Operating Systems

Lecture 5.2

Concurrency Control Mechanisms - I

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Data Sharing among Threads
- Thread-safe vs Reentrant Functions
- Overview of Concurrency Control Mechanisms
- Achieving Mutual Exclusion:
  - Using **`pthread_mutex_t`**
  - Using **`pthread_spinlock_t`**
  - Using **`x86`** H/W Instructions
- Synchronize Execution Phases:
  - Using **`pthread_barrier_t`**
- Conditional Waiting:
  - Using **`pthread_cond_t`**
- Classic Synchronization Problems
  - Producer Consumer Problem
  - Dining Philosopher Problem
  - Reader-Writer Problem
  - Sleeping Barber Problem

Data Sharing among Threads

Data Sharing among Threads of a Process

What Data is Shared among Threads of a Process?

- **Global Variables:** Variables declared outside of all functions are accessible by all threads.
- **Static Variables:** Local variables with static keyword are shared across function calls and threads.
- **Heap Memory:** Memory allocated on the heap (e.g., via malloc, new) is shared if its address is stored in a global/static variable or passed to multiple threads.
- **Object Members:** If multiple threads operate on the same object instance, its data members are shared.

What Data is not Shared among Threads of a Process ?

- **Local Variables:** The variables declared inside functions without static keyword are stored on the Function Stack Frame. As each thread has its own FSF, so they get separate copies.
- **Function Parameters:** These are also stored on the FSF, therefore, each thread calling the same function has its own copy of parameters.

Example: Data Sharing (race1.c)

```c
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc, NULL);
   pthread_create(&t2, NULL, dec, NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("balance:%ld\n", balance);
   return 0;
}
```

```c
void * dec(void * arg){
   for(long i=0;i<100000000;i++)
      balance--;
   pthread_exit(NULL);
}

void * inc(void * arg){
   for(long i=0;i<100000000;i++)
      balance++;
   pthread_exit(NULL);
}
```

Example: Data Sharing (race1a.c)

```c
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc, NULL);
   pthread_create(&t2, NULL, dec, NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("balance:%ld\n", balance);
   return 0;
}
```

```c
void * dec(void * arg){
   int temp = balance;
   usleep(10000);
   temp = temp - 1;
   usleep(10000);
   balance = temp;
   pthread_exit(NULL);
}
void * inc(void * arg){
   int temp = balance;
   usleep(10000);
   temp = temp + 1;
   usleep(10000);
   balance = temp;
   pthread_exit(NULL);
}
```

Example: Data Sharing (race2.c)

```c
int charcount = 0;
void* f1(void * arg);
int main(int argc, char* argv[]){
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)argv[1]);
   pthread_create(&tid2, NULL, f1, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   printf("Number of characters in both files: %d\n", charcount);
   return 0;
}
void* f1(void* args){
   char* filename = (char*)args;
   char ch;
   int fd = open(filename, O_RDONLY);
   while((read(fd, &ch, 1)) != 0)
      charcount++;
   close(fd);
   pthread_exit(NULL);
}
```

Example: Data Sharing (race3.c)

```c
char** ptr;
void * thread_function(void * localarg);
int main(){
pthread_t tid[2];
int thread_ids[2] = {0,1};
char* msg[2] = {"Hello from Arif", "Hello from PUCIT"};
ptr = msg;
for(int i=0;i<2;i++)
pthread_create(&tid[i], NULL, thread_function, (void*)&thread_ids[i]);
for(int i=0;i<2;i++)
pthread_join(tid[i], NULL);
return 0;
}
```

```c
void * thread_function(void * localarg){
int myid = *((int*)localarg);
static int svar = 0;
printf("Thread %d starting...\n", myid);
int temp = svar; // Read
usleep(10000); // 10ms delay to increase race window
temp = temp + 1; // Modify
usleep(10000); // Another delay
svar = temp; // Write back
printf("[%d]: %s (svar = %d)\n", myid, ptr[myid], svar);
printf("Thread %d finishing...\n", myid);
pthread_exit(NULL);
}
```

Summary of Data Sharing (`race3.c`)

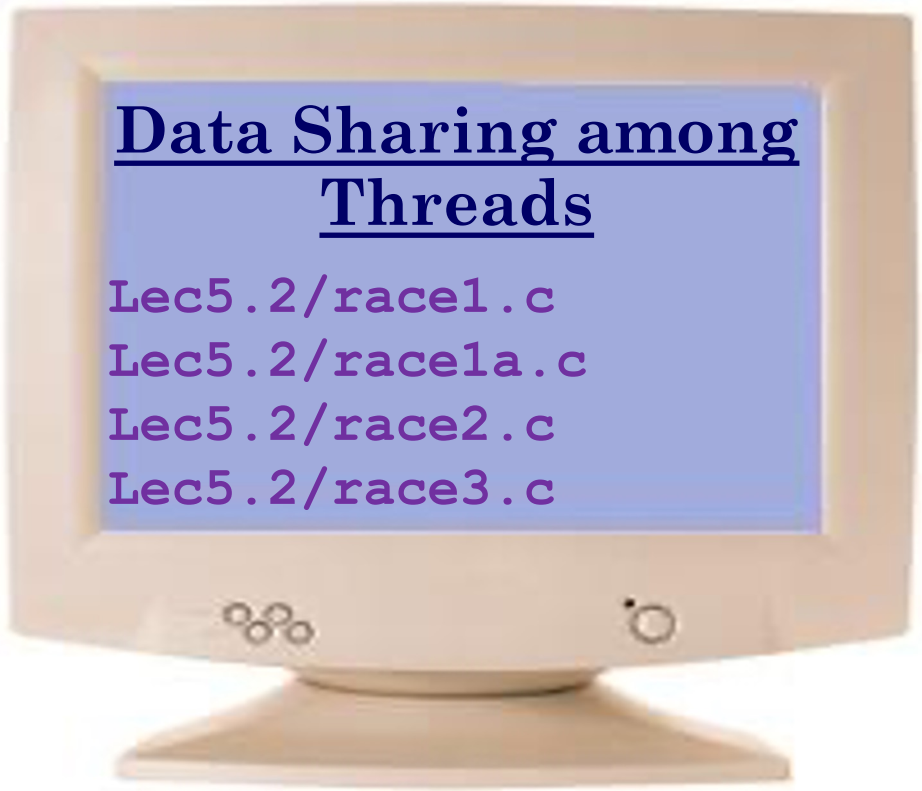
Race Condition occurs when:

- Variable is shared between threads
- At least one thread writes/modifies the variable
- Access is not synchronized

Only **`svar`** meets all the above three criteria, which is why it's the only variable with a race condition in `shareddata.c`

Variable	Shared?	Race Condition?	Reason
`ptr`	Yes	No	Read-only access
`svar`	Yes	Yes	Read-modify-write operations
`msg`	Yes	No	Read-only access
`myid`	No	No	Local variable (stack)
`temp`	No	No	Local variable (stack)
`tid`	No	No	Main thread only
`thread_ids`	Partial	No	Different indices per thread

Demonstration



Data Sharing among Threads

```
Lec5.2/race1.c
Lec5.2/race1a.c
Lec5.2/race2.c
Lec5.2/race3.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Thread-safe Functions
vs
Reentrant Functions

Four Classes of Thread unsafe Functions

Class I: Failing to protect shared variables:

- Multiple threads modify shared/global data simultaneously → race conditions.
- **Solution**: Use synchronization primitives to protect shared variable.

Class II: Relying on persistent state across invocations:

- Function assumes data stored from previous call is valid → concurrent calls overwrite it.
- **Solution**: Avoid persistent state, or protect with synchronization.

Class III: Returning a pointer to a static variable:

- All callers share the same static return buffer.
- **Solution**: Don't return static memory, use caller-provided buffers, or thread local storage.

Class IV: Calling a thread unsafe function:

- A function itself may internally use globals/statics, or call other unsafe functions.
- **Solution**: Call thread safe or re-entrant versions of functions.

Thread-safe vs Reentrant Function

Thread-safe Function

- A function that can be safely called by multiple threads at the same time even if they share data, as it uses synchronization primitives to ensures consistent results.
- A thread-safe function should meet the following criteria:
  - Produce consistent behavior when invoked concurrently by multiple threads.
  - Uses local variables or synchronized access to shared variables.
  - Avoids unsynchronized calls to non-thread-safe functions.
  - May use locks to serialize access.

Reentrant Function

- A function that can be safely interrupted (e.g., by signals, recursion, or concurrent execution) and re-entered before the previous execution completes. This is because each call uses its own data and must not rely on shared mutable state.
- A reentrant function should meet the following criteria:
  - Uses only local (stack) data.
  - Avoids static or global variables - or accesses them in a fully isolated/atomic way.
  - Doesn’t call non-reentrant functions.

Example 1: Reentrant vs Thread-safe

Is this function Thread-Safe?

No: The function uses a shared global variable **`temp`**, which is not protected by any synchronization mechanism. If two threads call `swap()` at the same time, they will both read and write **`temp`** concurrently, leading to a race condition and corrupted results.

Is this function Reentrant?

No: A reentrant function must not use global or static data. Since **`temp`** is a global variable, if the function is interrupted (e.g., by a signal handler calling `swap()` again), it would overwrite **`temp`** mid-execution, causing incorrect behaviour.

