Multithreading in C (Pthreads)

Level 1

- What are threads ?
- Why threads ?
- Execution environment for threads
  - Stack Memory
  - Heap Memory
- Launching a thread
- Terminating a thread
- Controlling Threads :
  - pthread_join
  - pthread_detach
  - pthread_cancel
- Thread specific Data
- Blocking the thread
- Signaling the thread
- Inter thread Communication

Pre-Requisite :
Must be good in C
Linux OS
Good with Basic Data Structures

Agenda

~ 12 hr

General Multithreading

- Thread Creation/Destruction
- Multi-Threading Vs Concurrency Vs Parallelism
- Joinable & Detached Threads, Race Conditions
- Thread Cancellation, Listener Threads, Returning Results from Threads
- Inter Thread Communication, Pausing and Resuming Threads

Thread Synchronization

- C.S, Mutexes, Locks, Deadlocks
- CV, CV Vs Mutex, Wait & Signal, Broadcast
- Producer-Consumer, Dining Philosopher, Spurious Wake ups
- Semaphores, Internal Implementation, Zero-One Semaphores
- Thread Management, Getting Ready for Sequel (Adv) Course on Thread Sync + Asynchronous Programming Concepts

Pre-Requisite

- Good in C/C++ Programming
- Basic knowledge of Data Structures
- Linux Machine (Native Or as a VM)
- Gcc Compiler
- Github Account (Free)
- Zeal to think and Learn

Telegram Grp : telecsepracticals
www.csepracticals.com

Take Away

- Able to Design Multi-threading Applns
- Choose when to thread and when not to
- Implement Thread Synchronization
- Apply Concepts to Other Programming Lang In future
- Answer Interview Questions, Drive the interview on this topic
- Enhanced Coding and Development Skills
- Build Resume and Github Portfolio
- Network with Instructors and Other Students, and get opportunity to get referrals

What are Threads ?

- Let us do start with some reverse engineering
- Let us write our first Multi-threaded Hello-World Program, and then we will discuss what exactly threads are !

Github : https://github.com/sachinites/MultithreadingBible

Code : MultithreadingBible/ThreadBasics/HelloWorld/hello_world.c

What are Threads ?

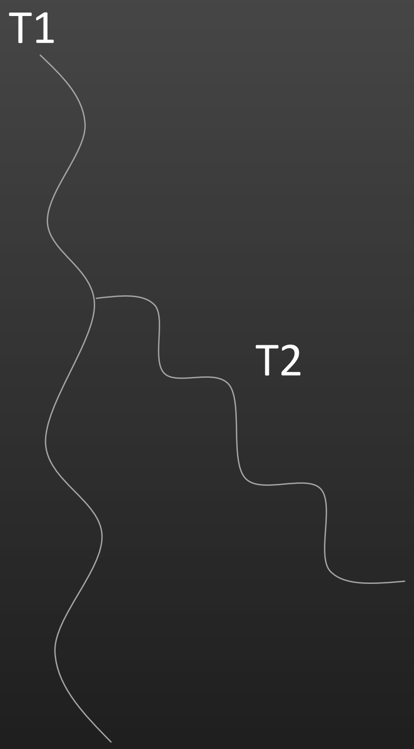
- A Thread is a basic unit of execution flow
- A Thread runs in the context of a process
- A process has at-least one thread -> main thread
- A thread can create other threads, other threads can create more threads and so on ..

HelloWorld Example : Discussions

Discussions :

- Argument passing
  - (Do not pass address of local variables)
- Race condition on Thread Creation
- Thread Termination
  - Thread fn returns
  - pthread_exit(0);
  - Thread Cancellation
- If main thread dies, all other thread dies by default, but vice-versa is not true
- Terminating main (return 0 Vs pthread_exit(0))

HelloWorld Example : Discussions



Thread Shares Resources

- An OS allocates resources to threads – Memory, CPU, Access to Hardware etc

- All threads are siblings, there is no parent-child (having extra prvileges) relationship between threads of the same process, no hierarchy

- Every threads has its own life-cycle – birth, live and death independent of other threads in the system
  - Exception Rule :
    - When main thread of a process dies – all other threads of a process are also terminated, Vice Versa is not true

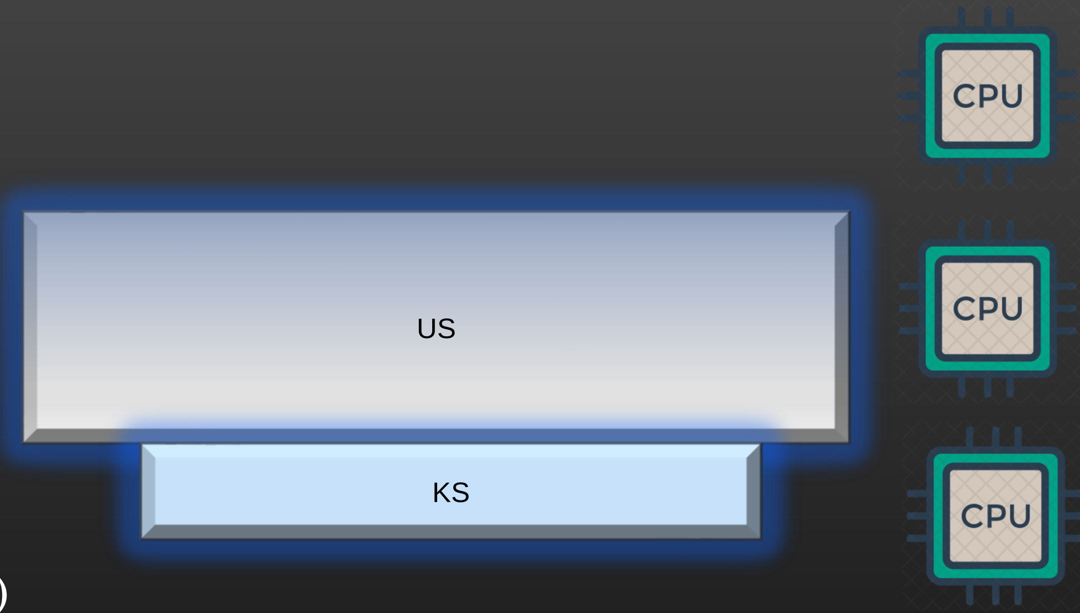
- Multiple Threads of the processes share same Virtual Address Space of a process
  - Resource allocated by one thread is visible to rest of the others
  - Heap Memory, Sockets, File Descriptors etc , Global Variables

- What threads do not share is the stack memory, every thread has its own stack memory 👌

What is Schedulable – Threads or Processes ?

- Kernel (OS) do not schedule processes, it schedules threads

- Thread is a schedulable entity, not a process
  - However, this rule is violated in certain error conditions:
    - If a thread Seg-fault, entire process is terminated (including all threads)
    - A signal is delivered per process, not per thread (better to understand when you study Signals)



- The race condition on thread creation is due to the fact that which thread the kernel chooses to allocate CPU - the parent thread or new child thread

- Kernel Schedules threads on multiple CPUs as per the scheduling policy

Context Switching

Concurrency

Parallelism

Concurrency in General

- Doing two or more different tasks :
  - One at a time
  - Switching between tasks
    - Preempting current task
    - picking up next, partially do it, then preempt it
    - picking up next, partially do it, then preempt it
    - Pick up the first task from the same point where it was left, partially do it, then preempt it
    - . . .

Eg :

Consider three well-diggers assigned a task to dig their respective 100 ft deep well, they have only one well drilling tool which they have to share :

- Only one person can dig at a time
- Current person take rest, handover the tool to 2nd person, 2nd person resume
- 2nd person take rest, handover the tool to 3rd, person, 3rd person resume
- And continue until task is complete

- Work of all the well-diggers is in progression, though slow

Parallelism in General

- Doing two or more different tasks :
  - In Parallel

Eg :

Consider three well-diggers assigned a task to dig their respective 100 ft deep well, each one of them have their personal well drilling tool :

- All three can dig in parallel
- There is no need for anybody to take rest

- Work of all the well-diggers is in progression, fast

Time taken In parallelism << Time taken in Concurrency

Singularism in General

- Doing two or more different tasks :
  - One task at a time
  - Don’t preempt until the task is complete

Eg :

Consider three well-diggers assigned a task to dig their respective 100 ft deep well, each one of them have their personal well drilling tool :

- Only one is allowed to dig at a time
- Once started, he cannot preempt until the task is complete
- Next start his job only when prev one completes

- Work of all the well-diggers is NOT in progression
- Whether they have one drilling tool or 3, don’t matter

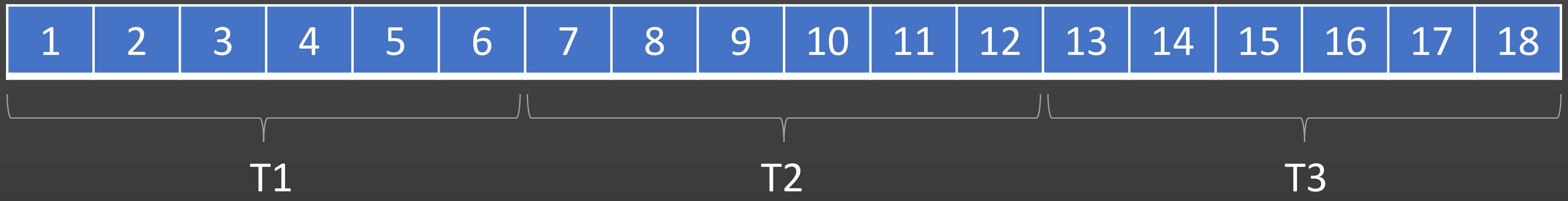
Time taken In parallelism << Time taken in Singularism < Time taken in Con-currency

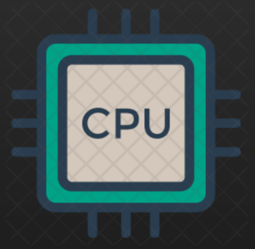
- Con-curency don’t give speed, it give progression
- Parallelism gives speed, but it demands hardware resources

Analogy Mapping

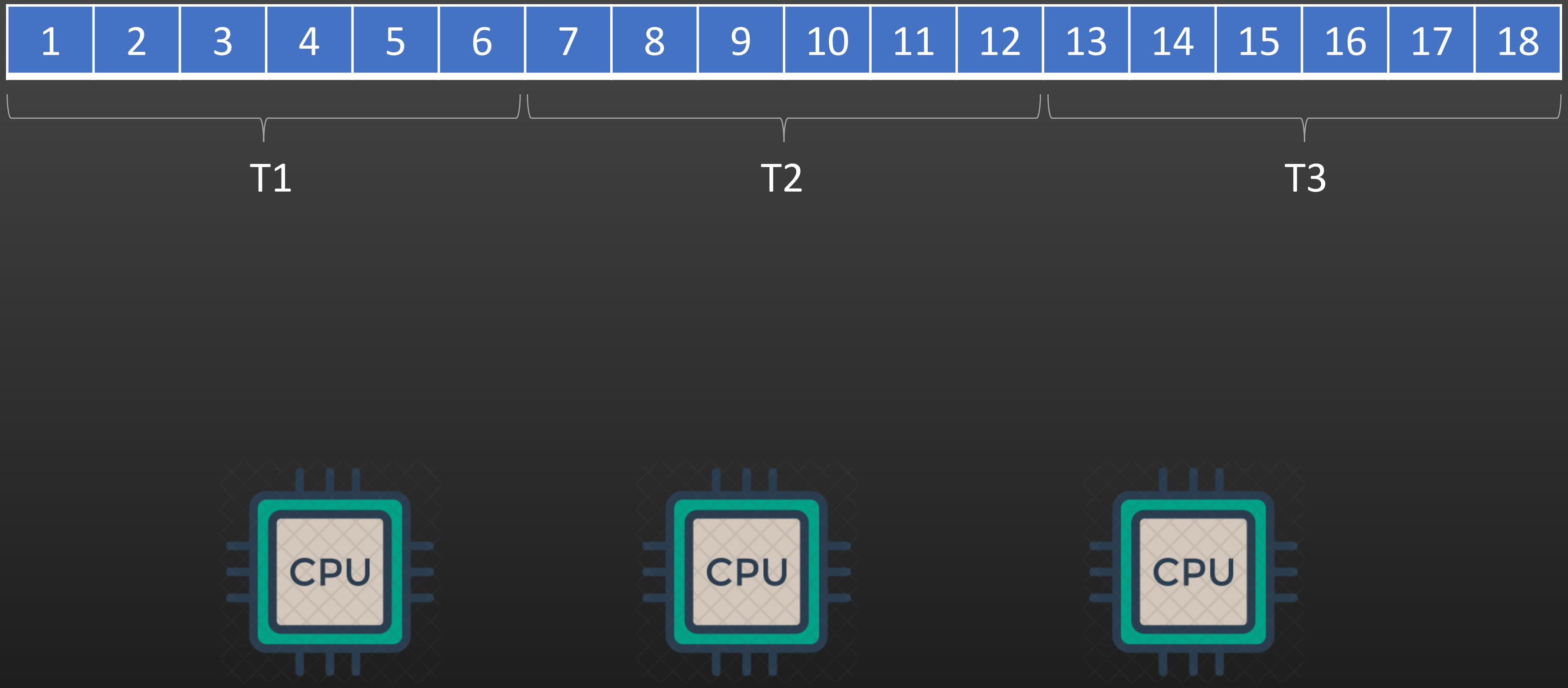
	Well Digging Analogy	Threading Env
Workers	Well diggers	Threads
Resources	Digging tool	CPU & Memory etc
Transition	One WD to another	Context Switching
Work to accomplish	100 ft well	Work to be done by each thread

