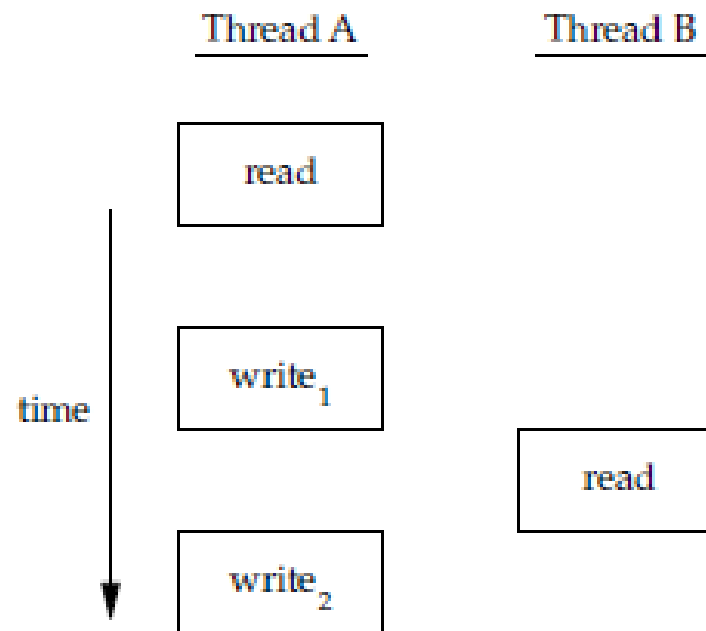
However, when **one thread modifies a variable that other threads want to read or modify, we must synchronize the threads.** This ensures that they don't use an invalid value when accessing the variable.



## 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**

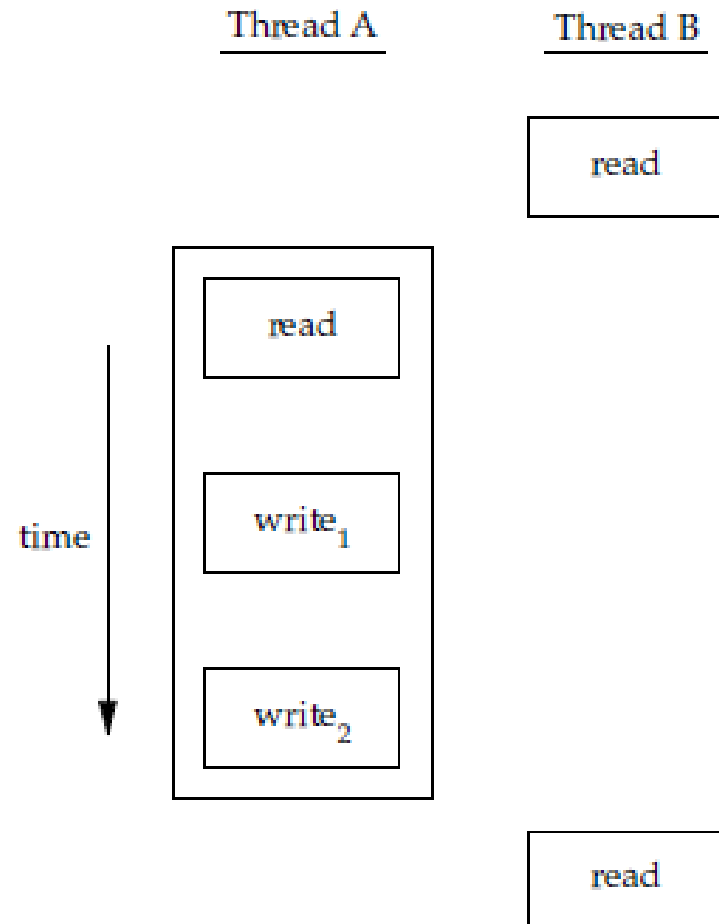


## EXAMPLE – MULTIPLE PTHREADS ACCESS A VARIABLE

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)