```
int temp;
void swap(int *x, int *y){
        temp = *x;
        *x = *y;
        *y = t;
}
```

Example 2: Reentrant vs Thread-safe

Is this function Thread-Safe?

Yes: The function is thread-safe because:

- The variable `temp` is declared with `__thread`, making it thread-local storage (TLS).
- Each thread gets its own independent copy of `temp`.
- Multiple threads can call `swap()` simultaneously without interfering with each other’s `temp` variable.
- The parameters `x` and `y` are local to each function call on each thread's stack.

```
__thread int temp;

void swap(int *x, int *y) {
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Is this function Reentrant?

No: The function is not reentrant because:

- Even though `t` is thread-local, it's still global within each thread.
- If the same thread calls `swap()` recursively or from a signal handler, the calls will share the same `temp` variable.
- This creates a race condition where nested calls can corrupt each other's intermediate state.

Example 3: Reentrant vs Thread-safe

Is this function Thread-Safe?

No: The function is not thread-safe because:

- The variable **t** is declared as static, making it global/shared across all threads.
- Multiple threads calling `swap()` simultaneously will access and modify the same **t** variable.
- This creates race conditions where one thread can overwrite another thread's stored value in **t**.
- Even though the function attempts to save/restore **t**, the save/restore operations themselves are not atomic.

```
static int t;
void swap(int *x, int *y) {
    int saved_t = t;
    t = *x;
    *x = *y;
    *y = saved_t;
    t = saved_t
}
```

Is this function Reentrant?

Yes: The function **is reentrant** because:

- It properly saves the original state of `t` at entry (`saved_t = t`).
- It restores the original state before exit (`t = saved_t`)
- Nested calls within the same thread will each have their own `saved_t` on the stack.
- Each level of recursion preserves and restores the global state correctly.

Example 4: Reentrant vs Thread-safe

Is this function Thread-Safe?

Yes: The function is thread-safe because:

- It does not use any global or shared variable
- Multiple threads can call this function at the same time.

```
void swap(int *x, int *y){
        int temp;
        temp = *x;
        *x = *y;
        *y = t;
}
```

Is this function Reentrant?

Yes: The function is reentrant because:

- It uses only local variables.
- Does not perform I/O or use non-reentrant functions.
- If the function is interrupted by a signal handler that also calls `swap()`, each invocation operates on its own stack frame with its own temp variable. The interrupted call can resume safely without any corruption of data.

Reentrant Functions

- In `glibc`, functions like **`asctime()`** return results in a static buffer, making them *not thread-safe* because multiple threads may overwrite the same memory.
- Their reentrant counterparts, such as **`asctime_r()`**, take a caller-supplied buffer, ensuring *per-call storage* and making them *safe for concurrent use* across threads.

```c
// Thread-unsafe returns pointer to static buffer
char *s = asctime(localtime(&t));
```

```c
// Thread-safe, uses caller-provided storage
char buf[26];
char *s = asctime_r(localtime_r(&t, &tm),buf);
```

So always compile your multi-threaded code with _REENTRANT defined:

```
$ gcc thread1.c -o thread1 -lpthread -D_REENTRANT
```

Category	Thread Unsafe Functions	REENTRANT versions
Time	`asctime()`	`asctime_r()`
Time	`ctime()`	`ctime_r()`
Host Lookup (by name)	`gethostbyname()`	`gethostbyname_r()`
Host Lookup (by address)	`gethostbyaddr()`	`gethostbyaddr_r()`
Random numbers	`rand()`	`rand_r()`
Time conversion	`localtime()`	`localtime_r()`
Password hash	`crypt()`	`crypt_r()`

Overview of Concurrency Control Mechanisms

Concurrency Control Mechanisms

Concurrency control mechanisms are synchronization primitives and techniques used to coordinate access to shared resources among multiple concurrent processes or threads, ensuring data consistency, preventing race conditions, and maintaining system correctness in multi-threaded/multi-process environments.

Overview of Concurrency Control Mechanisms

- **Software-level Synchronization Primitives:** Implemented through collaboration between C runtime libraries and the OS kernel. Examples include mutexes, spinlocks, condition variables, barriers, semaphores, and read-write locks.
- **Hardware-level Synchronization Primitives:** Hardware-based concurrency control mechanisms rely on special atomic CPU instructions (compare-and-swap and test-set-lock), that perform indivisible/atomic operations. These instructions enable synchronization directly at the hardware level without the need for kernel intervention or traditional locking mechanisms. Hardware atomics are commonly used in high-performance, low-latency applications where the overhead of locks is prohibitive.
- **Compiler-level Synchronization Primitives:** Higher-level concurrency control mechanisms are provided by programming languages, compilers, and frameworks to simplify thread synchronization. Instead of writing complex locking code manually, developers can use simple annotations or directives. These tools automatically generate the necessary synchronization logic behind the scenes. Examples include:
  - Java's `synchronized` keyword, which locks objects implicitly to prevent concurrent access.
  - GCC's `__transaction_atomic`, which allows blocks of code to run atomically, and rolling back changes if a conflict is detected.
  - C#'s `lock(object)` statement, which generates `Monitor.Enter()/Monitor.Exit()` calls wrapped in try-finally blocks to guarantee lock release even during exceptions.

S/W Level Primitives (Pthread Library)

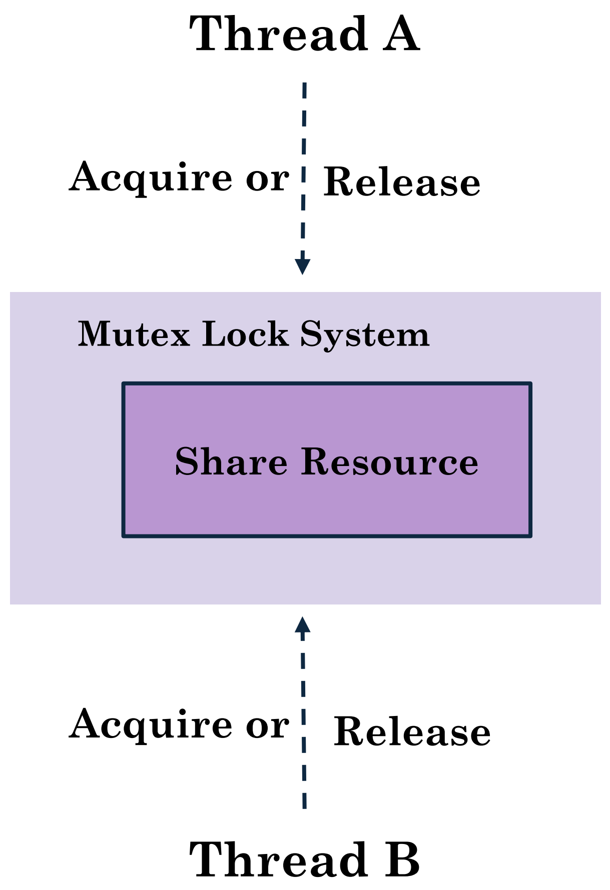
- **Mutexes (Mutual Exclusion Locks):** Binary locks ensuring only one thread can access a critical section at a time. Threads block/sleep when lock is unavailable, yielding CPU to other threads. Use a mutex when you need exclusive ownership of a critical section.
- **Spinlocks:** Like mutexes, spinlocks are also used to achieve mutual exclusion, but do busy waiting instead of blocking, i.e., threads continuously poll for lock availability instead of blocking. Use spinlocks when the CS is very short or scenarios where lock contention is rare.
- **Barriers:** Synchronization points where threads must wait until all participating threads reach the barrier before any can proceed, ensuring coordinated execution phases. Use in parallel algorithms where threads must complete one stage before starting the next.
- **Condition Variables:** Synchronization primitives that allow threads to wait for specific conditions to become true, working in conjunction with mutexes to provide efficient blocking/waking mechanisms. Use condition variable when you need threads to wait efficiently for a state change. (e.g., producer-consumer wait for buffer not empty/full.

By default `libpthread's` mutexes, barriers and condition variables (which are POSIX standard) and spinlocks (which are GNU/Linux extensions) are used by threads within the same process. However, if you place them in shared memory and change their attribute to PTHREAD_PROCESS_SHARED, then they can also synchronize cooperating processes.