Concurrent Threads

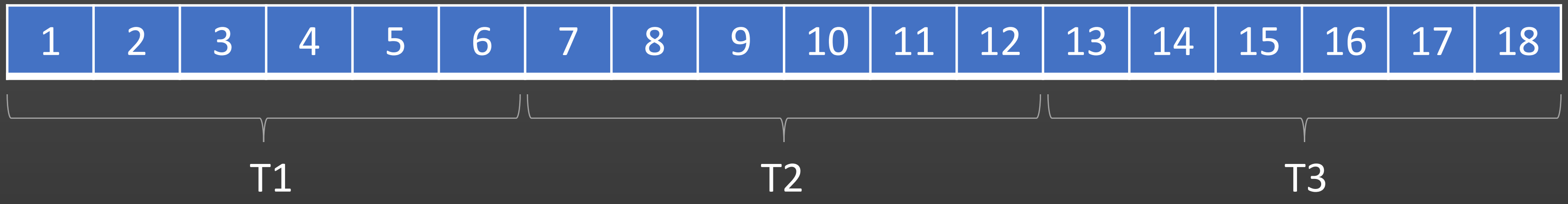


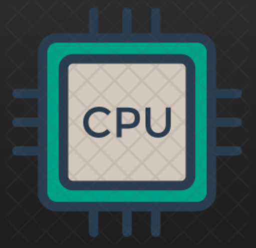


Parallel Threads



Singular Threads





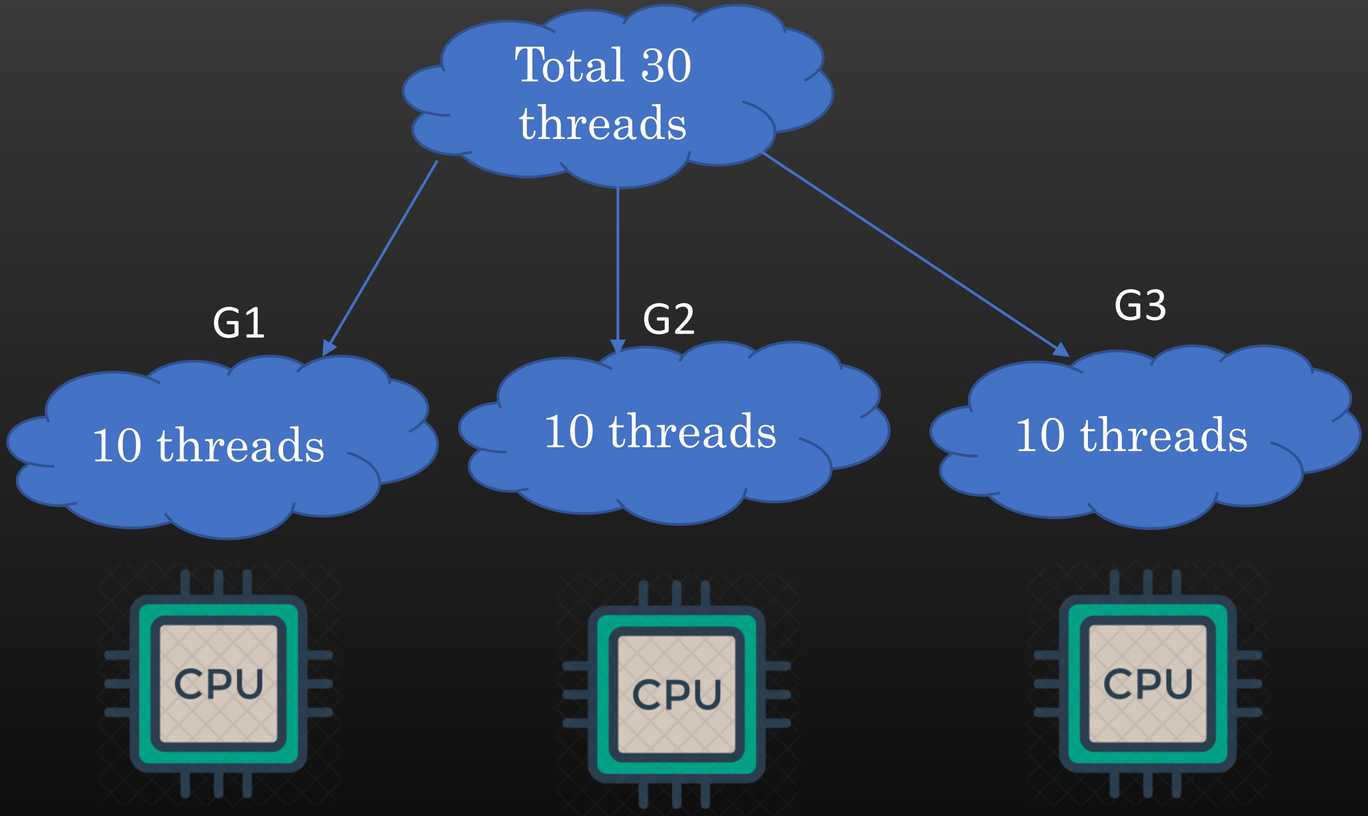
Time taken In parallelism << Time taken in Singularism < Time taken in Con-currency

Why do we need Concurrency (Threads)

- There are 100s of threads running on your system at a time
  - User threads
  - System threads

- We can have finite no of CPU –
  - 8 CPUs
  - 16 CPUs

- All live Threads have to share the CPUs
  - Parallelism on available CPUs
  - Con-currency on each CPU
  - Parallelism and Concurrency Co-exist in system



- Threads belong to the same group runs concurrently

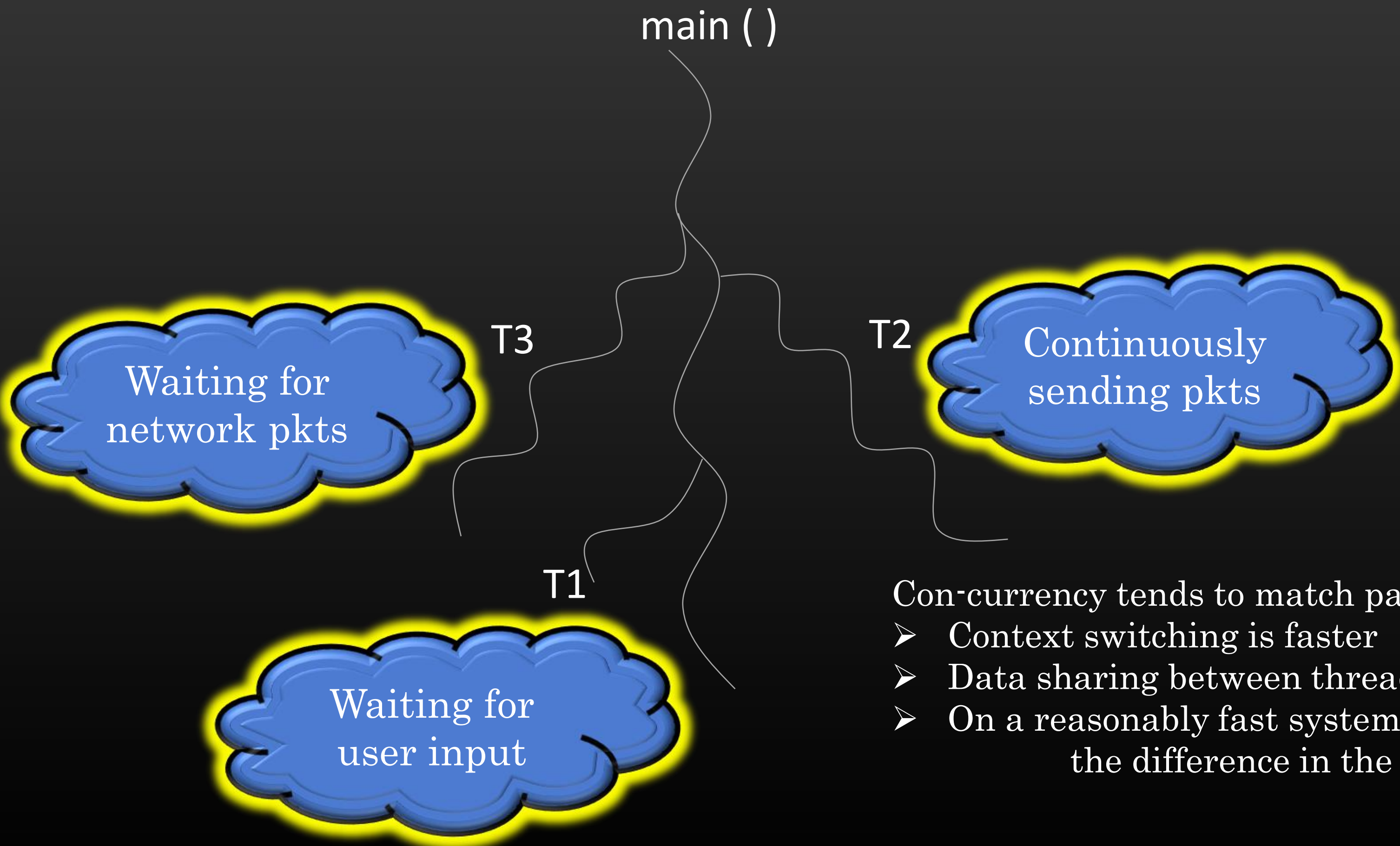
- Threads belonging to different groups runs in parallel

- Our Computer System is a hybrid of Con-currency and Parallelism

Concurrency Example 1

- We need concurrency because :
  - We cannot have large no of CPUs, CPUs need to be shared
  - We need to ensure progression

- When process needs to wait for I/O, while continue to execute to complete the work at the same time
  - Eg : A process waiting for network packets in one threads, while sending periodic packets in other threads, while waiting for user input in other threads

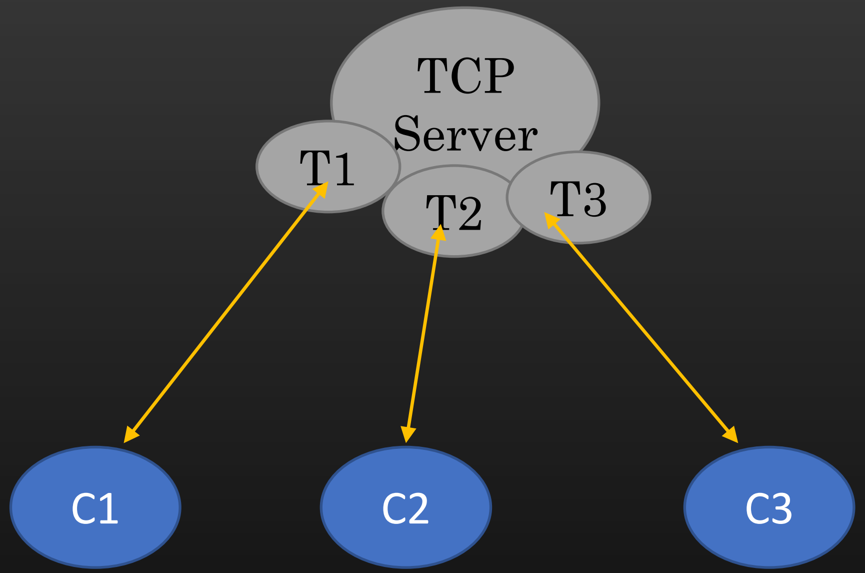


Con-currency tends to match parallelism if

- Context switching is faster
- Data sharing between threads is minimal
- On a reasonably fast systems, human being cannot detect the difference in the speed between parallelism and concurrency

Concurrency Example 2

- When work could be spitted into smaller independent chunks which then can be processed by worker threads
  - Eg : A Multithreaded TCP Server serving 100 TCP connected clients at the same time

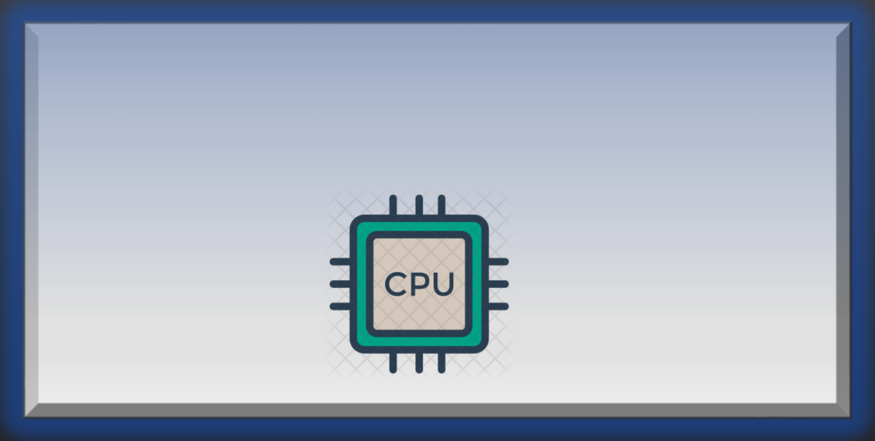


TCP Server delegates the task to Worker threads to entertain the client Request

Just like, a jewelry shop owner, after Understanding the client's needs, handover the client to one of his sub-ordinates

Context Switching

- We need to discuss context switching at a high level because, many of the thread concepts are directly and indirectly influenced by Context Switching



1. T1 is preempted, T2 is scheduled
2. When T1 is preempted, all its execution state and memory state is saved
3. T2 is resumed from same instruction and memory state
4. Context Switching implements Concurrency –
   Making things appear running in parallel, but actually it is one at a time

Necessity is the mother of invention.
Need for concurrency is the mother of Context Switching
Why do we need concurrency ?

Cause - CS
Effect - Concurrency

Why Threads are called Light-Weighted Processes

- A thread is called a light-weighted process because :
  - When a thread is created, it uses almost all the pre-existing resources of main-thread, hence OS don't have to work too hard to create (or delete) an isolated execution environment for a new thread
    - Page Tables are already setup
    - Shared Libraries are already loaded
    - Sockets are already opened
  - When thread dies, OS don't have to cleanup every resource used by the thread as those resources could be still in use by other threads of the same process
    Eg : Heap Memory, Sockets, Opened Files, IPCs etc
  - Context Switching happens fast from T1 to T2 (Thread Switching), where T1 and T2 are threads of the same process, contrary to when T1 and T2 belong to different processes (Process Switching)
    (Google and read more)

Overlapping and Non-Overlapping work

Overlapping & Non-Overlapping Work

If Thread T1 is doing work W1 and Thread T2 doing work W2 , then W1 and W2 is said to be overlapping if W1 and W2 operates on same Data

Eg : W1 - sorting the array A in Ascending order
W2 - sorting the array A in Descending order

Since Array is a common data on which Thread T1 and T2 are operating, therefore, W1 and W2 are overlapping work

- If Threads access the same Data structures of a process , say, some global variable then work done by two threads are overlapping work

Why do we need Threads ?

- Do we use threads for speed ?
  - Yes, if work can be divided into non-overlapping sub-work and system has multiple CPU units
  - Eg :

Consider a process P has to sort two arrays of integers.

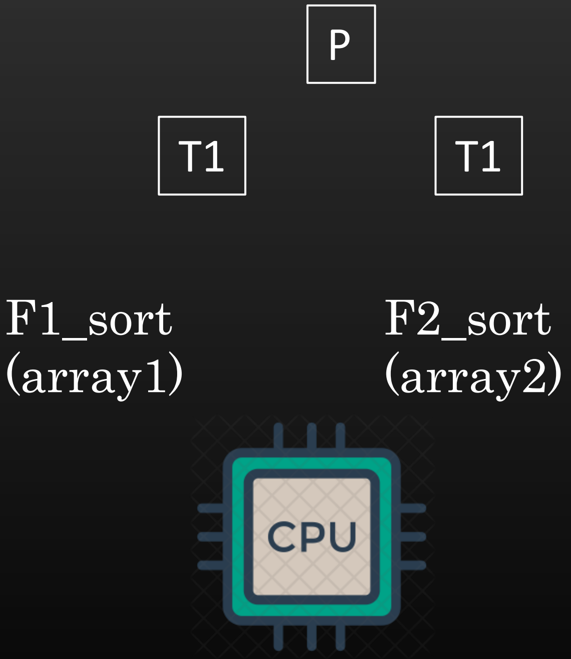
Sequential Processing

```
Do_sort_work()
{
        F1_sort (array1)
        F2_sort (array2)
}
```

Time : S1

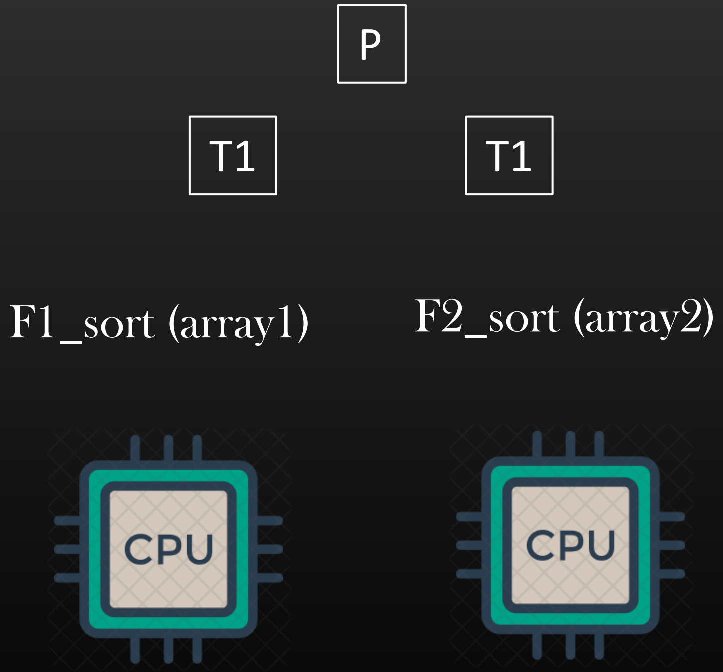
If CPU has a speed of N I/per sec, I will be anyway N I/per irrespective of no of threads it is executing ! It's a hardware limit.