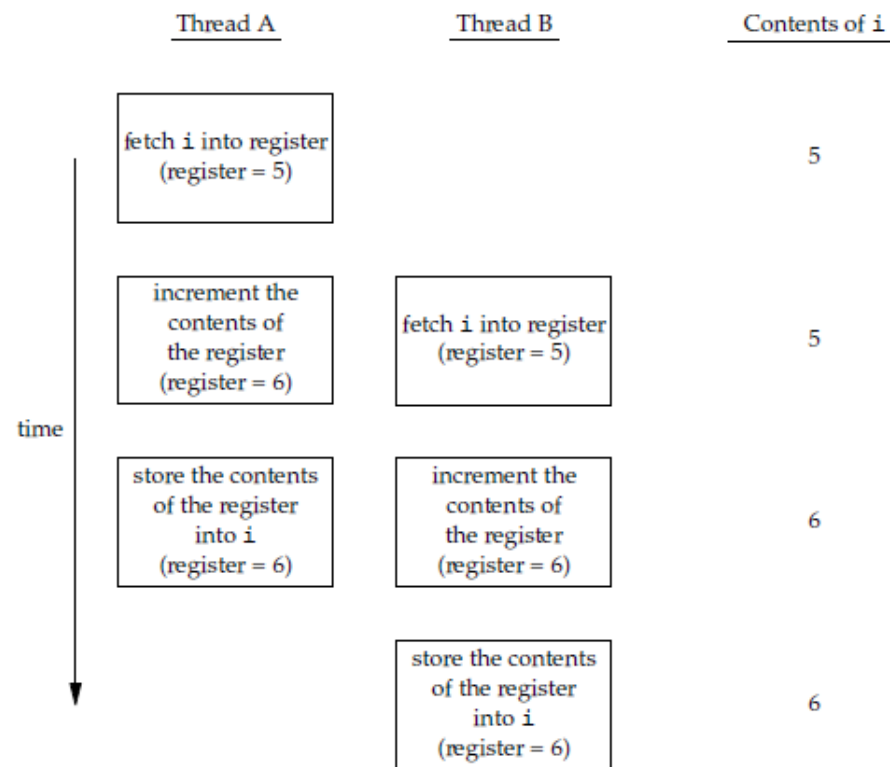


## EXAMPLE – MULTIPLE PTHREADS ACCESS A VARIABLE

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

## 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
```

## EXERCISE 2 - SOLVE DATA RACE (MUTEX)

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

pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;
int x = 0; // global share variable

void *addtoX(void *arg) {
    int i;
    int y = (intptr_t) arg;
    printf("Before sum :i Thread argument is :%d\n", y);
    for(i = 0; i < 10000; i++ ) {
        pthread_yield(); // yield to other threads
        pthread_mutex_lock(&L);
        x=x+1;
        pthread_mutex_unlock(&L);
    }
    printf("After sum :Thread argument is :%d\n", y);
    return 0; }
```



## EXERCISE 2 - SOLVE DATA RACE (MUTEX)

```
int main() {
    pthread_t tid[3];
    int i;

    // threads creation
    for (i=0; i<3; i++)
        pthread_create(&tid[i], NULL, addtoX, (void *) (intptr_t) i);

    // wait for all three threads to terminate
    for (i=0; i<3; i++)
        pthread_join(tid[i], NULL);

    printf("x:%d \n", x);
    fflush(stdout);

    return 0;
}
```

## EXERCISE 2 - SOLVE DATA RACE (MUTEX)

1. Download program\_name below from ECLASS folder:  
`06b_simple_pthreads_with_data_race.c`
2. Compile and run the program on the server or locally  
(in your Virtualbox VM)  

```
gcc name -lpthread  
./a.out
```
3. Read comments at the top and understand the code

## EXERCISE 2

1. Change the for loop to 10000
2. Uncomment `pthread_yield` function
3. Compile and run the program again

```
gcc name -lpthread  
./a.out
```

What do you notice if you run several time with/without lock?

What is the effect of `pthread_yield`?

## EXERCISE 3 – UPLOAD TO ECLASS

1. Use X threads to compute **matrix vector product**  $\mathbf{B}_{n \times 1} = \mathbf{A}_{m \times n} \times \mathbf{X}_{n \times 1}$

$$\mathbf{B}_{n \times 1} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

2. 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>

3. Use `pthread_setaffinity_np` to balance threads across all CPUs

## EXERCISE 3 – HELP: PIN A PTHREAD TO CPU

Call **pin\_cpu(0)** from thread function to fix execution to CPU 0

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

## EXERCISE 4

1. Write lock-based multithreaded code with X threads to compute the **dot product** of vectors a[n] & b[n] into c (integer)

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

2. Use `clock_gettime` to evaluate speedup for different X  
`#include <time.h>`  
`int clock_gettime(clockid_t clk_id, struct timespec *tp);`  
<https://gist.github.com/pfigure/9ce8a2c0b14a2542acd7>
3. Use `pthread_setaffinity_np` to balance threads across all available CPUs