Achieving Mutual Exclusion using `pthread_mutex_t`

Thread Synchronization using MUTEX

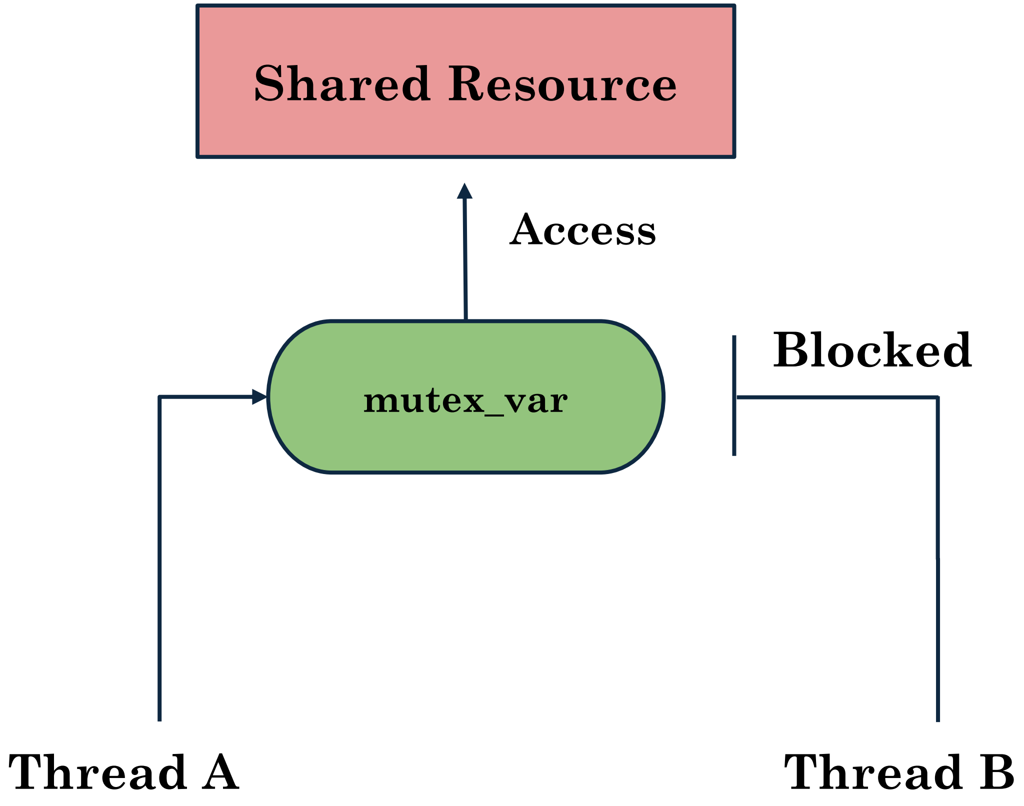
- A **Mutex** (MUTual EXclusion) is a binary lock that ensures that only one thread can access a critical section at a time. Threads block/sleep when lock is unavailable, yielding CPU to other threads. Use a mutex when you need exclusive ownership of a critical section.
- A mutex has two possible states:
  - **unlocked** → not owned by any thread.
  - **locked** → exactly one thread owns it. It can never be owned by two different threads simultaneously.



Thread Synchronization using MUTEX

- If a thread tries to lock an already locked mutex, the thread is blocked, until the mutex is released.
- Linux guarantees that no race condition occur among threads attempting to lock the same mutex.

- Mutex provides both achieve both:
    - **Mutual exclusion:** prevent multiple threads from accessing shared data simultaneously
    - **Serialization:** Enforce order in which threads access shared resources



How to use MUTEX?

1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one thread succeed → thread becomes owner of the mutex
4. The owner thread carry out operations on shared data
5. The owner threads unlock the mutex
6. Another waiting thread is allowed to acquire the mutex and repeat the process
7. After all work is done, the mutex is destroyed

MUTEX Initialization

Before using a mutex, it must be initialized. There are two ways to do this:

Static Initialization: Use this when the mutex is declared as a global or file-scope variable and default attributes are fine. The macro `PTHREAD_MUTEX_INITIALIZER` sets up the mutex automatically

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Run time initialization: In all other cases, we must dynamically initialize the mutex by calling `pthread_mutex_init()` function explicitly as shown:

```
pthread_mutex_t mut;

pthread_mutex_init(&mut, NULL);
```

- The above function creates a mutex referenced by `mut`.
- The second argument is a pointer to `pthread_mutexattr_t` object that specifies mutex attributes, which can specify the mutex type (normal, recursive, errorcheck), process sharing (private, shared). If the attribute argument is set to NULL, the default mutex attributes are used.

Locking, Unlocking and Destroying MUTEX

```
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
int pthread_mutex_destroy(pthread_mutex_t *mptr);
```

The **`lock()`** call will lock the `pthread_mutex_t` object referenced by `mptr`:

- If the mutex is unlocked → calling thread acquires the lock and continues.
- If the mutex is already locked → calling thread is blocked (put to sleep) until the mutex becomes available.
- Used when a thread is going to enter in its critical section.

The **`unlock()`** call release the mutex object referenced by `mptr`:

- If other threads are blocked on this mutex, one of them will be unblocked and allowed to acquire it. The scheduling policy shall determine which thread shall acquire the mutex.
- This call should be made only by the owner thread.
- Used when a thread comes out of the CS.

The **`destroy()`** call destroys the mutex object referenced by `mptr`.

MUTEX Deadlocks

Be sure to observe following points to avoid deadlocks while using mutexes:

- No thread should attempt to lock or unlock a mutex that has not been initialized.
- Only the owner thread of the mutex (i.e the one which has locked the mutex) should unlock it.
- Do not lock a mutex that is already locked.
- Do not unlock a mutex that is not locked.
- Do not destroy a locked mutex.

Achieving ME using pthread_mutex_t

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
pthread_mutex_t mut;
int main(){
   pthread_t t1, t2;
   pthread_mutex_init(&mut, NULL);
   pthread_create(&t1, NULL, inc, NULL);
   pthread_create(&t2, NULL, dec, NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   pthread_mutex_destroy(&mut);
   printf("balance:%ld\n", balance);
   return 0;
}
```

```
void * dec(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_mutex_lock(&mut);
      balance--;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
}

void * inc(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_mutex_lock(&mut);
      balance++;
      pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
}
```

Demonstration

CSP Solution using pthread_mutex_t

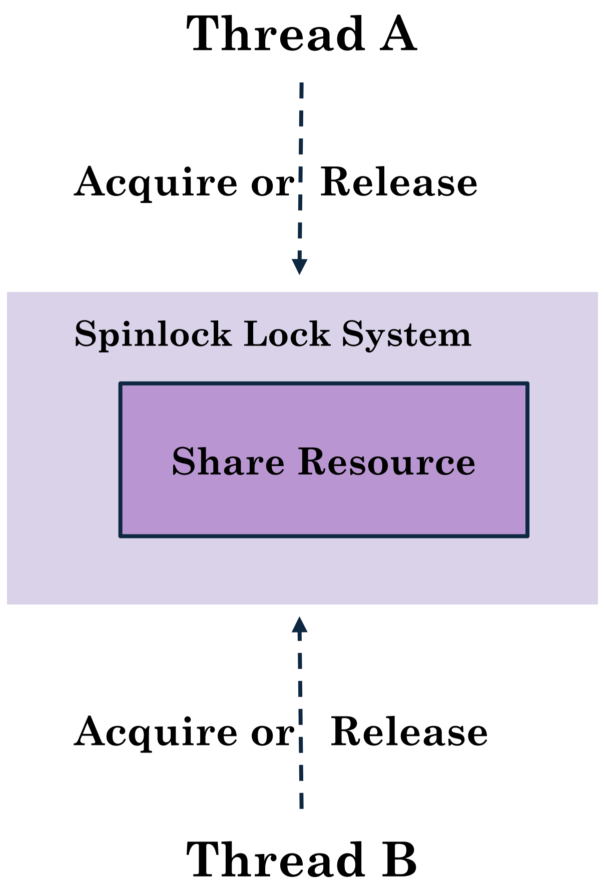
Lec5.2/solrace1-mutex.c
Lec5.2/solrace2-mutex.c

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Achieving Mutual Exclusion using `pthread_spinlock_t`

Thread Synchronization using Spinlocks

- Like mutexes, **spinlocks** are also used to achieve mutual exclusion, but do busy waiting instead of blocking, i.e., threads continuously poll for lock availability instead of blocking. Use spinlocks when the CS is very short or scenarios where lock contention is rare.
- A spinlock has two possible states:
  - **unlocked** → not owned by any thread.
  - **locked** → exactly one thread owns it. It can never be owned by two different threads simultaneously.



Spinlock Initialization

Before using a mutex, it must be initialized. There are two ways to do this:

Static Initialization: Use this when the spinlock is declared as a global or file-scope variable and default attributes are fine. The macro `PTHREAD_SPINLOCK_INITIALIZER` sets up the spinlock automatically