Concurrent Processing



Time : S2
S1 < S2

Parallel Processing

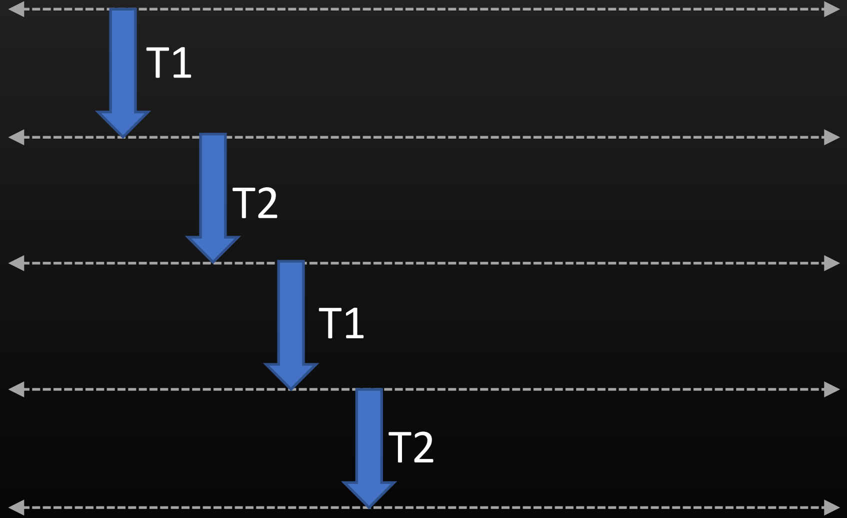


Time : S3 = S1/2
S3 < S1 < S2

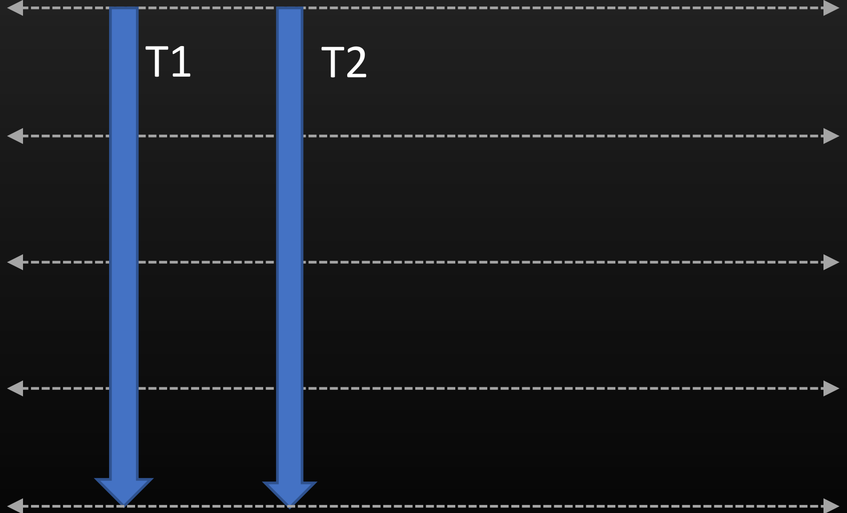
Summary

- When Multiple Threads (same or different Processes) share the same CPU with overlapping work, we get concurrency
- When Multiple Threads (same or different Processes) share same CPUs with non-overlapping work, We get concurrency
- When Multiple Threads (same or different Processes) share different CPUs with non-overlapping work, We get Parallelism

Concurrency



Parallelism



Understanding the term Asynchronous

- People love surprises , right ? Well, not really – depend on the surprise.

- Asynchronous refers to :
  - Anytime
  - Unpredictable
  - Unplanned

- A user has fired the command to the system, and while the system is busy processing it, User fired another one

- A process P send Network pkt to some destination, and proceed ahead to do its remaining work without worrying about the reply, it will process the reply whenever it would come

- A process P has just recvd a network pkt, and is processing it. Immediately it recvs another, and and another, and another, and some command from user to make situation even worst. A Well-Designed System should be able accommodate all operations, no need to process them instantly if not possible but queue them at-least

- Nature is Asynchronous, Life is Asynchronous

- Synchronous is exactly opposite

Understanding the term Synchronous

- Synchronous refers to :
  - Pre – planned
  - Deterministic
  - Do "Everything" in one go

- A Process P1 sends msg to process P2, but process P1 anxiously wait for the reply before it does any other work

- foo ( ) -> bar ( )

- A program P reading 1 million entries of the Database (Synchronous)

  Vs

  A Program P reading 1 million entries of the Database, but in batches (like 100k at a time) (Asynchronous)

Threads are highly Asynchronous ! They do their jobs whenever they want, if set free (spoiled kids) !

Reentrant functions

- A function is said to be reentrant if a context switch can be done from one thread to another in the system without any disruptive or ill results

- Reentrant function :
  - A function not using any global or static variables
  - A function not accessing data structures of application in Heap
  - A function which use appropriate synchronization techniques to protect global/shared data

```
node_t *
get_node_from_list(int a){

        node_t *node = search_node(list, a);
        return node;
}
```

Non-Reentrant function

```
void
delete_node_from_list(int a) {

        node_t *node = search_node(list, a);
        remove_node(node);
        free(node);
}
```

- Thread Synchronization Techniques need to be used to convert non-reentrant functions into reentrant ones

Should I Create my Application as Multi-threaded or Non-Muli-threaded ?

- This is the answer you would get by analyzing the end goal of your application and how you would going to design it

- An Application in which output of next step is present on prev step is a candidate of uni-threaded approach. Force Designing such a software system through multi-threaded approach would only give you spaghetti, untidy, prone to bug and low performing software

- Multi-threaded application would need to apply thread synchronization techniques – One thread causing other thread to wait only degrade application performance – Yes, Thread synch comes at a cost of performance

- Applications which deals with slow I/O (input from user) Or need to listen onto network sockets should launch a thread only for I/O, while in other thread application can process its logic

- Application which show natural parallelism should be multi-threaded or multi-process.
  - Eg, a TCP Server serving multiple TCP clients

- It is not necessary that Multi-threaded applications are always superior to equivalent uni-threaded applications in terms of speed or throughput, they will be superior only if multiple threads work on non-overlapping work and even more superior if threads execute on different CPUs

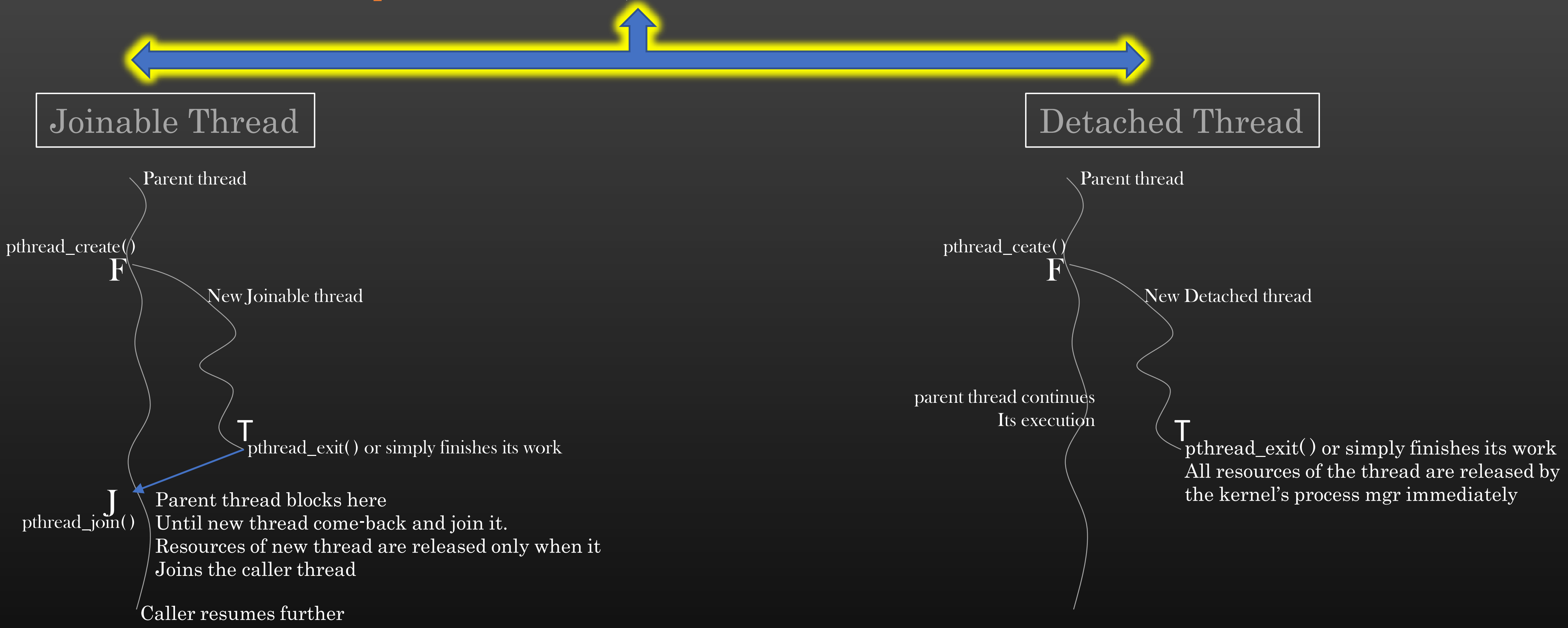
Should I Create my Application as Multi-threaded or Non-Muli-threaded ?

Drawing board for prev slide

Joinable
&
Detached
Threads

Joinable and Detached Thread

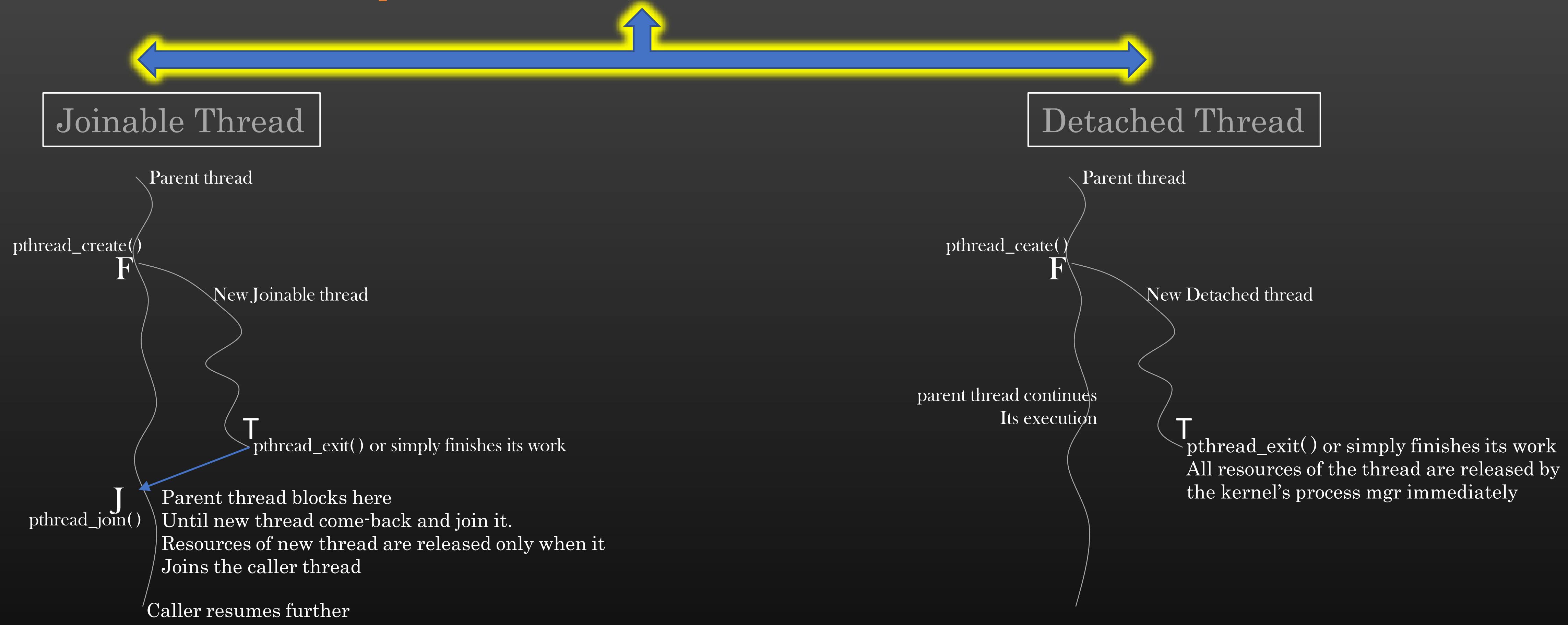
➢ A Thread when created (pthread_create), it can be created in one of the two modes :


Joinable Thread
Detached Thread
Parent thread
pthread_create()
F
New Joinable thread
T
pthread_exit() or simply finishes its work
J
pthread_join()
Parent thread blocks here
Until new thread come-back and join it.
Resources of new thread are released only when it
Joins the caller thread
Caller resumes further
Parent thread
pthread_ceate()
F
New Detached thread
parent thread continues
Its execution
T
pthread_exit() or simply finishes its work
All resources of the thread are released by
the kernel's process mgr immediately

- Resources of the joinable thread are not released until it joins the parent thread
- A Joinable thread can be converted into Detached while it is running or vice-versa
- By Default, thread runs on Joinable mode
- Joinable thread may return the result to Joinee thread

Joinable and Detached Thread

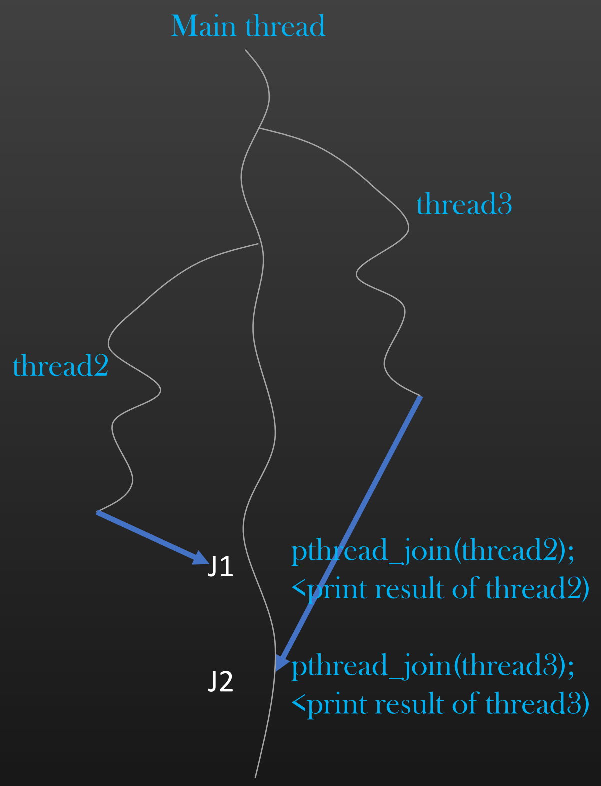
- A Thread when created (pthread_create), it can be created in one of the two modes :



Joinable Thread
Detached Thread
Parent thread
pthread_create()
F
New Joinable thread
T
pthread_exit() or simply finishes its work
J
pthread_join()
Parent thread blocks here
Until new thread come-back and join it.
Resources of new thread are released only when it
Joins the caller thread
Caller resumes further
Parent thread
pthread_ceate()
F
New Detached thread
parent thread continues
Its execution
T
pthread_exit() or simply finishes its work
All resources of the thread are released by
the kernel's process mgr immediately

- Resources of the Detached thread are released as soon as thread terminates
- A Detached thread can be converted into Joinable while it is running or vice-versa
- Detached thread do not return any result to Joining thread, they work and then die without telling anybody

Joinable and Detached Thread – Demo



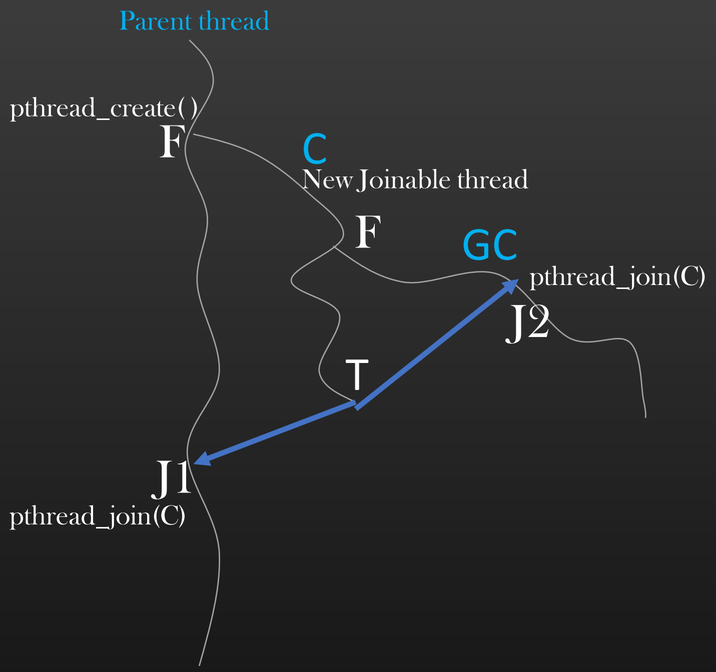
- Joinable threads must return the result in heap storage , not local variables !
- Parent thread blocked at join point J1 would stay blocked if thread3 terminates before thread2

What if thread3 terminates before thread2 ?