## EXERCISE 5 – MATRIX VECTOR MULTIPLICATION

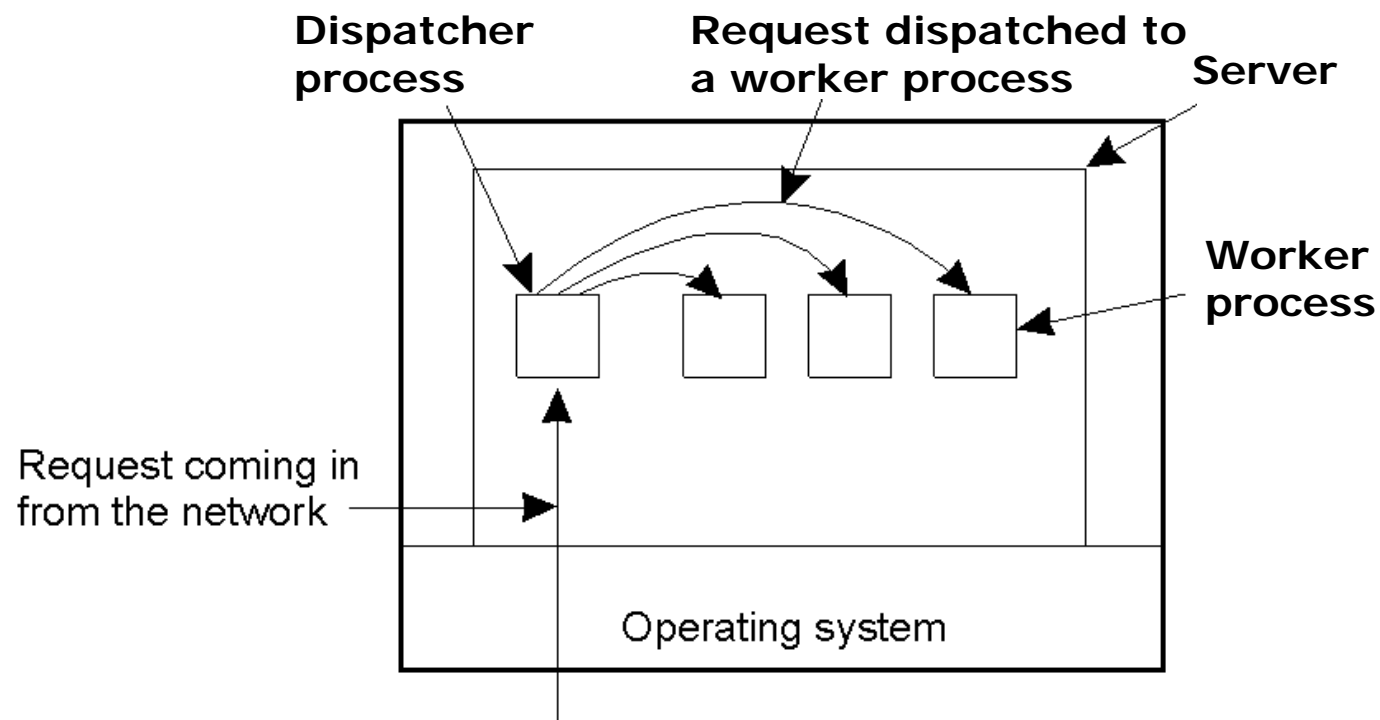
1. Write multithreaded code with  $X$  threads to perform matrix vector multiplication, i.e.  $x = A \times B$  (where  $A$ ,  $B$ ,  $C$  are matrices of sizes  $n \times n$ ,  $n \times 1$  and  $n \times 1$ )

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & & & \vdots \\ \vdots & & \ddots & \\ a_{n,1} & \cdots & & a_{n,n} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \Rightarrow x_i = \sum_{j=1}^n a_{i,j} \times b_j$$

2. Do you need locks to solve this problem?



# EXAMPLE: CONCURRENT SERVERS (TCP/UDP)





## MULTIPROCESS ARCHITECTURE – CHROME BROWSER



- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

## RELATED FUNCTIONS

```
#include <unistd.h>
long sysconf(int name);
```

See page 42 (2.5.4)

```
#include <sys/times.h>
clock_t times(struct tms *buf );
```

See page 280 (8.17)

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

See pages 426-427 (12.3)

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

See pages 430 - 4317 (12.4.1)



# **THREAD/PROCESS SYNCHRONIZATION**

**Deadlock Avoidance**  
**Barriers**

# BOOK

## ○ Read List

- Deadlock avoidance in Chapter 11.6.1 (pages 402 – 407) of main book
- Barriers in Chapter 11.6.8 (pages 418 – 419) of main book



# 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 :  $y = z + x$



## 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
```



## EXAMPLE – RUN – EXERCISE 1

1. Download from ECLASS folder:  
`08a_pthread_2locks_deadlock_avoidance.c`
2. Compile and run the program on the server or locally  
(in your Virtualbox VM)  