```
pthread_spinlock_t spin = PTHREAD_SPINLOCK_INITIALIZER;
```

Run time initialization: In all other cases, we must dynamically initialize the spinlock by calling `pthread_spin_init()` function explicitly as shown:

```
pthread_spinlock_t spin;

pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);
```

- The above function creates a spinlock object referenced by `spin`.
- The second argument can be either `PTHREAD_PROCESS_PRIVATE` or `PTHREAD_PROCESS_SHARED`. In the second scenario the spinlock can be shared between processes, when placed in shared memory.

Locking, Unlocking and Destroying Spinlock

```
int pthread_spin_lock(pthread_spinlock_t *spin);
int pthread_spin_unlock(pthread_spinlock_t *spin);
int pthread_spin_destroy(pthread_spinlock_t *spin);
```

The **`lock()`** call will lock the `pthread_spinlock_t` object referenced by `spin`:

- If the spinlock is unlocked → calling thread acquires the lock and continues.
- If the spinlock is already locked → calling thread actively spins (continuously polls the lock in a tight loop) until the spinlock becomes available.
- Used when a thread is going to enter in its critical section.

The **`unlock()`** call release the `pthread_spinlock_t` object referenced by `spin`:

- If other threads are spinning on this spinlock, one of them will immediately detect the unlock and acquire it. The scheduling policy shall determine which thread shall acquire the spinlock.
- This call should be made only by the owner thread.
- Used when a thread comes out of the CS.

The **`destroy()`** call destroys the `pthread_spinlock_t` object referenced by `spin`.

Achieving ME using pthread_spinlock_t

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
pthread_spinlock_t spin;
int main(){
   pthread_t t1, t2;
   pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);
   pthread_create(&t1, NULL, inc, NULL);
   pthread_create(&t2, NULL, dec, NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   pthread_spin_destroy(&spin);
   printf("balance:%ld\n", balance);
   return 0;
}
```

```
void * dec(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_spin_lock(&spin);
      balance--;
     pthread_spin_unlock(&spin);
   }
   pthread_exit(NULL);
}

void * inc(void * arg){
   for(long i=0;i<100000000;i++){
      pthread_spin_lock(&spin);
      balance++;
     pthread_spin_unlock(&spin);
   }
   pthread_exit(NULL);
}
```

Demonstration

CSP Solution using pthread_spinlock_t

Lec5.2/solrace1-spinlock.c
Lec5.2/solrace2-spinlock.c

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

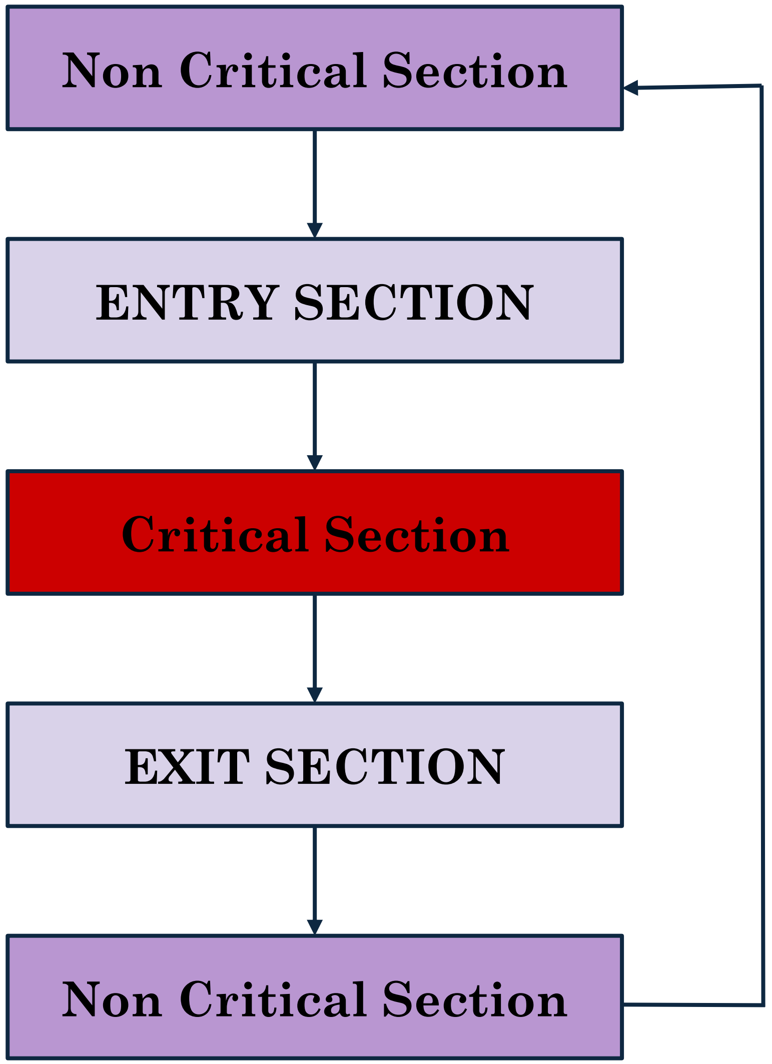
Achieving Mutual Exclusion using x86 H/W Instruction

H/W Level Synchronization Primitives

- H/W level synchronization does not mean that we are using some IC to handle the CS problem. It basically mean that we are using instructions specified in the Instruction Set Architecture of a processor to handle CSP.
- Disabling Interrupt: In a uni-processor environment, we can handle CSP, if we forbid/mask interrupts (by setting MI bit to 1), while some shared variable is being modified. However, if a user level program is given the ability to disable interrupts, then it can disable the timer interrupt. Thus context switching will not take place, thus allowing it to use the CPU w/o letting other processes to execute. In a multi-processor environment, it is not feasible to disable interrupts, because it will only prevent processes from executing on the CPU in which interrupts are disabled. Processes can execute on other CPUs and therefore does not guarantee mutually exclusive access to program state. Moreover, disabling interrupts on all CPUs gives a great performance loss. Therefore, disabling interrupts works, but safe to use only inside OS/kernel.
- Hardware-based concurrency control mechanisms rely on special atomic CPU instructions (compare-and-swap and test-set-lock), that perform indivisible/atomic operations.
- These instructions enable synchronization directly at the hardware level without the need for kernel intervention or traditional locking mechanisms.
- Hardware atomics are commonly used in high-performance, low-latency applications where the overhead of locks is prohibitive.

Structure of CSP by Disabling Interrupts

```
do {

Disable Interrupts

<Critical Section>

Enable Interrupts

<Remainder Section>

} while(true);
```



The compare_and_swap Instruction of x86

```
bool compare_and_swap(int *lock, int expected, int new_value) {
   if (*lock == expected) { // COMPARE step
        *lock = new_value;  // SWAP step
        return true;        // Success!
   } else
        return false;       // Failed - someone else changed it
}
```

- If the `lock` and expected value are both same, i.e., zero that means the lock is free. In this case the function replace the `lock` with the `new_value` (normally set to 1, i.e., occupied) and returns `true`. So the thread has acquired the lock and can proceed to critical section.
- But if the `lock` and `expected` do not match, that means, some other thread has changed the lock to 1 (i.e., occupied). In this case the function returns `false` and the thread can try again to acquire the lock.

The function executes as an **atomic** instruction: the comparison and assignment happen as one indivisible operation that cannot be interrupted.

Achieving ME using x86 compare_and_swap instr

```
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc, NULL);
   pthread_create(&t2, NULL, dec, NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("balance:%ld\n", balance);
   return 0;
}
void * dec(void * arg){
   for(long i=0;i<100000000;i++){
       acquire_lock();
       balance--;
      release_lock();
   }
   pthread_exit(NULL);
}
void * inc(void * arg){
   for(long i=0;i<100000000;i++){
       acquire_lock();
       balance++;
      release_lock();
   }
   pthread_exit(NULL);
}
```

```
long balance = 0;
volatile int lock = 0;
void acquire_lock(){
  while (1) {
    int expected = 0;
    if (__sync_bool_compare_and_swap(&lock, expected, 1))
       break;
  }
}


void release_lock() {
   __sync_lock_release(&lock);
}
```

Demonstration

CSP Solution using pthread_spinlock_t

`Lec5.2/solrace1-hw.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Synchronize Execution Phases using `pthread_barrier_t`

What is a Barrier?

- Barriers are synchronization points where a specified number of threads must all arrive before any of them can proceed further. You can think of a barrier as a checkpoint where threads must wait for their teammates before continuing.
- Key Characteristics of a barrier are:
  - Threads that reach the barrier first, must wait.
  - When the last thread arrives, ALL waiting threads are released simultaneously.
  - The same barrier can be used multiple times for repeated synchronization.
  - The number of participating threads is set at initialization.
- When to use barrier?
  - Iterative algorithms (each iteration needs all threads to finish previous iteration).
  - Pipeline stages (all threads must complete stage N before starting stage N+1).
  - Parallel data processing (ensure all data is processed before aggregation).
  - Scientific computing (synchronize computational phases in simulations).