- ➢ Join signal shall be sent to parent thread
- ➢ But parent thread would stay blocked at join point J1
- ➢ When thread2 terminates, parent thread will get resume beyond join point J1
- ➢ Subsequently, parent thread surpass the join point J2 without getting blocked at all !

Joinable and Detached Thread → Whom to Join ?

- A Child Joinable thread upon termination join all the threads which are blocked on pthread_join on former's thread handle (pthread_t)

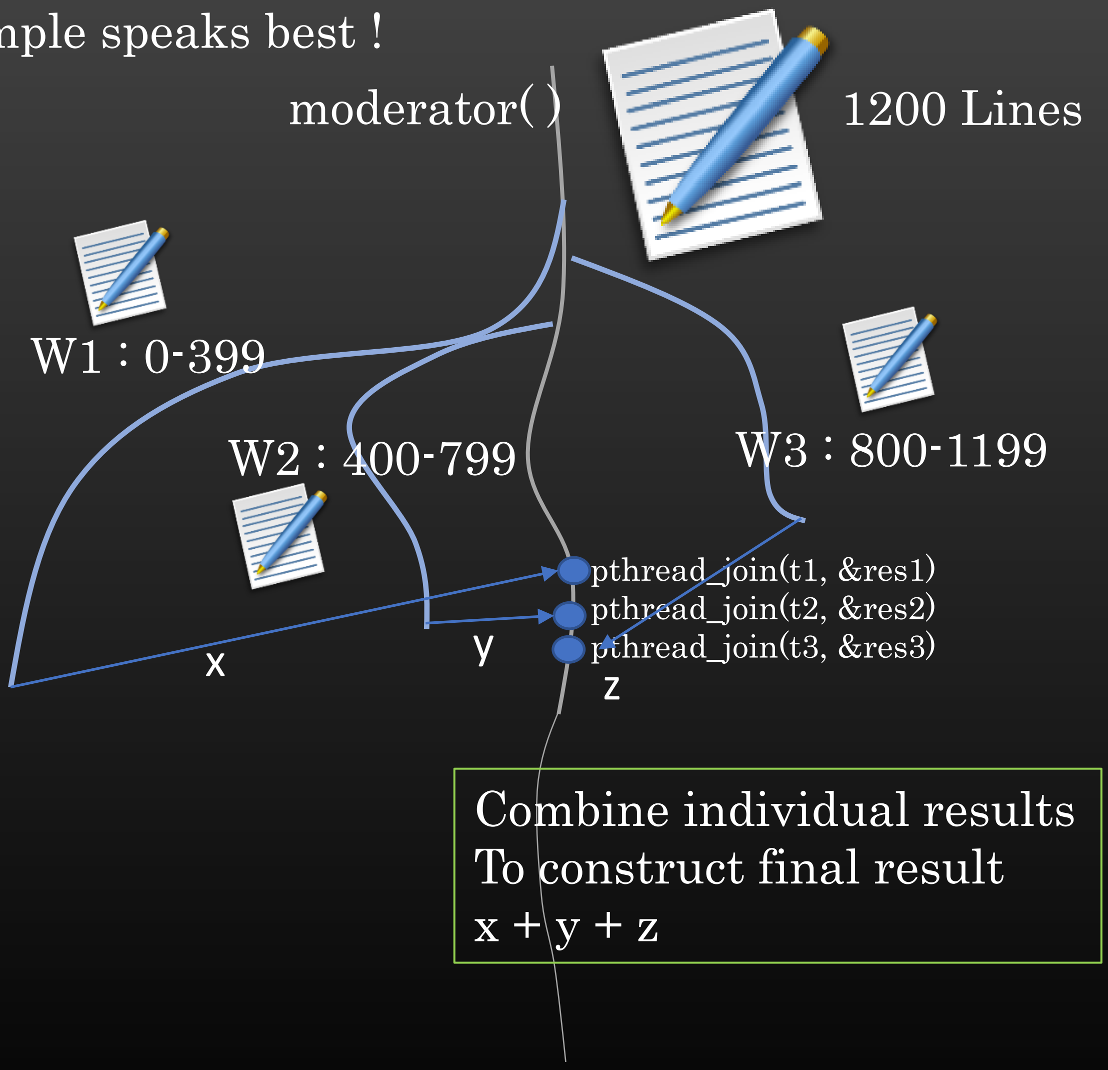


F – Fork Points
T – Termination Point
J – Join Point
C – Child Thread
GC – Grand Child Thread

- Any thread can invoke pthread_join( ) for any other Joinable thread, not just parent thread

Exercise – Implement Map-Reduce

- A Map-Reduce is a programming model based on Divide and Conquer Paradigm
- Example speaks best !



Q : Count no of words in a text file.

- Moderator thread splits and create worker threads
- Worker threads are called mappers
- Mappers work on non-shared data independently
- The thread who waits for all workers to finish is reducer thread
- Reducer thread build final result
- Moderator thread need not be reducer thread, they can be different

Joinable and Detached Thread → When to Create which type of thread ?

- Create thread T as Joinable When :
  - T is supposed to return some result to other threads
    - Eg : Map Reduce
  - When some threads are interested in being notified of other thread's termination

- Create thread T as Detached when :
  - No return result from T is expected
  - Nobody bothers about its death
  - T runs in infinite loop
    - Waiting for user input
    - Waiting for network pkt
    - TCP Server's Worker thread interacting with TCP Client

Inter Thread Communication

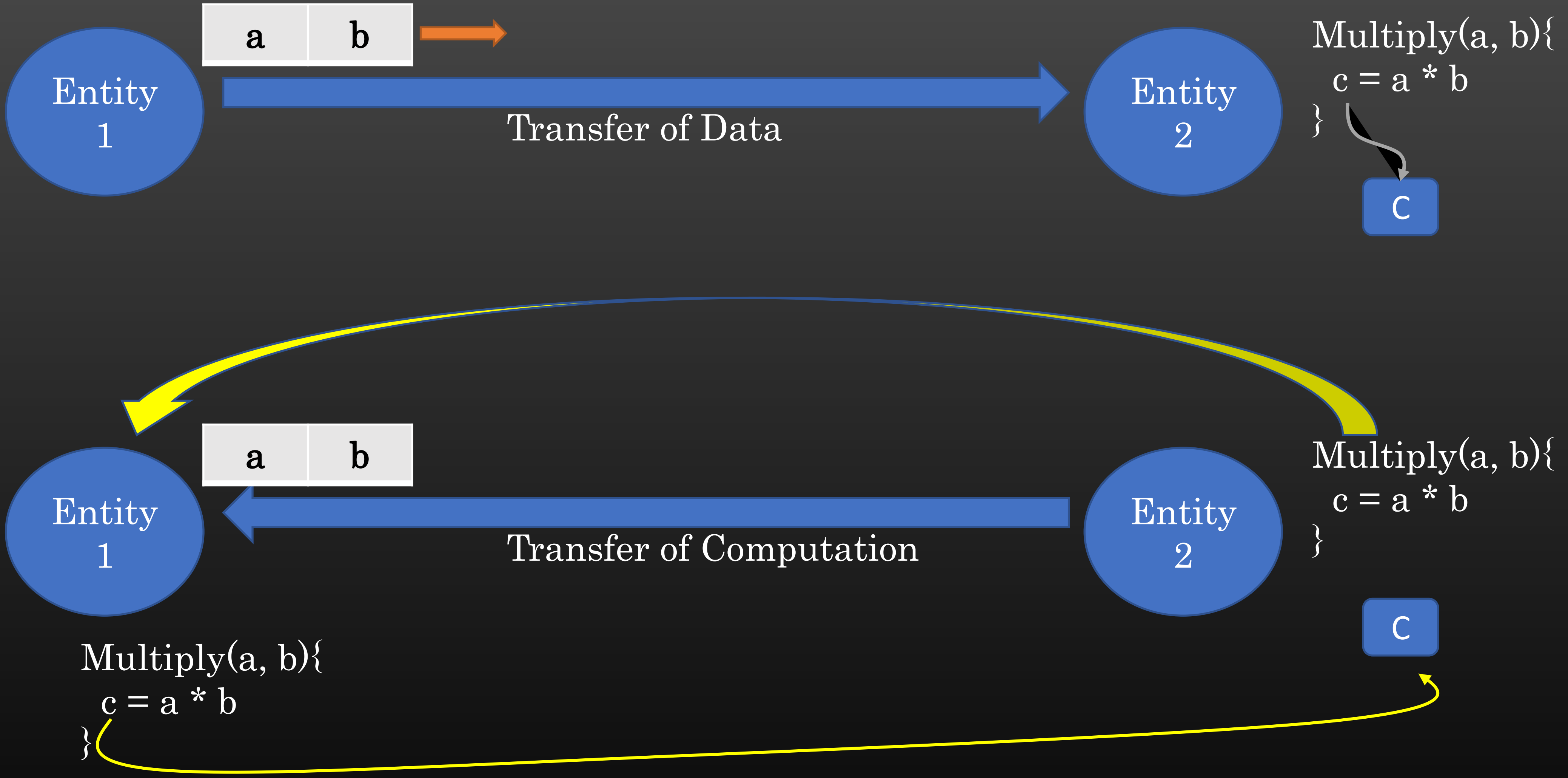
- We often feel the need to setup communication between threads (Exchange of Data)

- A big software system may have built as a multi-thread software and threads may require to exchange data with one another

- Famous IPC Techniques are usually used to setup data exchange between processes and technically nothing is stopping you from using it for threads
  - Sockets
  - Msg Queues
  - Pipes
  - Shared Memory

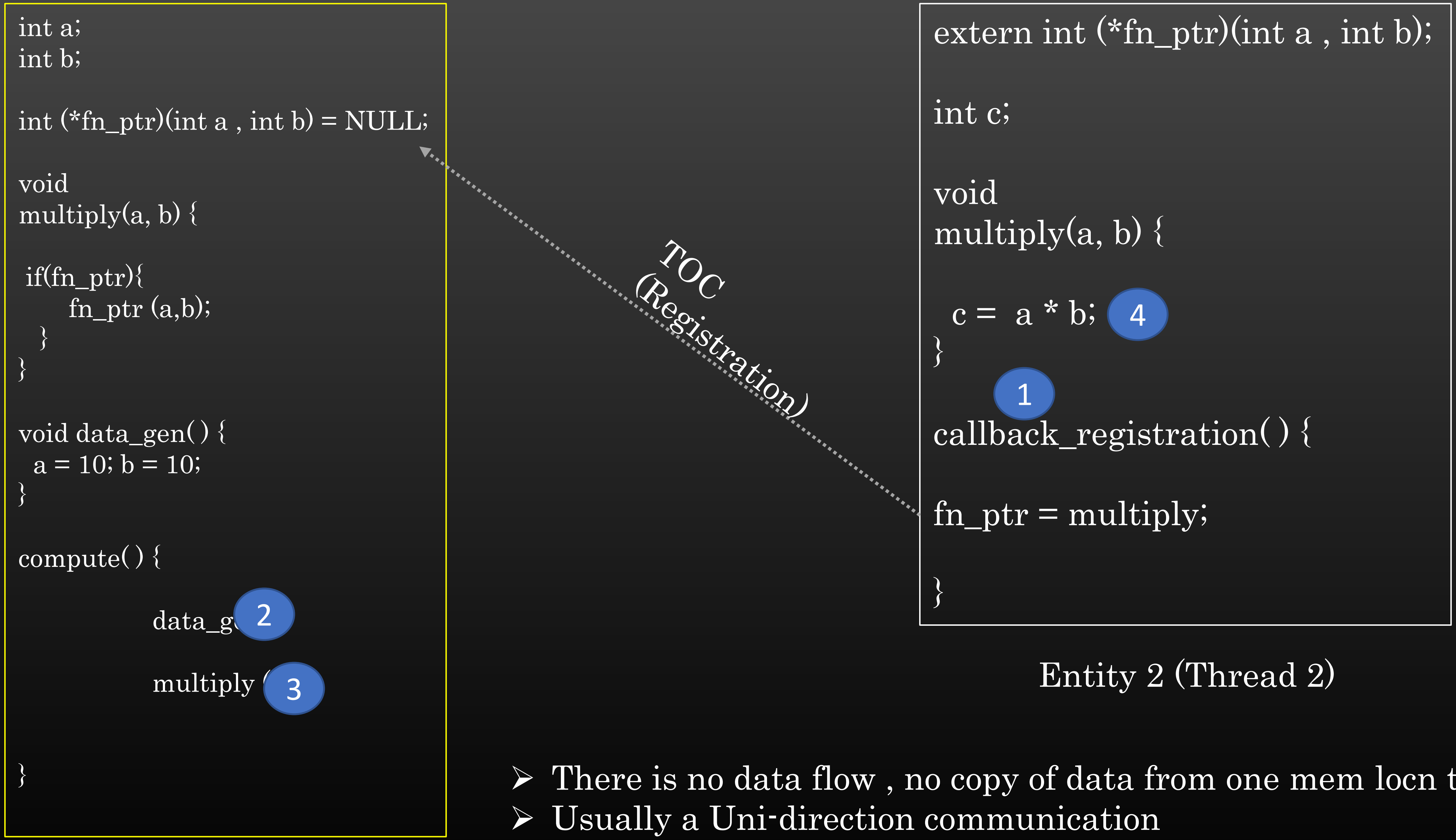
- But for inter-thread communication, IPC techniques is not the recommended way for data exchange
  - Communication between threads is preferred through callbacks/fn pointers
    - Very Fast
    - No Actual Transfer of data, but
    - Transfer of computation
    - No special attention required from Kernel, Completely run-in user space
    - Hence, no kernel resource need to be explicitly created

Inter Thread Communication -> Transfer of Data Vs Transfer of Computation



- In both cases, Result will be saved only on Entity 2's machine
- Transfer of Computation is feasible only when Entity 1 and 2 and in same Virtual Address Space
- TOC is no more than a fn call (through fn ptr)

Inter Thread Communication -> Transfer of Computation Example



```
int a;
int b;

int (*fn_ptr)(int a , int b) = NULL;

void
multiply(a, b) {

 if(fn_ptr){
      fn_ptr (a,b);
   }
}

void data_gen( ) {
  a = 10; b = 10;
}

compute( ) {

                  data_g

                  multiply (


}
```

Entity 1 (Thread 1)

```
extern int (*fn_ptr)(int a , int b);

int c;

void
multiply(a, b) {

  c =  a * b;
}

callback_registration( ) {

fn_ptr = multiply;

}
```

Entity 2 (Thread 2)

- ➢ There is no data flow , no copy of data from one mem locn to another
- ➢ Usually a Uni-direction communication
- ➢ Actual multiply( ) fn is owned by E2, but is executed by E1

Inter Thread Communication → Publisher Subscriber Model

TOC leads to a famous architectural Communication Model :

Publisher Subscriber Model

(Also Called Notification Chain)