```
gcc name -lpthread  
./a.out
```
3. Read comments at the top and understand the code



## EXERCISE 2

1. Extend `08a_pthread_2locks_deadlock_avoidance.c` to deadlock **3 threads that need to acquire successively 3 different locks**
2. In order to complete a round and go past the critical section, a racer must hold all three locks
3. Are there other possibilities of deadlocks in your code? If so, how many?



## 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



## EXAMPLE – RUN – EXERCISE 3

1. Download from ECLASS folder:  
`08b_simple_barrier_before_sum_4threads.c`
2. Compile and run the program on the server or locally  
(in your Virtualbox VM)  
(see the file README.txt)
3. Try to understand the code, in the part where  
`pthread_barrier_wait` calls
4. Why are these necessary?





## EXERCISE 4

Write a new multithreaded program, where each of the  $M$  threads

1. generates  $N$  (possibly random) numbers, and then
2. adds those  $N$  numbers to a global variable (initial value 0)

Addition to the global variable is performed either

1. using a lock (as in previous classes), or
2. using alternating barrier calls, so that threads enter the critical section in round-robin fashion ( $T_1, T_2, \dots, T_M, T_1, T_2, \dots, T_M, \dots$ ). Each time a thread enters the critical section, it adds its next  $X$  numbers

To validate correctness, each thread prints the value of the global variable before exiting the critical section. Remember to flush the buffer after each print using `fflush(stdout)`.

Time your implementations using `clock_gettime`, as in previous Labs, and include your comparisons (lock vs barrier for various values of  $M$ , and  $X$ ) as comments in your program.

Collaborate with others, but then sit down & write your solution alone!

# EXERCISE 5

Write a multithreaded elevator controller for an M-story building. the elevator and N passengers

1. Thread  $T_0$  models the elevator. The elevator struct includes `current_time`, `status={moving, door_closed, door_open}`, and `current_floor`. The elevator can carry ONLY one person at a time identified by `id`, `original_floor`, and `target_floor`. When the elevator arrives at `target_floor` with `door_open`, it waits until passenger calls `exit_elevator`.
2. N threads ( $T_1, \dots, T_N$ ) model N passengers (`id = 1, \dots, 50`). Each passenger performs a random delay (few secs), and then presses a button (only once) to set `original_floor`, and `target_floor`. If a passenger is selected, he waits until elevator arrives at `original_floor` with `door_open`. Then, elevator waits until a passenger can print `enter_elevator`.

Assumptions:

1. Initially the elevator is in floor 0.
2. If there is no pending passenger, the elevator waits on the last `target_floor` (initially floor 0)
3. Show that your code properly controls the elevator by correctly printing the status of elevator and passenger, e.g. using `door_open`, `enter_elevator/exit_elevator` and `door_closed`.
  1. Try to use specific messages to help validation, i.e., by providing `current_floor`, `passenger id` etc
  2. Remember to also flush the output buffer using `fflush(stdout)` after each print.
4. The elevator stops after all passengers have been served

## EXERCISE 6

1. Write a program where  $X$  threads ( $T_0, T_1, \dots, T_X$ ) implement square matrix multiplication, i.e.  $C = A \times B$ , where  $A, B, C$  are matrices of size  $n \times n$

```
for  $i = 1$  to  $n$ 
  for  $j = 1$  to  $n$ 
     $c_{ij} = 0$ 
    for  $k = 1$  to  $n$ 
```

Assume

$$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$$

1.  $A, B$  arrays are loaded by main with random numbers before thread creation
2. Each thread computes some rows of  $C$ , e.g.  $T_0$  computes rows:  $0, 0+X, 0+2 \cdot X \dots$
3. All elements in a row of  $C$  array are computed by the same thread
4. Use barriers to print the array **after a row computation is completed by each thread**
5. Main program must compute the total delay of matrix multiplication (without array initialization). This requires synchronizing main with the start and end of the threads.

# EXERCISE 6 – SAMPLE OUTPUT (N = 15, NPROC=5)

```
A 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3
6 0 6 2 6 1 8 7 9 2 0 2 3 7 5
9 2 2 8 9 7 3 6 1 2 9 3 1 9 4
7 8 4 5 0 3 6 1 0 6 3 2 0 6 1
5 5 4 7 6 5 6 9 3 7 4 5 2 5 4
7 4 4 3 0 7 8 6 8 8 4 3 1 4 9
2 0 6 8 9 2 6 6 4 9 5 0 4 8 7
1 7 2 7 2 2 6 1 0 6 1 5 9 4 9
0 9 1 7 7 1 1 5 9 7 7 6 7 3 6
5 6 3 9 4 8 1 2 9 3 9 0 8 8 5
0 9 6 3 8 5 6 1 1 5 9 8 4 8 1
0 3 0 4 4 4 4 7 6 3 1 7 5 9 6
2 1 7 8 5 7 4 1 8 5 9 7 5 3 8
8 3 1 8 9 6 4 3 3 3 8 6 0 4 8
8 8 9 7 7 6 4 3 0 3 0 9 2 5 4