Operations on pthread_barrier_t Variable

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_brrierattr_t *attr,
                         unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- Barriers can only be initialized via `pthread_barrier_init()` function.
  - The first argument `barrier`, is the object to be initialized.
  - The second argument `attr`, specifies the barrier attribute, which is mostly kept NULL.
  - The third argument `count` (>`0`), specifies the number of worker threads that much reach the barrier before any are released. Once count threads arrive, all are simultaneously released.
- The `pthread_barrier_wait()` function creates a synchronization checkpoint for threads. When a thread calls this function:
  - The thread blocks at the barrier until exactly the required number of threads have also called `pthread_barrier_wait()` on the same barrier object.
  - Once all required threads arrive (specified in the count parameter), the barrier releases ALL waiting threads simultaneously, allowing them to continue execution together.
- The `pthread_barrier_destroy()` function shall destroy the barrier referenced by barrier and release any resources used by the barrier.

Example No Barriers: barrier1.c

```
void * worker(void * arg);
int main(){
   pthread_t t1, t2, t3;
   int id1=1, id2=2, id3=3;
   pthread_create(&t1, NULL, worker, (void*)&id1);
   pthread_create(&t2, NULL, worker, (void*)&id2);
   pthread_create(&t3, NULL, worker, (void*)&id3);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);
   printf("\nAll threads. Completed\n");
   return 0;
}
```

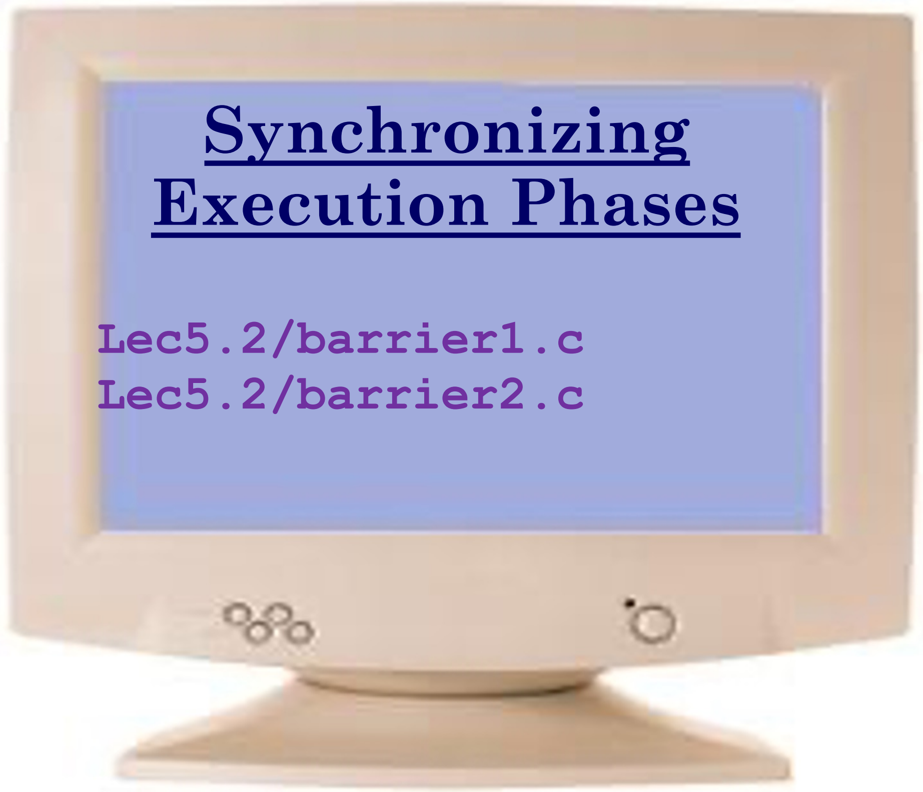
```
void * worker(void * arg){
   int tid = *(int*)arg;
// Phase 1: Each thread does its work
   printf("Thread %d: Starting Phase 1\n", tid);
   sleep(tid); // Simulate different work times
   printf("Thread %d: Finished Phase 1\n", tid);

//No barrier here - Threads proceed immediately to Phase 2

// Phase 2: Each thread does its work
   printf("Thread %d: Starting Phase 2\n", tid);
   sleep(1);
   printf("Thread %d: Finished Phase 2\n", tid);

   pthread_exit(NULL);
}
```

Example Barriers: barrier2.c

```c
void * worker(void * arg);
pthread_barrier_t phase_barrier;
int main(){
   pthread_t t1, t2, t3;
   int id1=1, id2=2, id3=3;
   pthread_barrier_init(&phase_barrier, NULL, 3);
   pthread_create(&t1, NULL, worker, (void*)&id1);
   pthread_create(&t2, NULL, worker, (void*)&id2);
   pthread_create(&t3, NULL, worker, (void*)&id3);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_join(t3, NULL);
   pthread_barrier_destroy(&phase_barrier);
   printf(”\nAll threads. Completed\n”);
   return 0;
}
```

```c
void * worker(void * arg){
   int tid = *(int*)arg;
// Phase 1: Each thread does its work
   printf("Thread %d: Starting Phase 1\n", tid);
   sleep(tid); // Simulate different work times
   printf("Thread %d: Finished Phase 1\n", tid);

// Threads wait for all threads to finish Phase 1
   printf("Thread %d: Waiting at barrier …\n", tid);
   pthread_barrier_wait(&phase_barrier;
   printf("Thread %d: Passed barrier (all threads finished
Phase 1\n", tid);

// Phase 2: Each thread does its work
   printf("Thread %d: Starting Phase 2\n", tid);
   sleep(1);
   printf("Thread %d: Finished Phase 2\n", tid);

   pthread_exit(NULL);
}
```

Demonstration



Synchronizing Execution Phases

```
Lec5.2/barrier1.c
Lec5.2/barrier2.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Conditional Waiting using `pthread_cond_t`

Why Condition Variables?