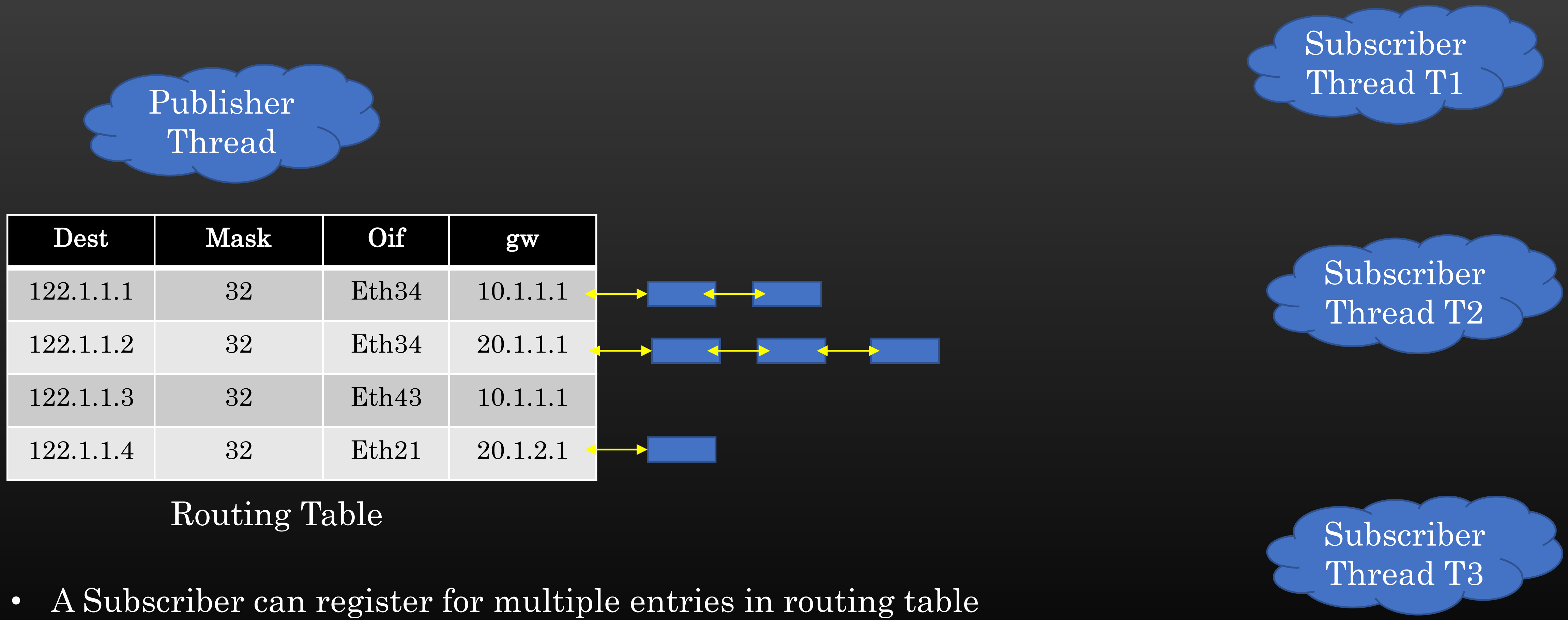
- This is the pattern of communication which is based on Transfer of communication
- The thread which generates the data is called Publisher
- The thread which owns the data processing function is called a Subscriber
- The activity of TOC is called Callback Registration
- The activity of invoking the fn through fn pointers by publisher is called Notification
- The next Section is all about Understanding and implementing NFC

Let us go on Vacations !

- Take a break, and let us go on Vacations …
- By Vacation I mean, lets learn something for fun !!
- Let us learn – Doubly Linked List !
  - What ?? You already know it ?
  - Then go to Appendix-A Section of this Course , and see you really learn the new way of using Doubly linked list – The Glue based Linked list !
  - It would be fun and more importantly -
  - We would be using this Doubly linked list implementation in our future assignments, so better learn it

Notification Chains → Implement Publisher Subscriber Example

- ➢ Now that, we have implemented a basic skeleton framework for Notification Chains, we shall now use it to implement an actual publisher Subscriber Model



Dest	Mask	Oif	gw
122.1.1.1	32	Eth34	10.1.1.1
122.1.1.2	32	Eth34	20.1.1.1
122.1.1.3	32	Eth43	10.1.1.1
122.1.1.4	32	Eth21	20.1.2.1

Routing Table

- A Subscriber can register for multiple entries in routing table
- A Subscriber can subscribe/unsubscribe at his will
- Code : MultithreadingBible/ThreadBasics/NFC

Notification Chains → Implement Publisher Subscriber Example

- To Begin with, We need a routing Table APIs to work with routing table

```
typedef struct rt_entry_keys_{

    char dest[16];
    char mask;
} rt_entry_keys_t;

typedef struct rt_entry_{

    /* A Structure which represents only the keys of the
     * Routing Table. */
    rt_entry_keys_t rt_entry_keys;

    char gw_ip[16];
    char oif[32];
    struct rt_entry_ *prev;
    struct rt_entry_ *next;
} rt_entry_t;

typedef struct rt_table_{

    rt_entry_t *head;
} rt_table_t;
```

Dest	Mask	Oif	gw
122.1.1.1	32	Eth34	10.1.1.1
122.1.1.2	32	Eth34	20.1.1.1
122.1.1.3	32	Eth43	10.1.1.1
122.1.1.4	32	Eth21	20.1.2.1

Routing Table

File : rt.h

Now we need to implement Create/Read/Update/Delete (CRUD) Operation on this Routing Table

Notification Chains → Implement Publisher Subscriber Example

```
void
rt_init_rt_table(rt_table_t *rt_table);
```

```
rt_entry_t *
rt_add_or_update_rt_entry(rt_table_t *rt_table,
    char *dest_ip, char mask, char *gw_ip, char *oif);
```

```
bool
rt_delete_rt_entry(rt_table_t *rt_table,
    char *dest_ip, char mask);
```

```
rt_entry_t *
rt_look_up_rt_entry(rt_table_t *rt_table,
                char *dest, char mask);
```

```
void
rt_dump_rt_table(rt_table_t *rt_table);
```

Dest	Mask	Oif	gw
122.1.1.1	32	Eth34	10.1.1.1
122.1.1.2	32	Eth34	20.1.1.1
122.1.1.3	32	Eth43	10.1.1.1
122.1.1.4	32	Eth21	20.1.2.1

Routing Table

#include “rt.h”

File : rt.c

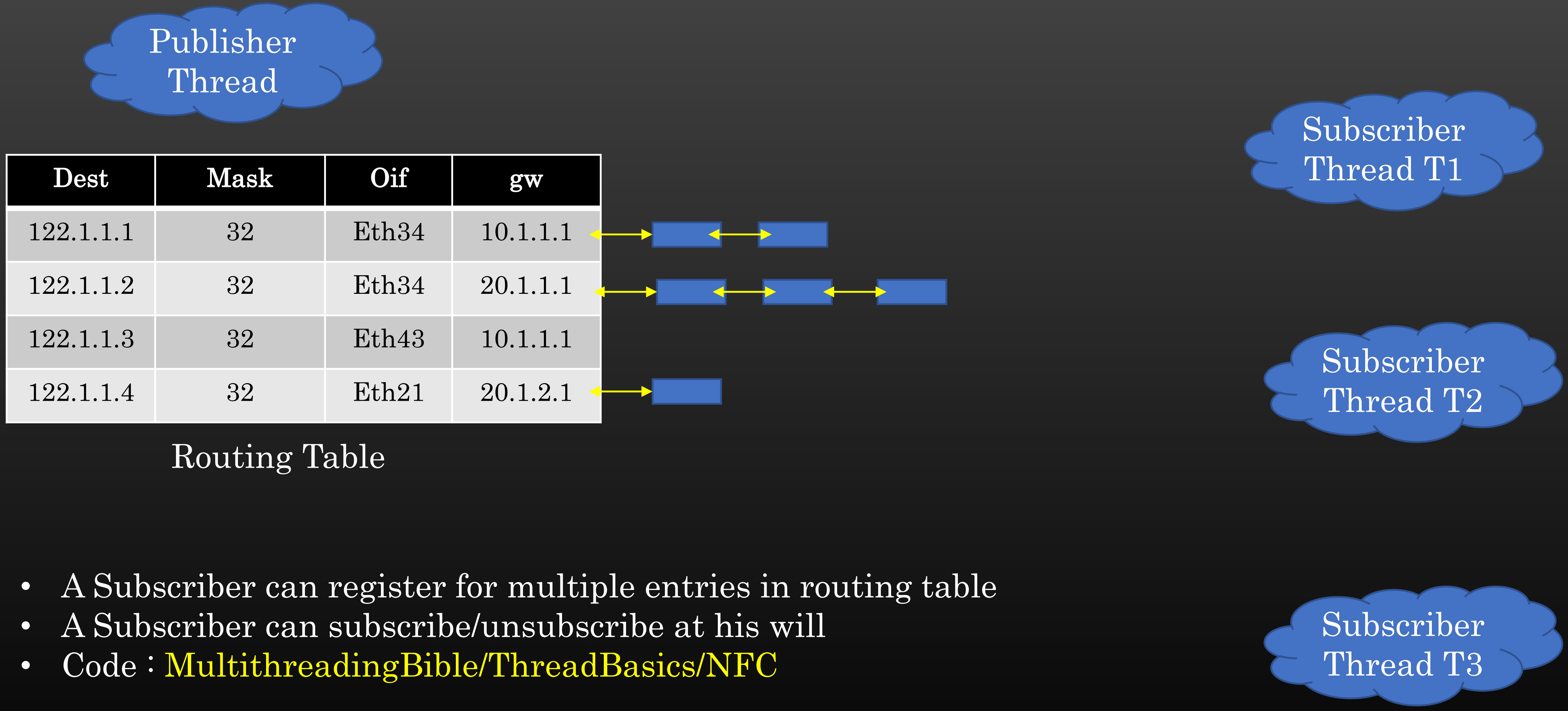
Implement CRUD APIs

Don’t bother anything about NFC, Just implement CRUD APIs over Routing Table

I have provided : rt_raw.h & rt_raw.c

Notification Chains → Implement Publisher Subscriber Example

➢ Setting up the project



Dest	Mask	Oif	gw
122.1.1.1	32	Eth34	10.1.1.1
122.1.1.2	32	Eth34	20.1.1.1
122.1.1.3	32	Eth43	10.1.1.1
122.1.1.4	32	Eth21	20.1.2.1

Routing Table

- A Subscriber can register for multiple entries in routing table
- A Subscriber can subscribe/unsubscribe at his will
- Code : MultithreadingBible/ThreadBasics/NFC

Notification Chains → Implement Publisher Subscriber Example

- Setting up the project

```
int
main(int argc, char **argv) {

    rt_init_rt_table(&publisher_rt_table);

    /* Create Subscriber threads */
    create_subscriber_thread(1);
    sleep(1);

    create_subscriber_thread(2);
    sleep(1);

    create_subscriber_thread(3);
    sleep(1);

    /* Create publisher thread*/
    create_publisher_thread();
    printf("Publisher thread created\n");

    main_menu( );
    pthread_exit(0);
    return 0;
}
```

rtm_publisher.c

```
static void
test_cb(void *arg, size_t arg_size,
        nfc_op_t nfc_op_code,
        uint32_t client_id) {

                /* Thread is notified by the publisher */
}



void *subscriber_thread_fn(void *arg) {

                /* Register for some RT table entries */
}


void
create_subscriber_thread(uint32_t client_id) {

                pthread_create( . . , subscriber_thread_fn, . .. )
}
```

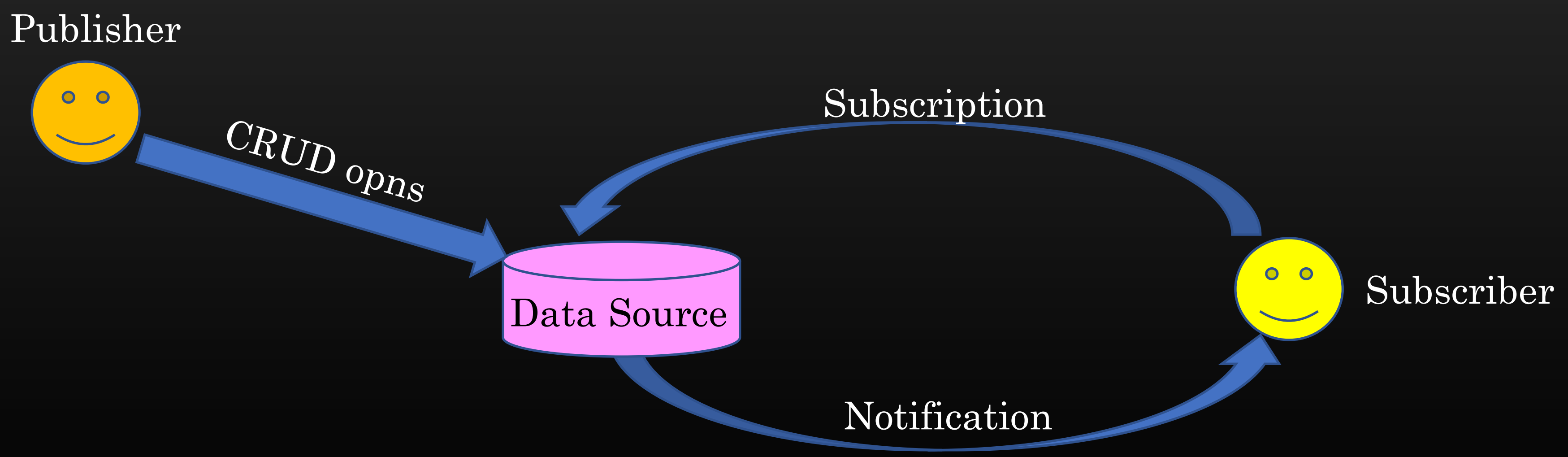
threaded_subsciber.c

Subscription

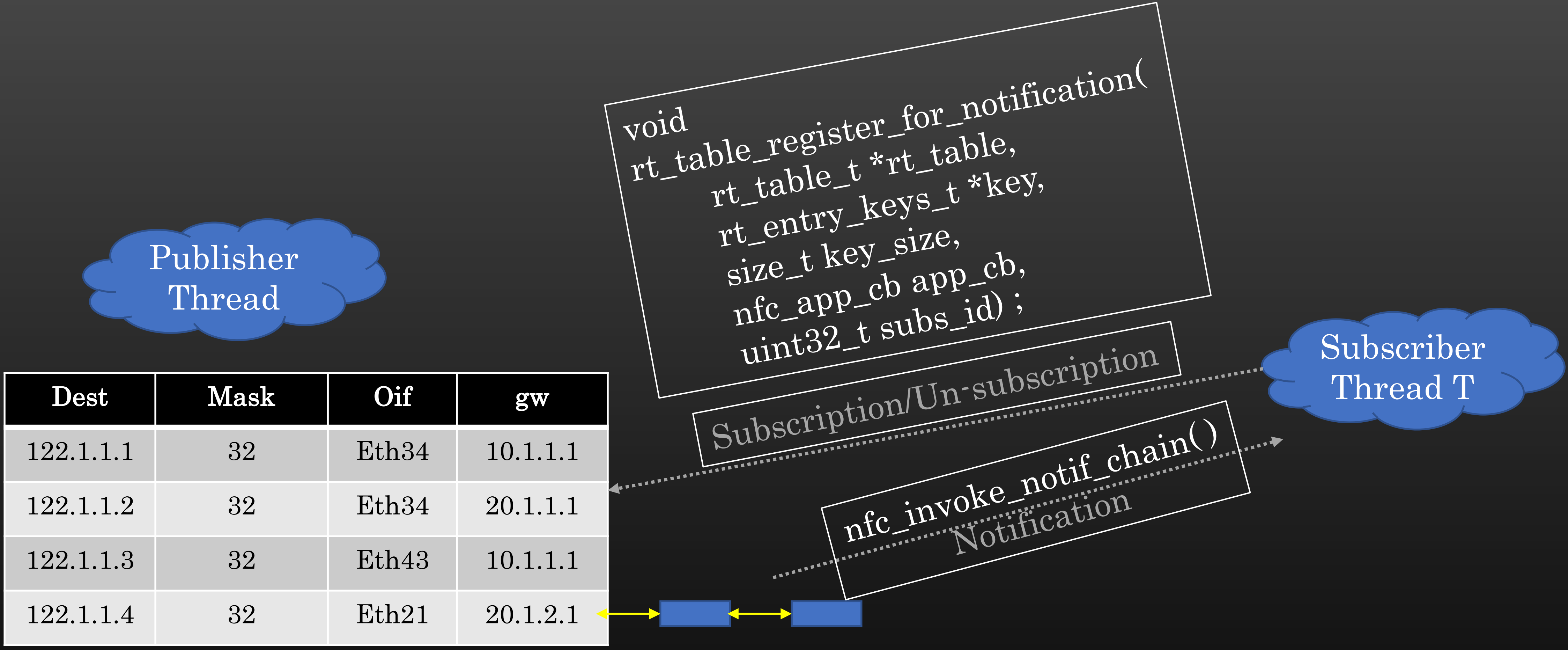
Allow Subscribers to (un)subscribe for the entry of interest for notifications