B 0 5 9 4 6 9 2 2 4 7 7 5 4 8 1
2 8 9 3 6 8 0 2 1 0 5 1 1 0 8
5 0 6 4 6 2 5 8 6 2 8 4 7 2 4
0 6 2 9 9 0 8 1 3 1 1 0 3 4 0
3 9 1 9 6 9 3 3 8 0 5 6 6 4 0
0 4 6 2 6 7 5 6 9 8 7 2 8 2 9
9 6 0 2 7 6 1 3 2 1 5 9 9 1 4
9 1 0 7 5 8 7 0 4 8 0 4 2 9 6
1 0 4 2 2 2 0 5 5 2 9 0 2 8 3
8 0 4 0 9 1 9 6 2 5 4 4 9 9 3
6 0 5 0 2 9 4 3 5 1 7 4 3 1 4
6 9 4 2 2 6 4 1 2 8 8 9 2 8 8
8 6 8 3 8 3 3 3 8 0 4 7 6 8 9
0 6 8 7 9 0 3 3 3 7 3 2 6 5 2
6 5 8 7 9 6 0 4 1 0 4 8 7 0 8

C 217 310 340 282 394 300 207 239 265 246 386 256 320 288 296
274 258 278 302 395 302 194 219 270 221 327 293 337 330 243
243 349 359 367 466 427 295 217 338 290 357 302 366 327 272
183 237 290 191 351 251 194 172 168 191 260 197 269 208 210
336 333 332 338 484 399 318 236 305 288 356 327 383 369 330
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3
6 0 6 2 6 1 8 7 9 2 0 2 3 7 5
9 2 2 8 9 7 3 6 1 2 9 3 1 9 4
7 8 4 5 0 3 6 1 0 6 3 2 0 6 1
5 5 4 7 6 5 6 9 3 7 4 5 2 5 4
7 4 4 3 0 7 8 6 8 8 4 3 1 4 9
2 0 6 8 9 2 6 6 4 9 5 0 4 8 7
1 7 2 7 2 2 6 1 0 6 1 5 9 4 9
0 9 1 7 7 1 1 5 9 7 7 6 7 3 6
5 6 3 9 4 8 1 2 9 3 9 0 8 8 5
0 9 6 3 8 5 6 1 1 5 9 8 4 8 1
0 3 0 4 4 4 4 7 6 3 1 7 5 9 6
2 1 7 8 5 7 4 1 8 5 9 7 5 3 8
8 3 1 8 9 6 4 3 3 3 8 6 0 4 8
8 8 9 7 7 6 4 3 0 3 0 9 2 5 4

B 0 5 9 4 6 9 2 2 4 7 7 5 4 8 1
2 8 9 3 6 8 0 2 1 0 5 1 1 0 8
5 0 6 4 6 2 5 8 6 2 8 4 7 2 4
0 6 2 9 9 0 8 1 3 1 1 0 3 4 0
3 9 1 9 6 9 3 3 8 0 5 6 6 4 0
0 4 6 2 6 7 5 6 9 8 7 2 8 2 9
9 6 0 2 7 6 1 3 2 1 5 9 9 1 4
9 1 0 7 5 8 7 0 4 8 0 4 2 9 6
1 0 4 2 2 2 0 5 5 2 9 0 2 8 3
8 0 4 0 9 1 9 6 2 5 4 4 9 9 3
6 0 5 0 2 9 4 3 5 1 7 4 3 1 4
6 9 4 2 2 6 4 1 2 8 8 9 2 8 8
8 6 8 3 8 3 3 3 8 0 4 7 6 8 9
0 6 8 7 9 0 3 3 3 7 3 2 6 5 2
6 5 8 7 9 6 0 4 1 0 4 8 7 0 8