```
void * f1(void * arg);
int main(){
   pthread_t t1, t2, t3;
   int id1=1, id2=2, id3=3;
   pthread_create(&t1, NULL, f1, (void*)&id1); //pass thread no
   pthread_create(&t2, NULL, f1, (void*)&id2);
   pthread_create(&t3, NULL, f1, (void*)&id3);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   pthread_join(t3,NULL);
   return 0;
}
void * f1(void * arg){
   int tid = *(int*)arg;
   char* messages[] = {"", "Learning is ", "fun with ", "Arif Butt"};
   usleep(100000);
   fprintf(stderr, "%s", messages[tid]); //print message
   pthread_exit(NULL);
}
```

Demonstration

Serialization using Condition Variables

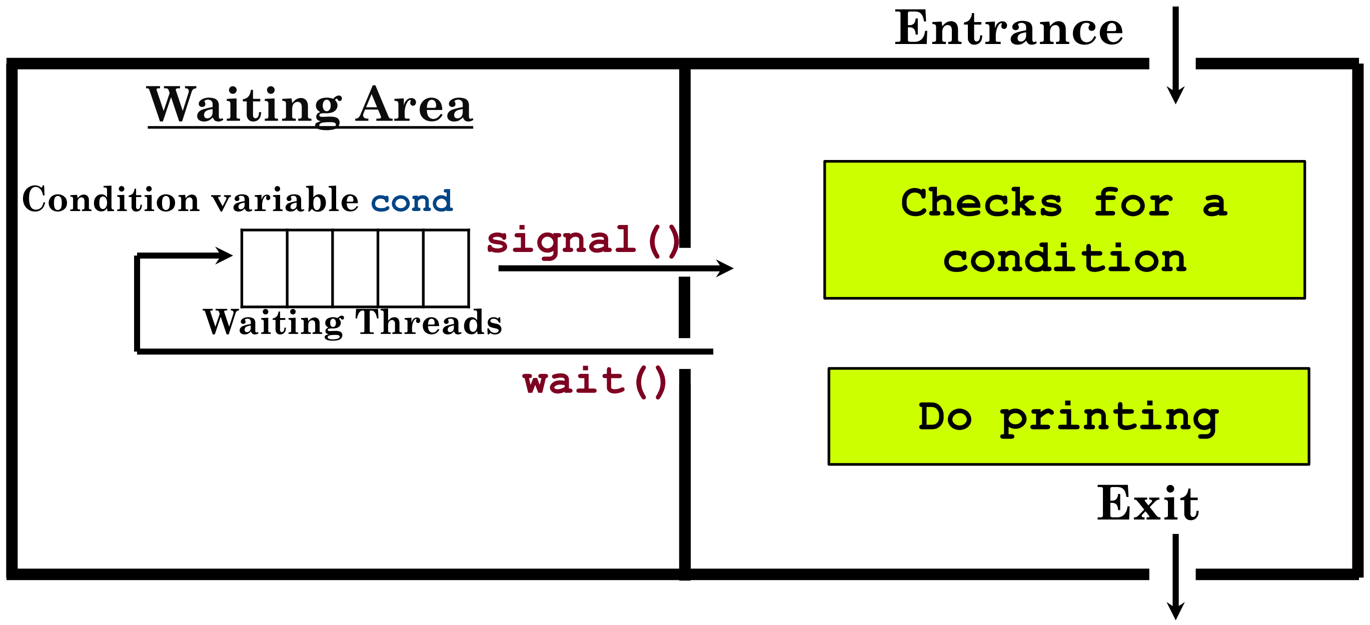
`Lec5.2/serialize1.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

What is a Condition Variable?

- A condition variable is a synchronization construct that allows threads to suspend execution and relinquish the processors until some condition/state is satisfied.
- The two basic operations on condition variables are:
    - Signal(): Wake up a sleeping thread on this condition variable.
    - Wait(): Release lock, goto sleep, reacquire lock after you are awoken up.
- Every condition variable works together with an associated mutex that protects the shared data representing the condition being waited for. So we can say that a condition variable enables a thread to sleep inside a CS. Any lock held by the thread is automatically released when the thread is put to sleep.
- The wait() call atomically releases the mutex and puts the thread to sleep, then automatically reacquires the mutex when the thread wakes up. This prevents race conditions between checking the condition and going to sleep.

Mutex is for locking, while condition variable is for waiting.



Initializing pthread_cond_t Variable

Static Initialization: In case where default attributes are appropriate, the following macro can be used to initialize a **`pthread_cond_t`** variable

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Run time initialization: In all other cases, we must dynamically initialize the condition variable using **`pthread_cond_init()`**

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

This function initializes the condition variable object pointed to by `cond` using the condition attributes specified in `attr`. If `attr` is `NULL`, default attributes are used instead. Linux Threads implementation supports no attributes for conditions, hence the `attr` parameter is actually ignored.

Operations on pthread_cond_t Variable

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

- The thread that calls **`pthread_cond_wait()`** atomically unlocks its second argument `mutex` and waits for the condition variable `cond` to be signaled by suspending its execution.
- The **`pthread_cond_signal()`** restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted, but it is not specified which.
- Unlike `pthread_cond_signal()` which wakes only one thread, **`pthread_cond_broadcast()`** ensures that all waiting threads get a chance to check if the condition they're waiting for has been satisfied. This is useful when the condition change might affect multiple threads or when you're unsure which specific thread should handle the condition.

Serialization using Condition Variables

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int turn = 1;
void * f1(void * arg);
int main(){
  pthread_t t1, t2, t3;
  int id1=1, id2=2, id3=3;
  pthread_create(&t1, NULL, f1, (void*)&id1);
  pthread_create(&t2, NULL, f1, (void*)&id2);
  pthread_create(&t3, NULL, f1, (void*)&id3);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  printf("\n\n");
  return 0;
}
```

```c
void * f1(void * arg){
  int tid = *(int*)arg;
  char* messages[] = {"", "Learning is ", "fun with
", "Arif Butt"};

  pthread_mutex_lock(&mutex);
  while(turn != tid)
      pthread_cond_wait(&cond, &mutex);
  usleep(100000);
  fprintf(stderr, "%s", messages[tid]);
  turn++; //increase the turn
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mutex);
  pthread_exit(NULL);
}
```

Demonstration

GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

Classic Synchronization Problems

Producer Consumer Problem

Producer Consumer Problem

Producer produces information that is consumed by a consumer process. To allow producer and consumer run concurrently we must have a buffer that can be filled by the producer and emptied by the consumer. The buffer can be bounded or unbounded

- **Unbounded Buffer:** Places no practical limit on the size of the buffer. The consumer may have to wait for new items if the buffer is empty, but the producer can always produce new items
- **Bounded Buffer:** Assumes a fixed size buffer. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full

While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer. If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item

Producer Consumer Problem (cont…)

Implicit Synchronization:

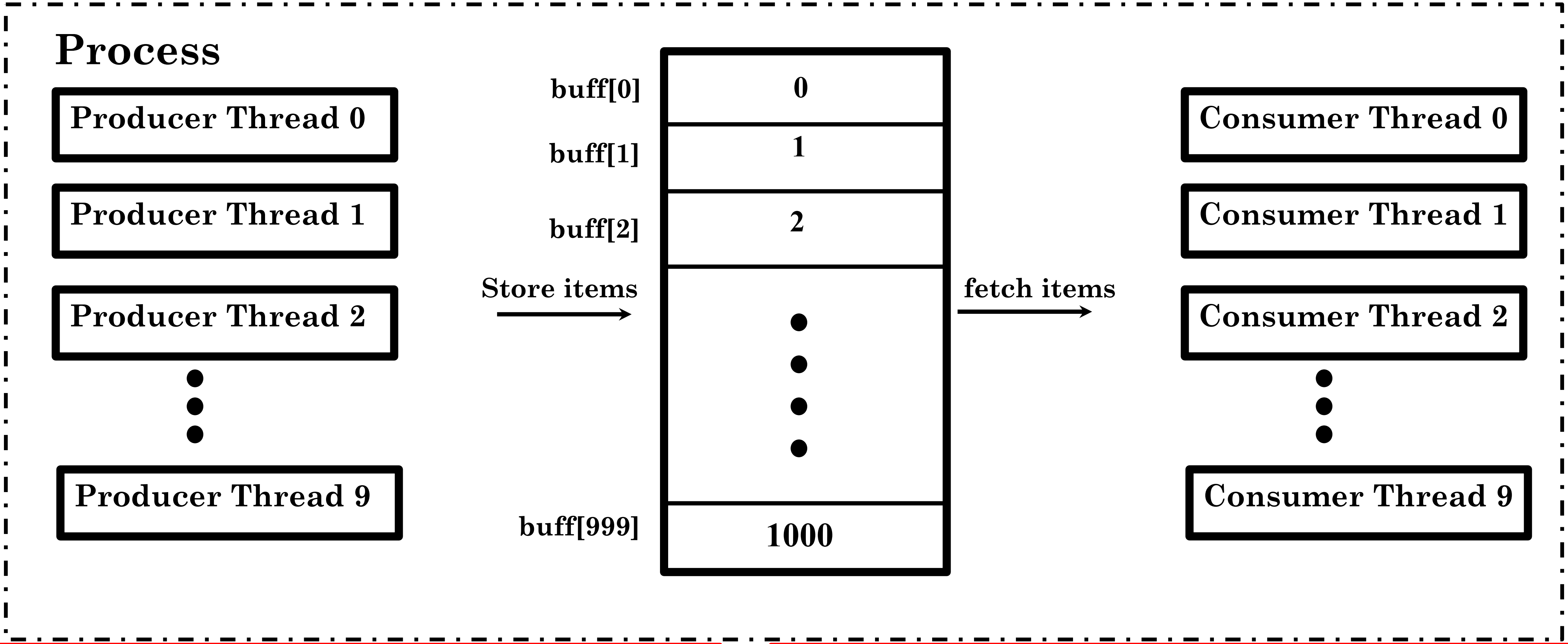
```
$ grep prog1.c | wc -l
```

`grep` is a producer process and **`wc`** is a consumer process. **`grep`** writes into the pipe and **`wc`** reads from the pipe. The required synchronization is handled implicitly by the kernel. If producer gets ahead of the consumer (i.e. the pipe fills up), the kernel puts the producer to sleep when it calls **`write()`**, until more room is available in the pipe. If consumer gets ahead of the producer (i.e. the pipe is empty), the kernel puts the consumer to sleep when it calls **`read()`**, until some data is there in the pipe.

Explicit Synchronization:

When we as programmers are using some shared memory/data structure, we use some form of IPC between the procedure and the consumer for data transfer. We also need to ensure that some type of explicit synchronization must be performed between the producer and consumer.