Notification

Notify Subscribers whenever the entry in the data Source is updated by the publisher



Notification Chains → Implementing Notification and Subscription



```
void
rt_table_register_for_notification(
        rt_table_t *rt_table,
        rt_entry_keys_t *key,
        size_t key_size,
        nfc_app_cb app_cb,
        uint32_t subs_id) ;
```

Dest	Mask	Oif	gw
122.1.1.1	32	Eth34	10.1.1.1
122.1.1.2	32	Eth34	20.1.1.1
122.1.1.3	32	Eth43	10.1.1.1
122.1.1.4	32	Eth21	20.1.2.1

Routing Table

Pizza Delivery Problem

- Usually when Pizza Delivery boy delivers pizza to your house, he handover the pizza and move on to the next house

- He don't bother when you will be going to consume the pizza, or consume it at all

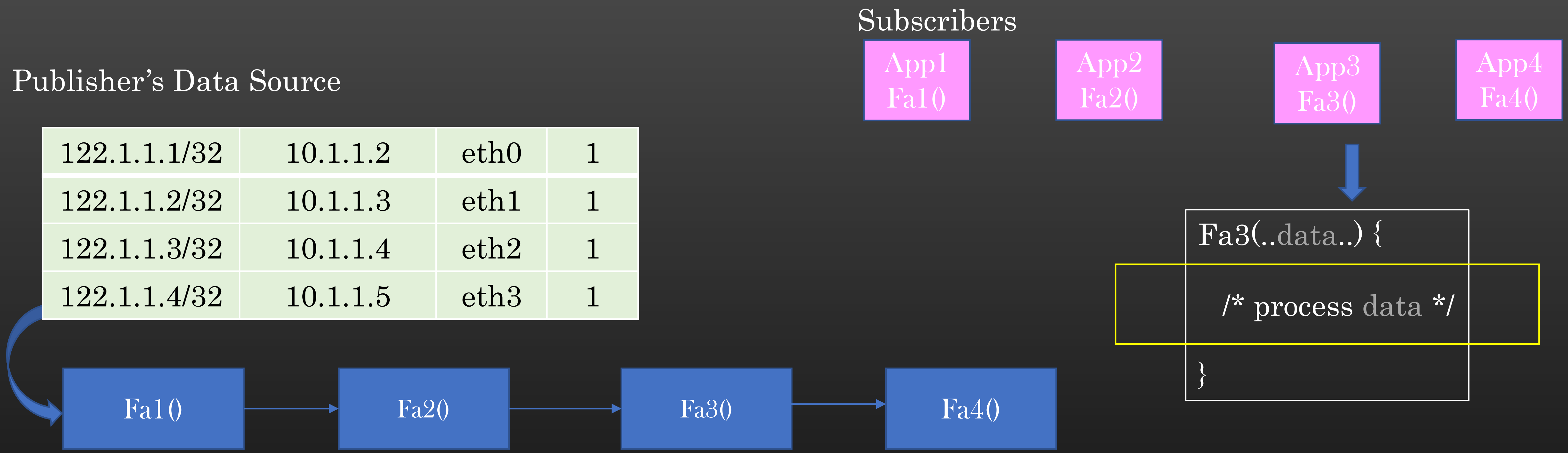
- Now, let us put restriction on pizza delivery boy –

  - Constraint :
    He should not move onto the next delivery unless the current customer has enjoyed the pizza meal

I know, the constraint doesn't make much sense, but this is what our NFC communication Model is infected from !

The Publisher (or data source) do not notify the update to the next subscriber in the notification chain until the current subscriber has processed the data

Pizza Delivery Problem



122.1.1.1/32	10.1.1.2	eth0	1
122.1.1.2/32	10.1.1.3	eth1	1
122.1.1.3/32	10.1.1.4	eth2	1
122.1.1.4/32	10.1.1.5	eth3	1

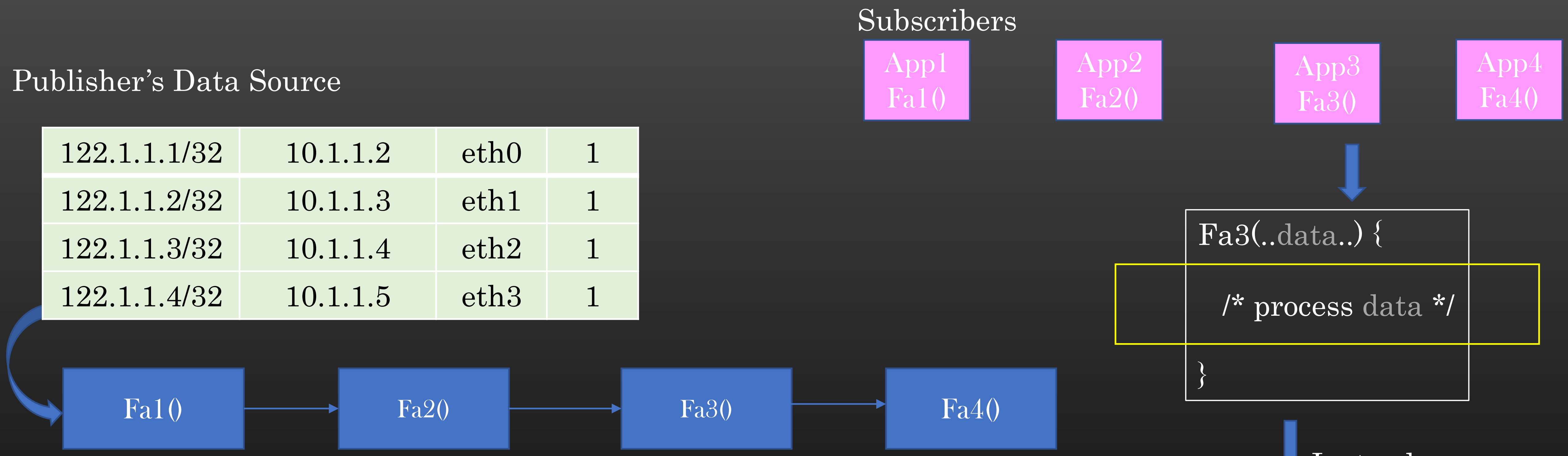
```
Fa3(..data..) {

   /* process data */

}
```

- ➢ Sequence of function invocations : Fa1( ) → Fa2( ) → Fa3( ) → Fa4( )
- ➢ What if the subscriber do heavy processing in their notification callback fns (Notification Consumption)
- ➢ The subscribers would get delayed notification, unnecessarily the publisher has to wait until the subscriber consume the *notification*
- ➢ Solution : Subscriber must create new thread and then consume the notification

Pizza Delivery Problem --> Solution (Work Off-loading)

Publisher's Data Source

122.1.1.1/32	10.1.1.2	eth0	1
122.1.1.2/32	10.1.1.3	eth1	1
122.1.1.3/32	10.1.1.4	eth2	1
122.1.1.4/32	10.1.1.5	eth3	1



```
Fa3(..data..) {

   /* process data */

}
```

Instead

```
Fa3(..data..) {

  pthread_create(&new_thread,
                        0, process_fn, data

}
```

- ➢ process_fn executes in the context of the subscriber new thread and not publisher's thread
- ➢ Fa3() returns immediately after doing a very small amount of work
- ➢ Processing work is offloaded from publisher thread to Subscriber's new thread

Thread Cancellation

Thread Cancellation

- Many times, a thread which is in the state of execution needs to be cancelled

- Ex : You may want to cancel on-going search operation
  : you may want to stop sending periodic packets
  : you may want to stop downloading the file

So, Thread Cancellation is Quite common, right !!

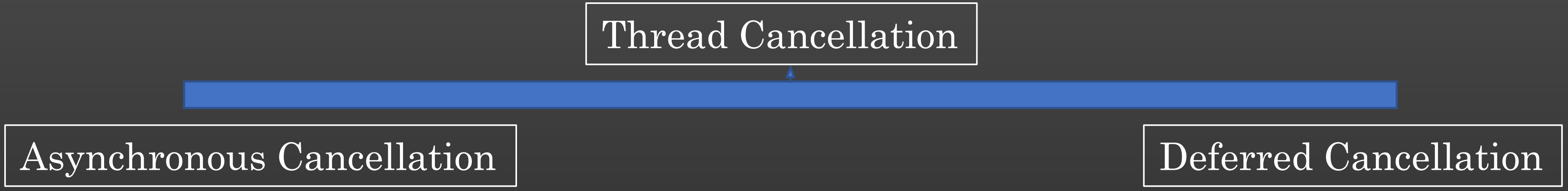
Thread Cancellation is a lot like telling a human to stop something they are doing

Any thread of the process can choose to cancel the other thread

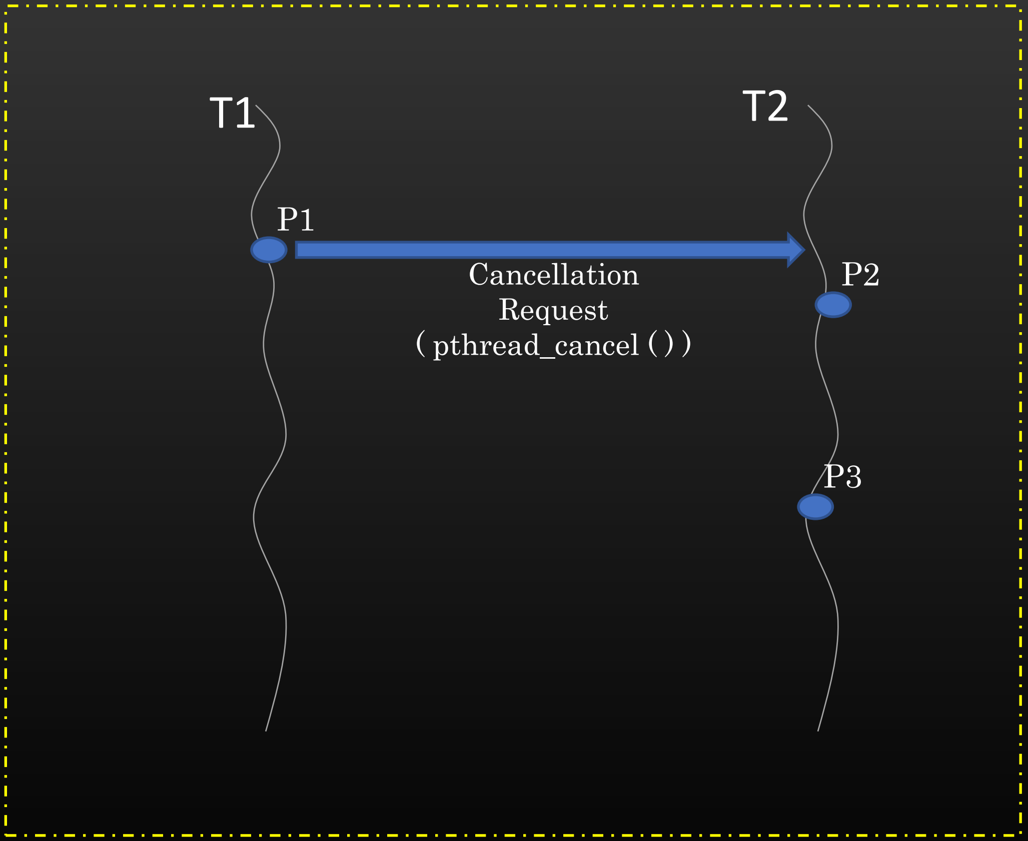
Once the thread is cancelled, it is terminated (thread cease to exist)

Let us dive deep into thread cancellation . . .

Thread Cancellation



t = 0



- The Cancellation Request (CR) is Queued by OS
- Having Recvd the CR, the OS is now looking an opportunity to terminate thread T2
- OS may or may not terminate the thread instantly (Asynchronous)
- At t = t + Δt , OS delivers the cancel signal to T2, which results in T2 termination instantly
- When P2 is terminated, it may be executing at some point P3, not exactly P2

Thread Cancellation -> Asynchronous Cancellation

- Let us See thread Cancellation in action ..

Code : MultithreadingBible/ThreadBasics/ThreadCancellation
file : master_slave1.c
soln : master_slave1_async_cancellation.c

Program to create 5 threads, and all 5 threads write a string into file thread_x.txt, where x is thread id.

string to write – *I am thread <thread id>*

Thread Cancellation -> Asynchronous Cancellation Problems

- Problem with Asynchronous Cancellation
  - What will happen if you are driving a car, and suddenly you are asked to leave the steering wheel ?
  - Accident !

1
- Resource Leaking
  - Not closing the open file descriptor/sockets
  - Not free-ing the memory

2
- Cause Invariants
  - Data Structure Corruption

3
- Deadlocks
  - Mutexes left in locked state for forever

Thread Cancellation -> Asynchronous Cancellation

- Problem with Asynchronous Cancellation
  - What will happen if you are driving a car, and suddenly you are asked to leave the steering wheel ?
  - Accident !

1
- Resource Leaking
  - Not closing the open file descriptor/sockets
  - Not free-ing the memory

← Thread must be given one last chance
To clean up resource before it is
terminated – cleanup handlers

2
- Cause Invariants
  - Data Structure Corruption

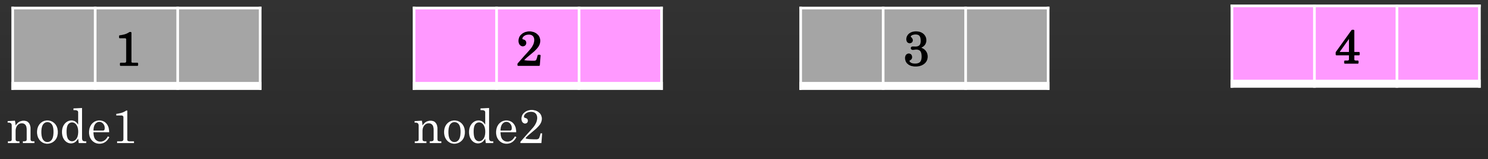
← Thread must cancel at specific points in
Execution flow, and not just anywhere randomly
Cancellation points (only Deferred Cancellation)

3
- Deadlocks
  - Mutexes left in locked state for forever

Thread Cancellation -> Invariants Problem

- Threads when cancelled abruptly may lead to the problem of invariants which may in-turn lead to Data structure corruption, memory leak, wrong computation etc

- Invariants means – A data structure in inconsistent state



```
. . .
node1->next = node2->next;
node2->next->prev = node1
node2->prev = NULL;
node2->next = NULL;
. . .
```

- Operation updating the data structures must not be left incomplete on thread cancellation
- Thread must not get cancelled while it is updating the data structures

Thread Cancellation -> Invariants Problem

Ex :

Cancelling the thread removing/adding a node in a Balanced Tree (red-black/AVL trees)

Cancelling the thread which is in the process of executing system calls

> Abruptly terminating the system call may lead to kernel corruption/variants in kernel space

Thread Cancellation -> Dead lock Problems

What if the thread has locked the mutex M, and then it is cancelled !