C 217 310 340 282 394 300 207 239 265 246 386 256 320 288 296
274 258 278 302 395 302 194 219 270 221 327 293 337 330 243
243 349 359 367 466 427 295 217 338 290 357 302 366 327 272
183 237 290 191 351 251 194 172 168 191 260 197 269 208 210
336 333 332 338 484 399 318 236 305 288 356 327 383 369 330
330 269 379 271 472 383 258 277 279 284 401 327 408 344 363
345 296 320 368 524 325 320 270 320 218 334 326 435 350 271
303 334 337 265 449 271 214 194 214 147 274 310 338 254 336
333 336 365 310 447 371 268 231 300 187 374 299 322 367 360
247 324 448 332 498 369 275 276 374 229 411 260 378 345 353
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3
6 0 6 2 6 1 8 7 9 2 0 2 3 7 5
9 2 2 8 9 7 3 6 1 2 9 3 1 9 4
7 8 4 5 0 3 6 1 0 6 3 2 0 6 1
5 5 4 7 6 5 6 9 3 7 4 5 2 5 4
7 4 4 3 0 7 8 6 8 8 4 3 1 4 9
2 0 6 8 9 2 6 6 4 9 5 0 4 8 7
1 7 2 7 2 2 6 1 0 6 1 5 9 4 9
0 9 1 7 7 1 1 5 9 7 7 6 7 3 6
5 6 3 9 4 8 1 2 9 3 9 0 8 8 5
0 9 6 3 8 5 6 1 1 5 9 8 4 8 1
0 3 0 4 4 4 4 7 6 3 1 7 5 9 6
2 1 7 8 5 7 4 1 8 5 9 7 5 3 8
8 3 1 8 9 6 4 3 3 3 8 6 0 4 8
8 8 9 7 7 6 4 3 0 3 0 9 2 5 4

B 0 5 9 4 6 9 2 2 4 7 7 5 4 8 1
2 8 9 3 6 8 0 2 1 0 5 1 1 0 8
5 0 6 4 6 2 5 8 6 2 8 4 7 2 4
0 6 2 9 9 0 8 1 3 1 1 0 3 4 0
3 9 1 9 6 9 3 3 8 0 5 6 6 4 0
0 4 6 2 6 7 5 6 9 8 7 2 8 2 9
9 6 0 2 7 6 1 3 2 1 5 9 9 1 4
9 1 0 7 5 8 7 0 4 8 0 4 2 9 6
1 0 4 2 2 2 0 5 5 2 9 0 2 8 3
8 0 4 0 9 1 9 6 2 5 4 4 9 9 3
6 0 5 0 2 9 4 3 5 1 7 4 3 1 4
6 9 4 2 2 6 4 1 2 8 8 9 2 8 8
8 6 8 3 8 3 3 3 8 0 4 7 6 8 9
0 6 8 7 9 0 3 3 3 7 3 2 6 5 2
6 5 8 7 9 6 0 4 1 0 4 8 7 0 8