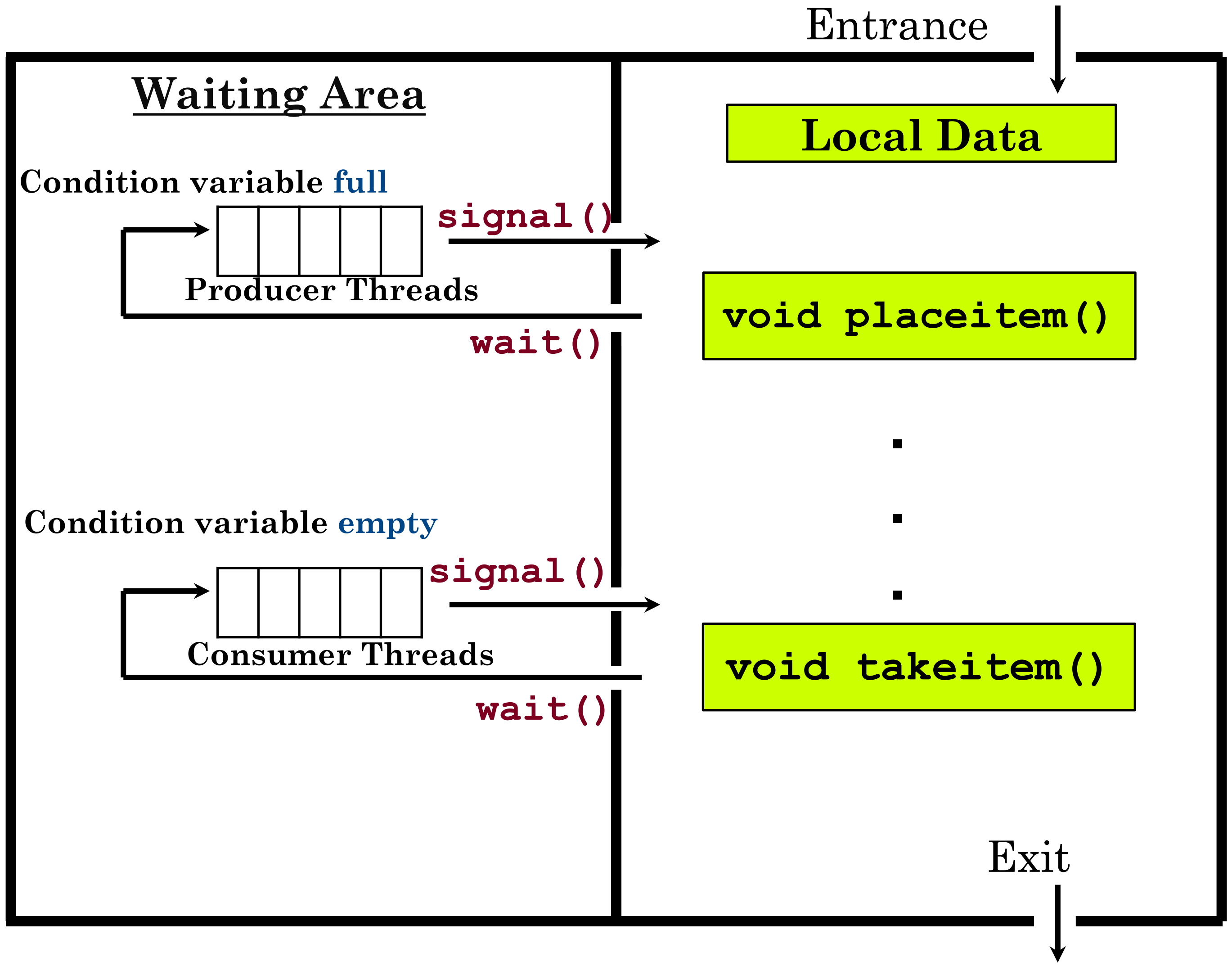
Producer Consumer Problem (cont…)



Each producer thread obtains a mutex lock and then accesses the buffer and places a number at that location. Producer must block, when the buffer is full. Moreover, producer must signal to a blocked consumer on empty buffer.

Each consumer thread obtains a mutex lock and then accesses the buffer and removes the number from that location. Consumer must block when the buffer gets empty. Moreover, consumer must signal to a blocked producer on full buffer.

Use of pthread_cond_t to Notify



Producer Consumer Problem (Un-Bounded Buffer)

```
buffer = [];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t itemAvailable = PTHREAD_COND_INITIALIZER;
```

Producer

```
do {
   item = produceItem();
   pthread_mutex_lock(&mutex);
   placeItem(buffer, item);
   pthread_cond_signal(&itemAvailable);
   pthread_mutex_unlock(&mutex);
} while(1);
```

Consumer

```
do {
  pthread_mutex_lock(&mutex);
  while (isEmpty(buffer))
    pthread_cond_wait(&itemAvailable, &mutex);
  item = takeItem(buffer);
  pthread_mutex_unlock(&mutex);
  consumeItem(item);
} while(1);
```

Producer Consumer Problem (Bounded Buffer)

```
buffer = [SIZE];
int count = 0; //current number of items in buffer
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t itemAvailable = PTHREAD_COND_INITIALIZER;// items available to consume
pthread_cond_t spaceAvailable = PTHREAD_COND_INITIALIZER; // space available to produce
```

Producer

```
do {
   item = produceItem();
   pthread_mutex_lock(&mutex);
   while (count == SIZE)
      pthread_cond_wait(&spaceAvailable, &mutex);
   placeItem(buffer, item);
   count++;
   pthread_cond_signal(&itemAvailable);
   pthread_mutex_unlock(&mutex);
} while(1);
```

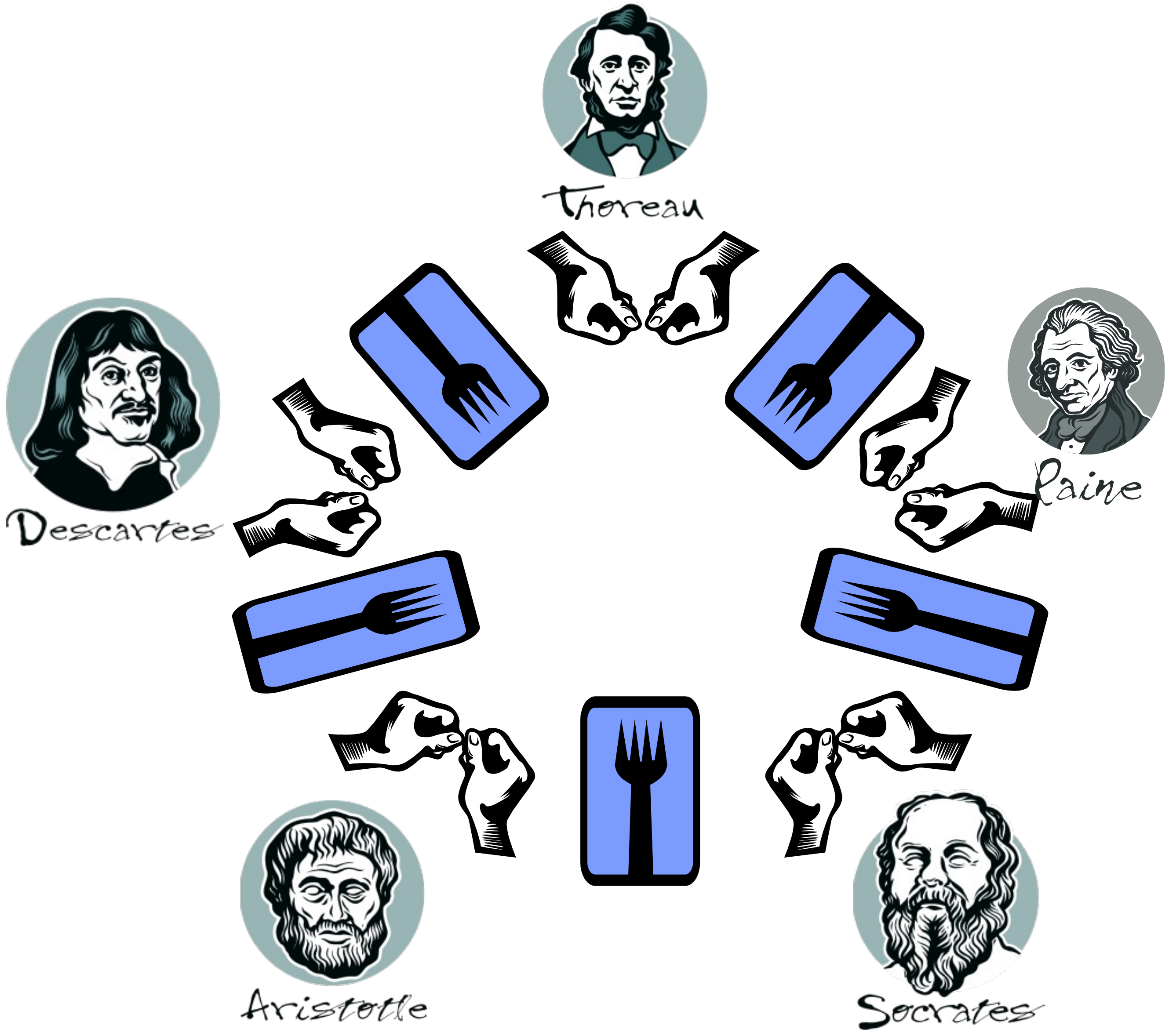
Consumer

```
do {
  pthread_mutex_lock(&mutex);
  while (count == 0)
    pthread_cond_wait(&itemAvailable,&mutex);
  item = takeItem(buffer);
  count--;
  pthread_cond_signal(&spaceAvailable);
  pthread_mutex_unlock(&mutex);
  consumeItem(item);
} while(1);
```

Dining Philosopher Problem

Dining Philosopher Problem

- Five Chinese philosophers, who spend their lives just thinking and eating
- Sit on a round table with five plates of rice and five chopsticks
- A philosopher requires two chopsticks to eat (so at a time a maximum of two philosophers can eat)
- Protocol used for eating:
  - Picks up left chopstick and then right chopstick, one at a time in either sequence
  - If successful in acquiring two chopsticks, the philosopher eats for a while, then puts down the chopstick and continues to think
  - One fine day all became hungry at a time. All pick up the left chopstick first and then look for the right chopstick, which was not there. They did not fight like us but waited and waited and waited and finally starved to death. Sad day in China….