> Mutex would stay locked by the non-existing thread

> Any other live thread would enter into deadlock if try to lock the same mutex

> We need to ensure that when thread is cancelled, it must not have any mutex held in locked state

Thread Cancellation -> Preventing Resource Leaking

- If we could give thread being cancelled one last chance to clean up his mess, then resource leaking could be handled
- POSIX standards provide the concept of *Thread Clean up handlers*
- *Thread clean up handlers* are functions which are invoked just before the thread is about to cancelled

void (*cleanup_handler)(void *);

- When clean up handler function returns, thread is cancelled immediately
- Let us extend our example . .

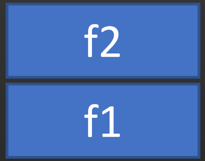
File : MultithreadingBible/ThreadBasics/ThreadCancellation/master_slave1_async_cancellation.c

File : MultithreadingBible/ThreadBasics/ThreadCancellation/master_slave1_async_cancellation_cleanup_handlers.c

Thread Cancellation -> Preventing Resource Leaking -> clean up handlers

- Thread clean up handlers are specified in the form of stack (stack of functions)
- Clean up handlers are invoked from top of the stack to bottom of the stack

```
thread_fn( ) {
            pthread_cleanup_push(f1, arg);
            pthread_cleanup_push(f2, arg);

            . . .
            . . .
            . . .

            pthread_cleanup_pop(0);
            pthread_cleanup_pop(0);
}
```



Thread's Cancellation cleanup stack

← Pop the cleanup handlers from stack if thread_fn do not Cancel and execute to completion successfully

Thread Cancellation -> Preventing Resource Leaking -> clean up handlers

- Push is replaced by some ‘ { ‘ and some inter mediate code at compile time
- Pop is replaced by some by some inter mediate code and ‘ } ‘ at compile time

```
thread_fn( ) {
        pthread_cleanup_push(f1, arg);
        pthread_cleanup_push(f2, arg);

        . . .
        . . .
        . . .

        pthread_cleanup_pop(n);
        pthread_cleanup_pop(n);
}
```

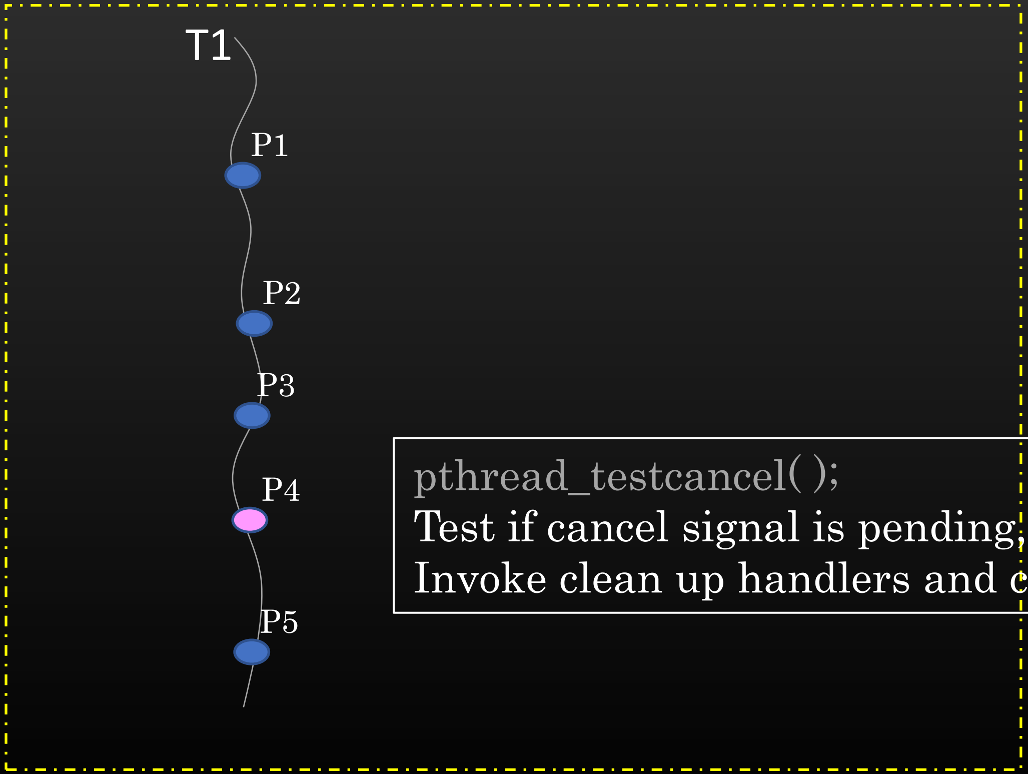
```
thread_fn( ) {
        {
                some code is inserted by the compiler

        {
                some code is inserted by the compiler

        . . .
        . . .
        . . .

                some code is inserted by the compiler
        }
                some code is inserted by the compiler
        }
}
```

Thread Cancellation -> Preventing Resource Leaking -> clean up handlers

- Ensure, parenthesis are balanced (by imaging push as “{“ and pop as “}”)
- If n = 0 is passed in *pthread_cleanup_pop(n)* clean up fn is popped out from stack
- If n = ~0 is passed in *pthread_cleanup_pop(n)* clean up fn is popped out from stack and Invoked
- Cleanup fns are also invoked when thread terminates using pthread_exit( )
- Cleanup fns are not invoked when thread terminates by virtue of return statement

Thread Cancellation → Deferred Cancellation

- Deferred Cancellation allow the programmer to control as to which points in the execution flow of the thread, the thread is allowed to cancelled,
- Contrary to Async Cancellation where thread could be cancelled at any point in its execution flow
- Deferred Cancellation is used to handle the problem of Invariants
- Such points are called Cancellation Points
- Cancel Signal can be delivered by the kernel to the thread being cancelled, but processed only at cancellation points of the executing thread
- It is the programmer's responsibility to choose CP wisely such that when thread is cancelled at CP no Variants/Leak/Deadlocks must occur
- Let's Code , Files :
- master_slave1_deferred_cancellation.c



pthread_testcancel( );
Test if cancel signal is pending, if yes,
Invoke clean up handlers and cancel thread

Thread Cancellation → Comparison

Asynchronous Cancellation	Deferred Cancellation
Target thread cancels at any point in its execution flow, not under Dev control	Target thread cancels only at cancellation points
Can’t handle Invariants	Can handle Variants well
Almost impractical, Not recommended to use	Recommended

Thread Cancellation → Returning Value

- Can a thread that is be being cancelled return value on cancellation to all threads waiting on pthread_join ( ) ?

Returning value when thread is cancelled (pg 43)

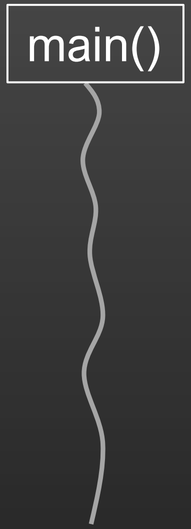
pthread_cancel

Returning value when thread is cancelled (pg 43)

Returning a Value from Thread

Returning value via thread_exit
Normal thread completion
Returning value when thread is cancelled (pg 43)

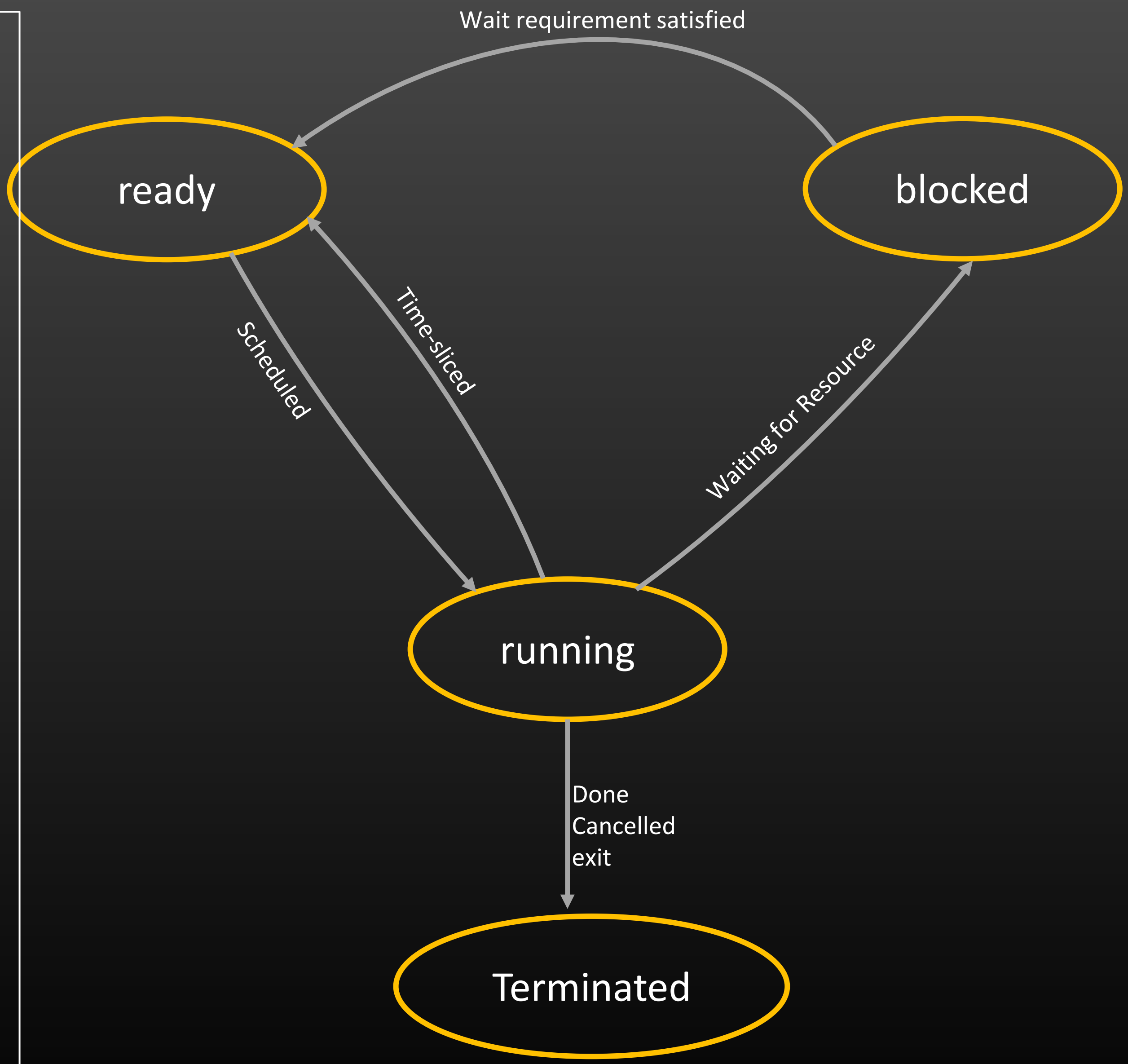
Which thread to create and why – Joinable or Detached ?



	When to Create Joinable Thread	When to Create Detached Thread
1	When Parent threads needs a return value from child thread	When parent thread do not need any return value from child thread
2	When parent thread do not wat to proceed further until child thread completes its work	When parent thread and child thread can run independently and do their job
3	When parent thread need to be notified about completion of child thread	When parent thread do not worry about the death of child thread
4	For computationally non-intensive work (which takes less time)	For computationally intensive work, but result is not needed by parent thread
5	Ex : Parent thread launching a child thread to compute the SHA of some big file	Ex : Writing a lots of data into log files on disk Ex : Launching Child threads as TCP Servers to entertain client requests

Thread Life Cycle

- Ready State:
  - Thread is ready for CPU allocation
  - Immediately after pthread_create()
  - Thread de-allocated CPU due to normal context switching (timeslicing)
- Blocked State:
  - Blocked by blocked-Mutex
  - Blocked by Condition Variable
  - Blocked on I/O
  - Page fault
  - (Forced Context Switching)
- Running State:
  - When thread is actually executing the instructions
  - Allocated the CPU
- Terminated State:
  - Thread finished its thread fn
  - Thread is cancelled by other thread
  - Thread invoked pthread_exit(0)
  - All resources are released

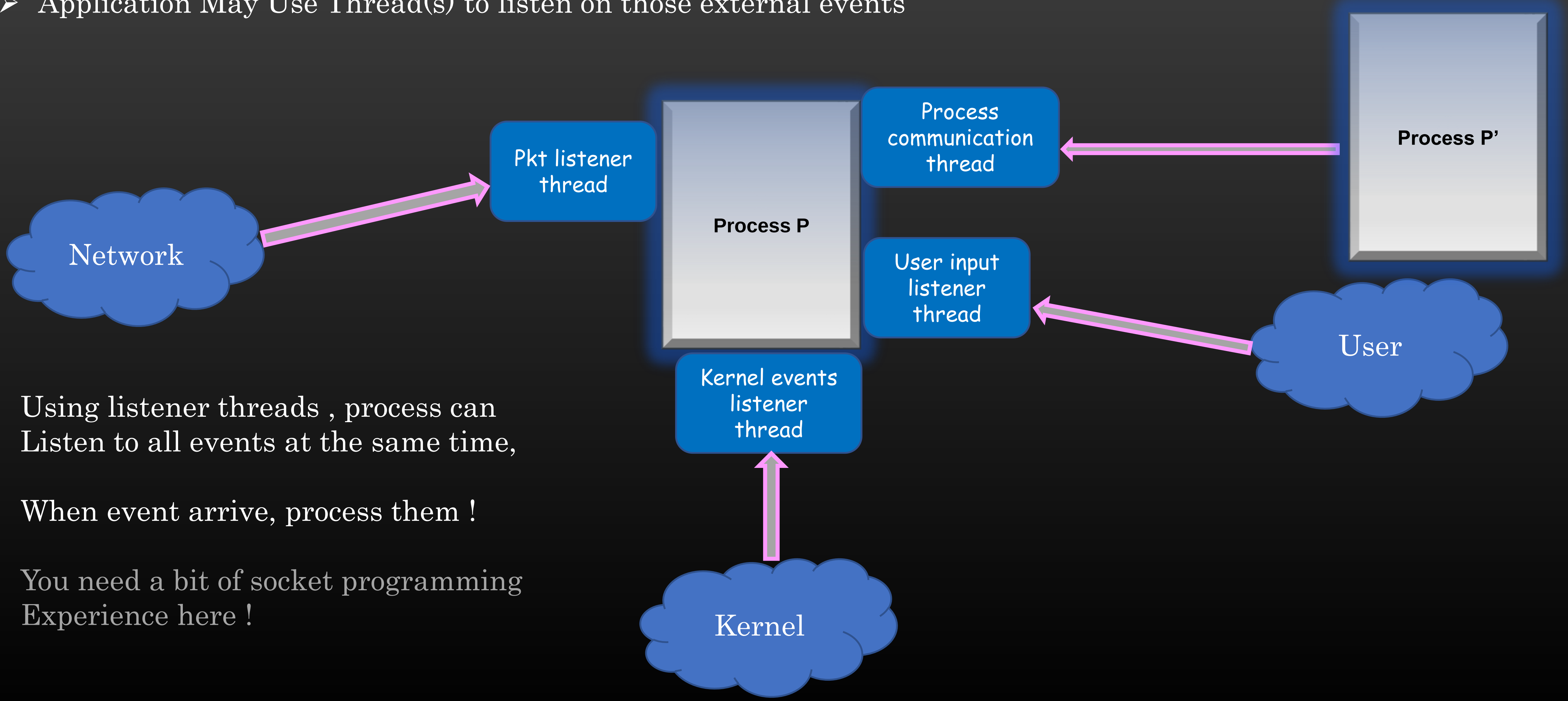


Inter Thread Communication

Listener Threads

Listener Threads

- It is a common scenario that an application needs to constantly listen to external events
- Those external events can arrive anytime, and application needs to process those events
- Application May Use Thread(s) to listen on those external events


Process P'
Process communication thread
Pkt listener thread
Process P
Network
User input listener thread
User
Kernel events listener thread
Kernel

Using listener threads , process can
Listen to all events at the same time,

When event arrive, process them !

You need a bit of socket programming
Experience here !