C 217 310 340 282 394 300 207 239 265 246 386 256 320 288 296
274 258 278 302 395 302 194 219 270 221 327 293 337 330 243
243 349 359 367 466 427 295 217 338 290 357 302 366 327 272
183 237 290 191 351 251 194 172 168 191 260 197 269 208 210
336 333 332 338 484 399 318 236 305 288 356 327 383 369 330
330 269 379 271 472 383 258 277 279 284 401 327 408 344 363
345 296 320 368 524 325 320 270 320 218 334 326 435 350 271
303 334 337 265 449 271 214 194 214 147 274 310 338 254 336
333 336 365 310 447 371 268 231 300 187 374 299 322 367 360
247 324 448 332 498 369 275 276 374 229 411 260 378 345 353
316 368 366 272 436 389 265 251 312 225 401 329 373 279 349
271 308 292 292 395 285 218 182 247 243 287 279 303 324 316
329 315 385 307 469 373 297 299 356 231 449 347 412 338 369
260 362 350 341 446 434 262 220 304 247 381 331 359 304 294
263 403 391 346 476 394 277 244 304 279 395 335 371 321 337
```



## EXERCISE 7

1. Work on the Nbody problem.
2. Request for the demo: `08b_nbody.zip`
2. Compile and run the program on the server or locally (in your Virtualbox VM)  
(see the file README.txt)
3. Try to understand the code, in the part where `pthread_barrier_wait` calls
4. Why are these necessary?



## EXERCISE 8 – EXTRA WORK

See example in:

Book (Stevens & Rago): page 419





# SEMAPHORE SYNCHRONIZATION

Semaphores – `sem_open`, `sem_close`, `sem_unlink` (named)  
`sem_init`, `sem_close`, `sem_destroy` (unnamed)  
`sem_wait`, `sem_post` (all)

# BOOK FOR READING

## Read List

- Chapters for reading:

- 11.6.1 (pages 579 – 584) Posix Semaphores

- `sem_open/sem_destroy/sem_close, sem_init,`  
`sem_wait, sem_post`





# 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 shared, 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 /somenam. The `sem_open()` function **creates and 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()`



# 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 unnamed semaphore anymore, we can discard it from virtual address space of the process by calling the `sem_destroy` function

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

Returns: 0 if OK, -1 on error



## EXERCISE 1

1. Download (producer consumer problem)

`09a_pthread_sem_prod_cons.c`

2. Read the comments at the top of file and understand the code

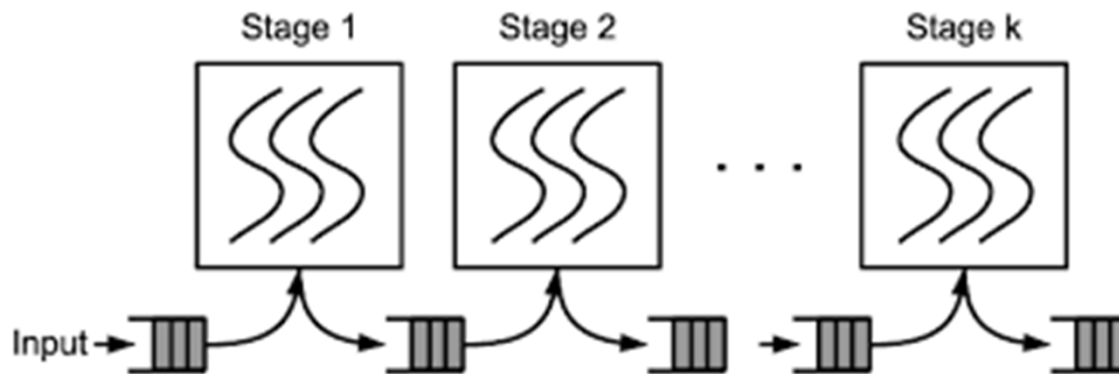
3. Compile and run the program

`gcc name -lpthread`

`./a.out`



## EXERCISE 2



1. Write a program for a 2-stage pipelined consumer/producer
2. First stage consumers are also producers for next stage
3. Use appropriate semaphores to extend the classical solution
4. Assume that all producers/consumers are POSIX threads
5. What if producers/consumers are related (forked), or unrelated processes?
  - Provide at least your comments related to such implementations

## EXERCISE 3 – 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 4 – 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**



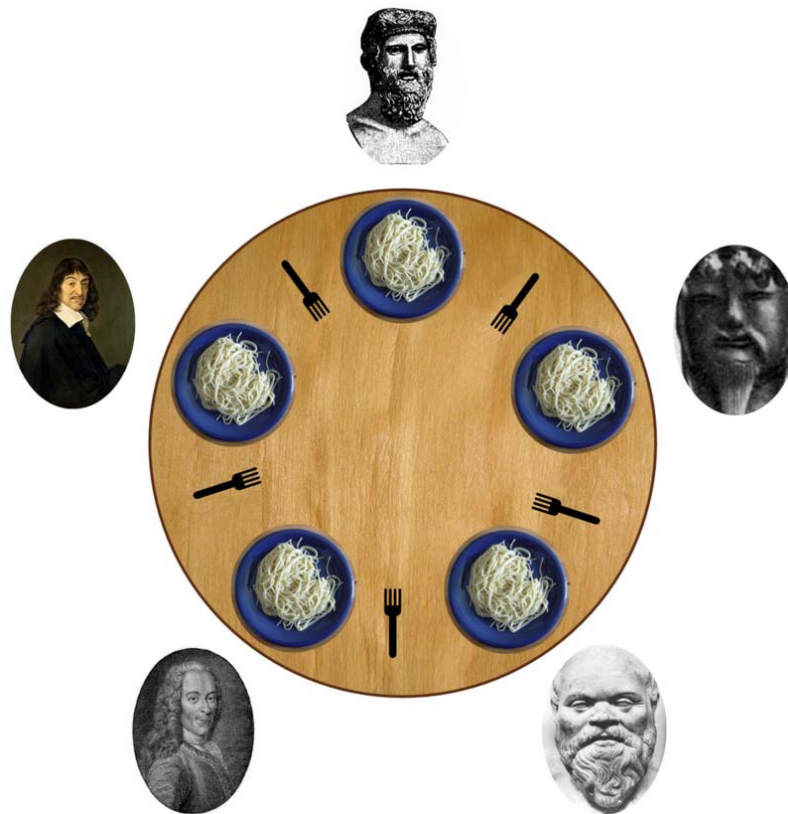
## EXERCISE 5 – M SLEEPING BARBERS (N CHAIRS)

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 6 - DINING PHILOSOPHERS PROBLEM



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



# CONDITION VARIABLES

`pthread_cond_init`

`pthread_cond_wait, pthread_cond_signal`

`pthread_cond_destroy`

# BOOK FOR READING

Book : W. Richard Stevens and Stephan A. Rago, "Advanced Programming, Addison Wesley, 2014, 3rd edition

- 11.6.6 (pages 413-416) Condition Variables



# CONDITION VARIABLES

- `#include <pthread.h>`
- The `pthread_cond_t` object has two main operations
  - Wait: `pthread_cond_wait(...)`
  - Signal: `pthread_cond_signal(...)` or `_bcast(...)`
- 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);
```