Dining Philosopher Problem (Deadlock Version)

```
pthread_mutex_t chopstick[5]; // 5 chopsticks, all initialized
for (int i = 0; i < 5; i++)    // Initialize chopsticks
   pthread_mutex_init(&chopstick[i], NULL);
```

Philosopher$_i$

```
do {
   think();
   pthread_mutex_lock(&chopstick[i]);          // pick up left chopstick
   pthread_mutex_lock(&chopstick[(i+1)%5]);    // pick up right chopstick
   eat();
   pthread_mutex_unlock(&chopstick[(i+1)%5]);// put down right chopstick
   pthread_mutex_unlock(&chopstick[i]);        // put down left chopstick
} while(1);
```

DEADLOCK SCENARIO:

- All 5 philosophers pick up their left chopstick simultaneously
- Each waits for right chopstick, but it's held by the next philosopher - Circular wait → DEADLOCK!
- Everyone starves to death.

Dining Philosopher Problem (Asymmetric Solution)

```
pthread_mutex_t chopstick[5]; // 5 chopsticks, all initialized
for (int i = 0; i < 5; i++)    // Initialize chopsticks
   pthread_mutex_init(&chopstick[i], NULL);
```

Philosopher$_i$

```
do {
   think();
   if (i == 4){ //last philosopher picks up in reverse order
      pthread_mutex_lock(&chopstick[(i+1)%5]);   // pick up right chopstick
      pthread_mutex_lock(&chopstick[i]);         // pick up left chopstick
   }else{ // others picks up left first
      pthread_mutex_lock(&chopstick[i]);         // pick up left chopstick
       pthread_mutex_lock(&chopstick[(i+1)%5]); // pick up right chopstick
   }
   eat();
   pthread_mutex_unlock(&chopstick[(i+1)%5]);// put down right chopstick
   pthread_mutex_unlock(&chopstick[i]);      // put down left chopstick
} while(1);
```

Dining Philosopher Problem (Maximum Concurrent Dinners)

```
pthread_mutex_t chopstick[5]; // 5 chopsticks, all initialized
for (int i = 0; i < 5; i++)   // Initialize chopsticks
   pthread_mutex_init(&chopstick[i], NULL);
pthread_mutex_t diningRoom = PTHREAD_MUTEX_INITIALIZER;
int diningCount = 0;
const int MAX_DINERS = 4; // Allow only 4 philosophers to dine simultaneously
```

Philosopher$_i$

```
do {
   think();
// Entry to dining room
   pthread_mutex_lock(&diningRoom);
   while (diningCount >= MAX_DINERS) {
      pthread_mutex_unlock(&diningRoom);
      usleep(1000); // brief delay
      pthread_mutex_lock(&diningRoom);
   }
   diningCount++;
   pthread_mutex_unlock(&diningRoom);
```

Philosopher$_i$

```
// Pick up chopsticks
   pthread_mutex_lock(&chopstick[i]);
   pthread_mutex_lock(&chopstick[(i+1)%5]);
   eat();
   // Put down chopsticks

pthread_mutex_unlock(&chopstick[(i+1)%5]);
   pthread_mutex_unlock(&chopstick[i]);
// Exit dining room
   pthread_mutex_lock(&diningRoom);
   diningCount--;
   pthread_mutex_unlock(&diningRoom);
} while(1);
```

Dining Philosopher Problem (Condition Variables)

```
// Shared variables
typedef enum {THINKING,HUNGRY,EATING} philosopher_state_t;

philosopher_state_t state[5] = {THINKING, THINKING,
THINKING, THINKING, THINKING};

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t self[5];

for (int i = 0; i < 5; i++)
   pthread_cond_init(&self[i], NULL);

// Helper function to test if philosopher i can eat
void test(int i) {
    if (state[i] == HUNGRY && //I am hungry
        state[(i+4)%5] != EATING && //left not eating
        state[(i+1)%5] != EATING)//right not eating
     {
            state[i] = EATING;
            pthread_cond_signal(&self[i]);
     }
}
```

Philosopher$_i$

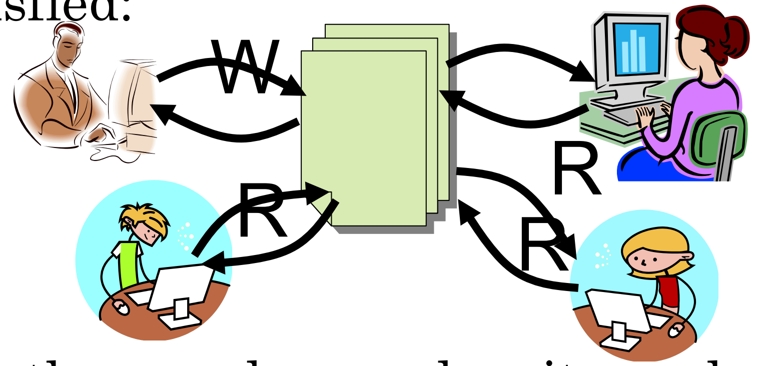
```
do {
    think();
// Try to eat
    pthread_mutex_lock(&mutex);
    state[i] = HUNGRY;
    test(i);//see if both chopsticks are available
    while (state[i] != EATING)
       pthread_cond_wait(&self[i], &mutex);
    pthread_mutex_unlock(&mutex);
    eat();
    pthread_mutex_lock(&mutex);
    state[i] = THINKING;
    test((i+4)%5);// check if left neighbor can eat
    test((i+1)%5);// check if right neighbor can eat
    pthread_mutex_unlock(&mutex);
} while(1);
```

Reader-Writer Problem

Reader Writer Problem

For successful read-write operations, following conditions must be satisfied:

- Two or more readers can access shared data simultaneously.
- Only one writer can access it at a time.
- If a writer is writing to the file, no reader may read it.



Readers have Priority

- If one or more readers are reading a shared resource and some other readers and writers also want to access that shared resource; we will let the readers read and writers wait until there is no reader reading the shared resource.
- If a reader want to read, it wait for a minimum amount of time.

Writers have Priority

- If one or more readers are reading a shared resource and some other readers and writers also want to access that shared resource; we will NOT let any further readers to come in and read, rather let the old readers finish reading and let a writer write.
- If a writer wants to write, it waits for minimum amount of time.

Reader Writer Problem (Readers have Priority)

```
int readCount = 0; // number of active readers
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // protects readCount
pthread_mutex_t wrt = PTHREAD_MUTEX_INITIALIZER; // writer exclusion
```

Writer

```
do {
   pthread_mutex_lock(&wrt);//acquire write access
// CRITICAL SECTION - WRITING IS PERFORMED
   performWriting();
   pthread_mutex_unlock(&wrt); //release write access
} while(1);
```

Reader

```
do {
   pthread_mutex_lock(&mutex);
   readCount++;
   if (readCount == 1)//1st reader block writers
      pthread_mutex_lock(&wrt);
   pthread_mutex_unlock(&mutex);
// CRITICAL SECTION - READING IS PERFORMED
   performReading();
   pthread_mutex_lock(&mutex);
   readCount--;
   if (readCount == 0)//last reader allow writers
      pthread_mutex_unlock(&wrt);
   pthread_mutex_unlock(&mutex);
} while(1);
```

Sleeping Barber Problem

Sleeping Barber Problem

A barber shop consists of a room with **n** waiting chairs and **one** barber chair:

- If there are no customers to be served the barber goes to sleep.
- If a customer arrives and the barber is asleep, the customer wakes up the barber.
- If the barber is busy, but chairs are available, then the customer sits on one of the free chairs.
- If a customer enters barber shop and all chairs are occupied, then the customer leaves the shop.

Customer

– Check if chair available, if not leave.

– Inform barber that I have arrived.

– Wait until barber cuts his hair.

Barber

– Sleep until a customer wakes him up.

– Service customer (during that remember to update available chairs).

– Tell customer to leave after finished.

– Repeat for other customers.

To Do

- Watch SP video on Synchronization among threads

https://youtu.be/SvFr7rPWI3g?si=QVdmv9njl0342EBN

- Write down working C programs for the classic synchronization problems discussed in the lecture slides.

Coming to office hours does NOT mean that you are academically weak!