## ○ 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
  - Sleeps until signaled
  - Reacquires the lock when woken up
  - Causes the thread to release the mutex
- Variation: `pthread_cond_timedwait`



# SIGNALING CONDITION VARIABLES

```
int pthread_cond_signal(pthread_cond_t *restrict cond);
```

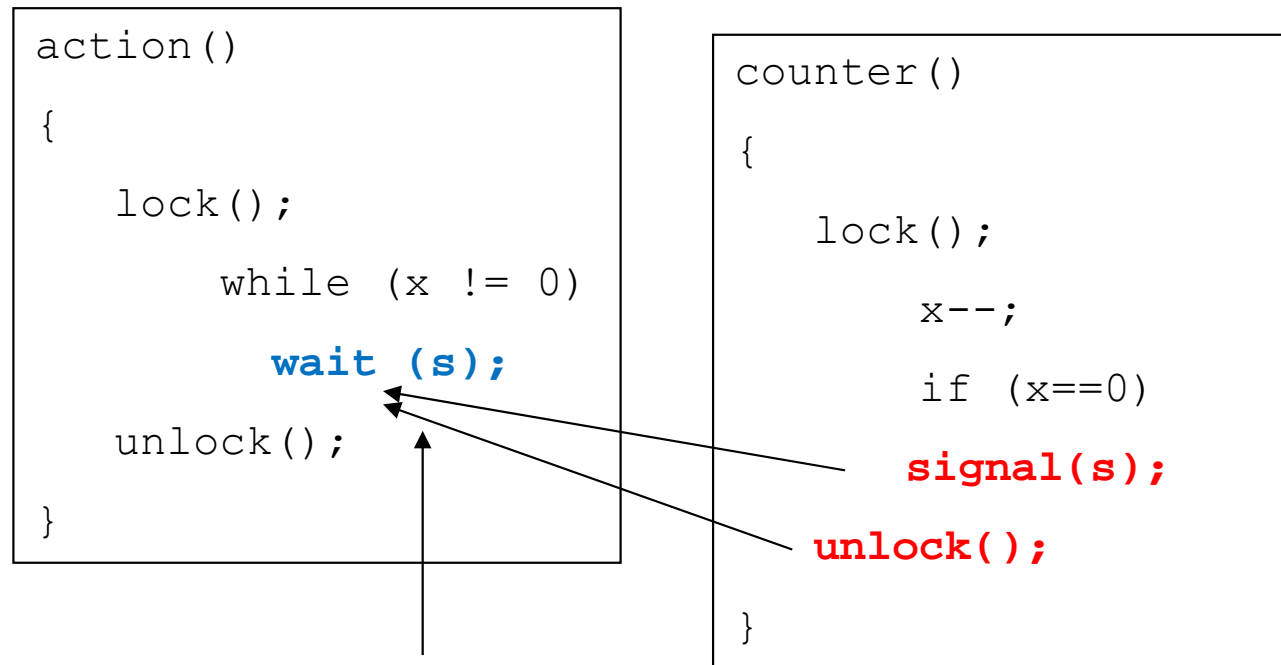
- Called with the mutex lock held
- Internals
  - Wake up at least one of the threads blocked on the specified condition variable
  - Scheduling policy determines the thread unblocked (if any)
  - Return zero on success; otherwise, an error number to indicate the error
- Variation: 

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

  
unblock all threads currently blocked on the specified condition variable



# CONDITIONAL WAITING



Both must occur before `wait()` returns

## EXERCISE 1 – RUN

1. Download

```
11c_pthread_cond_var.c
```

```
11c_pthread_cond_var_avoid_multiple_signals.c
```

2. Read the comments and understand the code involving sensors & actuators using condition variables

3. Compile and run the program

```
gcc program_name
```

```
./a.out
```

Which program runs faster and why? How to avoid multiple signals?



## EXERCISE 2

- In the classic producer consumer problem replace semaphores with condition variables.
- In the same problem, possibly implement the following functions using condition variables
  - `int getitem(buffer_t *itemp)`
    - removes item from buffer and put in \*itemp
  - `int putitem(buffer_t item)`
    - inserts item in the buffer

## REFERENCES

- Book: W. R. Stevens and S.A. Rago, "Advanced Programming in the Unix Environment", Addison Wesley, 2014, 3rd edition.
- W.R. Stevens, "UNIX Network Programming: Interprocess Communications", Vol. 2, Prentice Hall, 1999, 2nd Edition.
- [http://man7.org/linux/man-pages/man3/](http://man7.org/linux/man-pages/man3/pthread_create.3.html) (pthread\_create, pthread\_join)
- [http://man7.org/linux/man-pages/man3/pthread\\_yield.3.html](http://man7.org/linux/man-pages/man3/pthread_yield.3.html)