Listener Threads

- We shall be going to implement an application which can listen on :

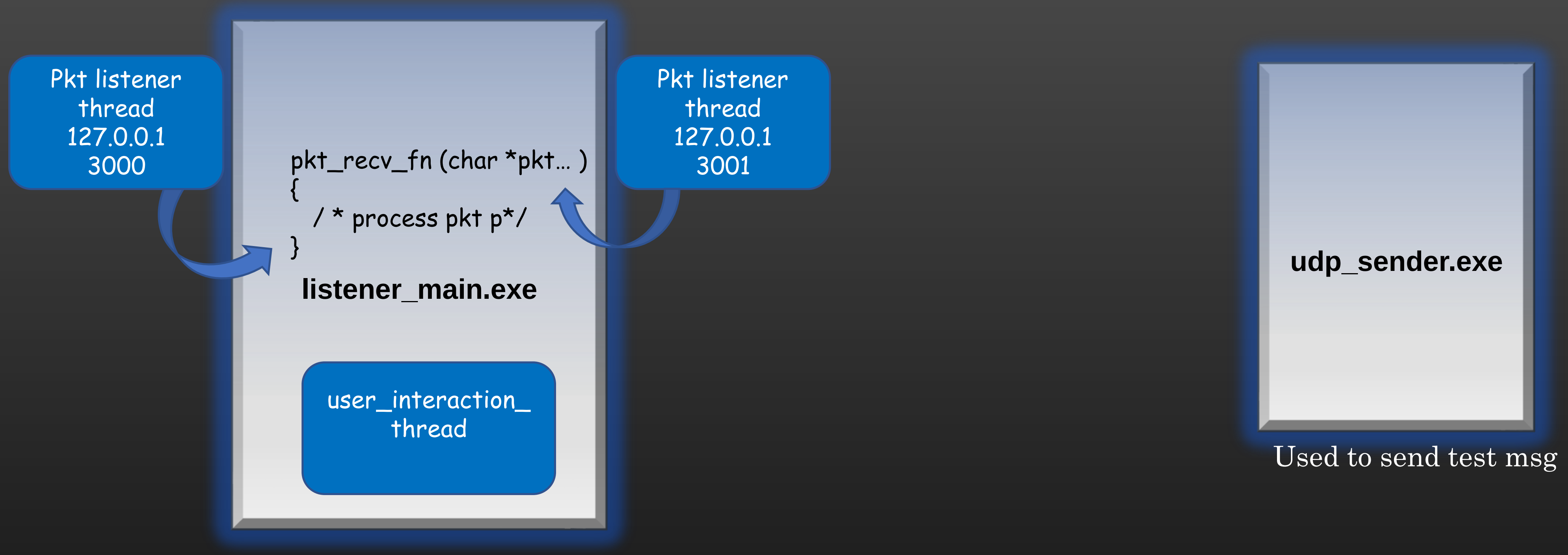
  1. Multiple Network UDP sockets

  2. Listen on User Input

- The Technique to listen to extern event is same, irrespective of the type of event
  - Kernel Events
  - Events from other process

Codes : MultithreadingBible/EventListeners

Listener Threads → Multiple Events Listening


Pkt listener thread 127.0.0.1 3000
pkt_recv_fn (char *pkt…) { / * process pkt p*/ }
Pkt listener thread 127.0.0.1 3001
listener_main.exe
user_interaction_ thread
udp_sender.exe
Used to send test msg

- I have provided the library : network_utils.h / network_utils.c

  API : To recv : udp_server_create_and_start ( )
  API : To Send : send_udp_msg( ) used by udp_sender.exe

Let us code up the listener_main.exe and see things in action . . .
This section needs that you have some basic background in socket programming …

Cancellation of Listener Threads

- In our Multi-Listening Application example, our listener threads are none but Socket Monitoring threads
- Out Listener threads stay blocked on blocking system call, in this case, recvfrom( )
- Other Examples of blocking calls :
  - sleep
  - scanf
  - pthread_mutex_wait
  - select/epoll
  - recvfrom/recv/read/recvmsg
  - and many more ..
- When threads get blocked, they still respond to thread cancellation
- It means – the most blocking calls provide by glibC are also Cancellation Points
  - Were they not CP, our listener threads would not be able to cancelled when they are in blocked state
  - Complete list of other blocking glibC calls is listed in next slide

Thread Synchronization

Thread Synchronization

- The hardest and most important aspect of Multi-threading is – Thread Synchronization

- Thread Synchronization is required in multi-threaded programs whenever multi-threads compete to perform conflicting operations (Read-Write Or Write-Write) on a shared resource

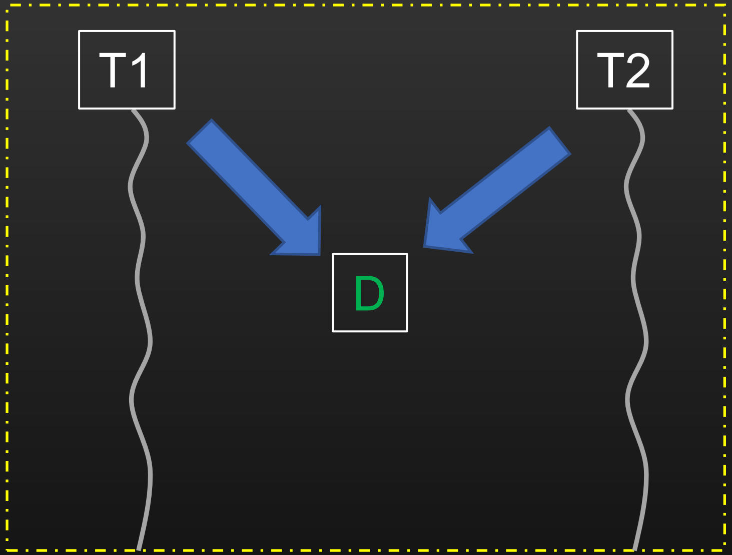
  - Shared Resources :

    - Heap Data Structures

    - Global Variables

    - File Descriptors (opened Files, Sockets)

    - Receiving Data from External Sources via multiple inlets



- Let us first try to understand the problems we would have if we don't have Thread-Synchronization

Thread Synchronization → Data Inconsistency/Corruption

- Consider a multi-threaded process P having two threads T1 and T2
- Also, consider a Process P maintains a linked list of integers

- Thread T1 and T2 are scheduled on the CPU(s) in any order – do not assume determinism/pattern !

Thread T1

```
node_t *
get_node_from_list(int a){
I1
I2          node_t *node = search_node(list, a);
I3
I4          return node;
}
```

Thread T2

```
void
delete_node_from_list(int a) {
I5
I6          node_t *node = search_node(list, a);
I7
I8          remove_node(node);
I9
I10         free(node);
}
```

- Concurrent access of shared data structures between Multiple-Threads opens a window during which data is in-consistent – Root Cause is Concurrency

- The region in code where shared data is accessed by multiple threads are called Critical Sections
- Goal : Concurrency + Data Consistency

Critical Section

```
foo() {
. . .
. . .

global_var1++;
global_var2++;

. . .
. . .
}
```

☞ Code Excerpt accessing the shared Data are critical sections

☞ Shared Data –
- ☞ Global Variables
- ☞ Heap Data Structures
- ☞ Static Variables

Rule of Thumb : Critical Sections Must be be executed by Concurrent threads but by one and only one thread at a time
> Unexpected behavior
> Segment Fault
> Data Corruption
> Any abnormal behavior

➢ Thread Synchronization – Identify CS and apply several techniques to prevent unprotected Concurrent access to shared resources by several threads of the process

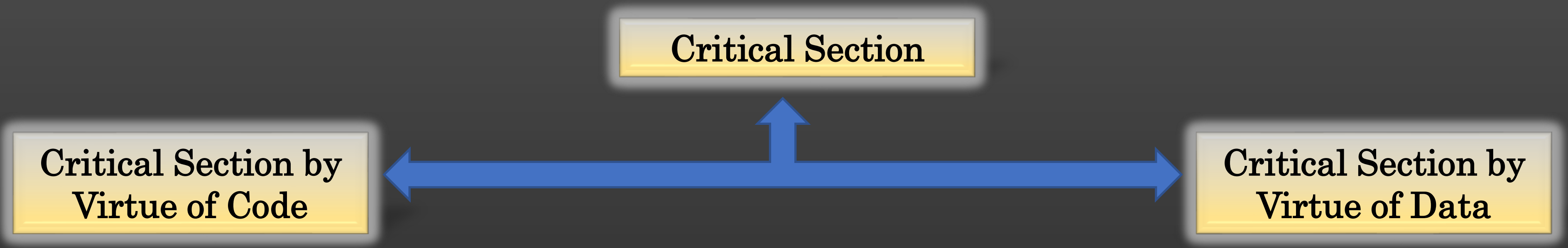
Critical Sections

```
foo() {

          static int i = 1;

          I1 : i = i + 1;
          I2 : i = i + 2;
          I3 : print i ;

}
```

☞ If the function foo( ) is executed by three concurrent threads, what are possible outputs of the program ?

- Different outputs based on sequence of threads execution
- If i is changed to local variable, there is no Critical Section
- More Examples of CS :
  - Two or more threads trying to update same database
  - Two or more threads trying to read/write into same memory buffers
- CS are critical in the sense, if they are allowed to be accessed by multiple threads without deploying thread synchronization techniques, the application sooner, or later, is bound to fail

Thread Synchronization → Critical Section


Critical Section
Critical Section by Virtue of Code
Critical Section by Virtue of Data

```
foo() {

        static int i = 1;

        I1 : i = i + 1;
        I2 : i = i + 2;
        I3 : print i ;

}
```

Seeing at the code Visually you can Find out CS !

```
void
delete_node_from_list(list lst, int a) {
I5
I6        node_t *node = search_node( lst, a);
I7
I8        remove_node(node);
I9
I10       free(node);
}
```

CRUD Operations on Data Structures themselves are CS

Cannot be find out by reading the code visually

Question

```
void
pkt_receive (char *pkt, int pkt_size) {


        memset (global_buffer, 0, sizeof(global_buffer));

        memcpy (global_buffer, pkt, pkt_size);

        forward_pkt (global_buffer, pkt_size);

}
```

- Identify the Critical Section in the code ?
- If the Critical Section by Virtue of Code or by Virtue of Data ?
- What are implications of executing the above code in the context of multiple threads in a thread unsafe application ?

Thread Synchronization → Mutexes

- Mutexes are Thread Synchronization constructs/tools which provide Mutual Exclusivity while accessing a critical Section by multiple concurrent threads

Analogy :

- Mutexes are like Keys to the locker, whoever wants to access the locker need to have keys
- Whoever do not have keys, cannot access locker and has to wait
- Whoever is accessing the locker, need to handover the keys when done accessing the locker
    - Locker = Critical Section
    - Whoever = Threads
    - Keys = Mutexes

- Half of the work is done, when you are able to map Thread Synchronization Concepts to real world scenarios

Thread Synchronization → How Mutexes Work ?

- Grant Access to C.S to only one and one thread at a time

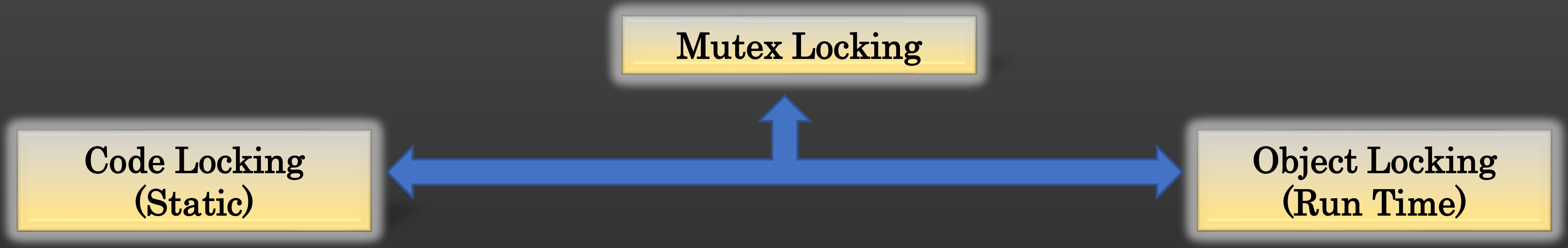
```
pthread_mutex_t mutex;

   foo() {
   . . .
   . . .
   . . .
I1 pthread_mutex_lock(&mutex);

I2 /* Critical Section */

I3 pthread_mutex_unlock(&mutex);
   . . .
   . . .
   }
```

- Which ever threads locks the Mutex, shall be able to enter into CS
- If a thread tries to lock already locked mutex, thread is blocked
- Multi-threads may be blocked by same Mutex
- When thread exits out the Critical Section, it MUST unlock the Mutex
- Among many threads waiting for the same Mutex, Mutex shall be granted to only one depending on several factors such as : Thread Priority, OS scheduling policy etc
- Thread must not intentionally die while holding the Mutex lock, that Mutex become unusable for forever
- Use of Mutexes causes Threads to block and unblock. More the blocking and unblocking of threads, More Scheduling work overhead on OS, poorer shall be the appln performance
- Bigger the size of CS, larger the time the blocked threads would have to wait, hence again poorer shall be application performance.

Thread Synchronization → Mutex locking Rules

- If T1 locks a mutex M, only T1 can unlock it
- T1 cannot unlock an already unlocked mutex -> Undefined behavior
- If T1 locks the Mutex M, T2, T3 . .. will be blocked if they tries to lock M
- If T1, T2 … are blocked to acquire lock on already locked Mutex M, the OS scheduling policy will decide which thread (i.e. among T1 or T2 ..) would acquire the lock on M when M is unlocked by its owner
- If thread T attempts to Double lock the Mutex M, it will self-deadlocked
- You must try writing small programs and verify above Rules/behavior by yourself
- Mutexes Must be unlocked in LIFO Order (Recommendation)
- Homework : Explore pthread_mutex_trylock( ) API

Thread Synchronization → Mutex locking Types



- ➢ If you need to protect a code snippet against concurrent thread unsafe access, go for Code locking
- ➢ If you need to protect Object against concurrent thread unsafe access, go for Object locking

Thread Synchronization → Two ways to use Mutexes

Code Locking

foo.c

```
static pthread_mutex_t mutex;
foo() {
. . .
. . .
. . .
pthread_mutex_lock(&mutex);
/* Critical Section */
pthread_mutex_unlock(&mutex);

}
```

Code locking
Mutex are defined at Src file level

```
char global_buffer[256];
static  pthread_mutex_t mutex;
void
pkt_receive (char *pkt, int pkt_size) {

        pthread_mutex_lock(&mutex);
        memset (global_buffer, 0, sizeof(global_buffer));

        memcpy (global_buffer, pkt, pkt_size);

        forward_pkt (global_buffer, pkt_size);
        pthread_mutex_unlock(&mutex);

}
```

Example of Code locking

Thread Synchronization → Two ways to use Mutexes

Data Locking (Run Time Locking)

```
foo() {
. . .
. . .
. . .
pthread_mutex_lock(&list->mutex);

* Critical Section */
* perform_operation(&list); */

pthread_mutex_unlock(&list->mutex);

}
```

Data Structure locking
Every Data Structure is associated with its own
Mutex (acts like a body-guard)

Example

```
void
delete_node_from_list ( list lst, int a ) {

I5   pthread_mutex_lock(&lst->mutex);
I6           node_t *node = search_node( lst, a);
I7
I8           remove_node(node);
I9
I10          free(node);
I11  pthread_mutex_unlock(&lst->mutex);
}
```

Thread Synchronization → Mutex APIs

Declaration : pthread_mutex_t my_mutex;

Initializing : pthread_mutex_init(&my_mutex, NULL);

Destruction : pthread_mutex_destroy(&my_mutex);

Locking and Unlocking the Mutex :

```
pthread_mutex_lock(&mutex);

pthread_mutex_unlock(&mutex);
```

Note : Mutex must never be memcpy-ied --> Undefined behavior !

Let us see the example program : ThreadBasics/mutex_example.c

Thread Synchronization → Mutex Example

Goal :

- Witness undefined behavior without Mutual Exclusion
- Fix it using Mutexes

D

```
int arr[] = { 1, 2, 3, 4, 5 };
```

Let us see the example program : ThreadBasics/MutexExample/mutex_example.c

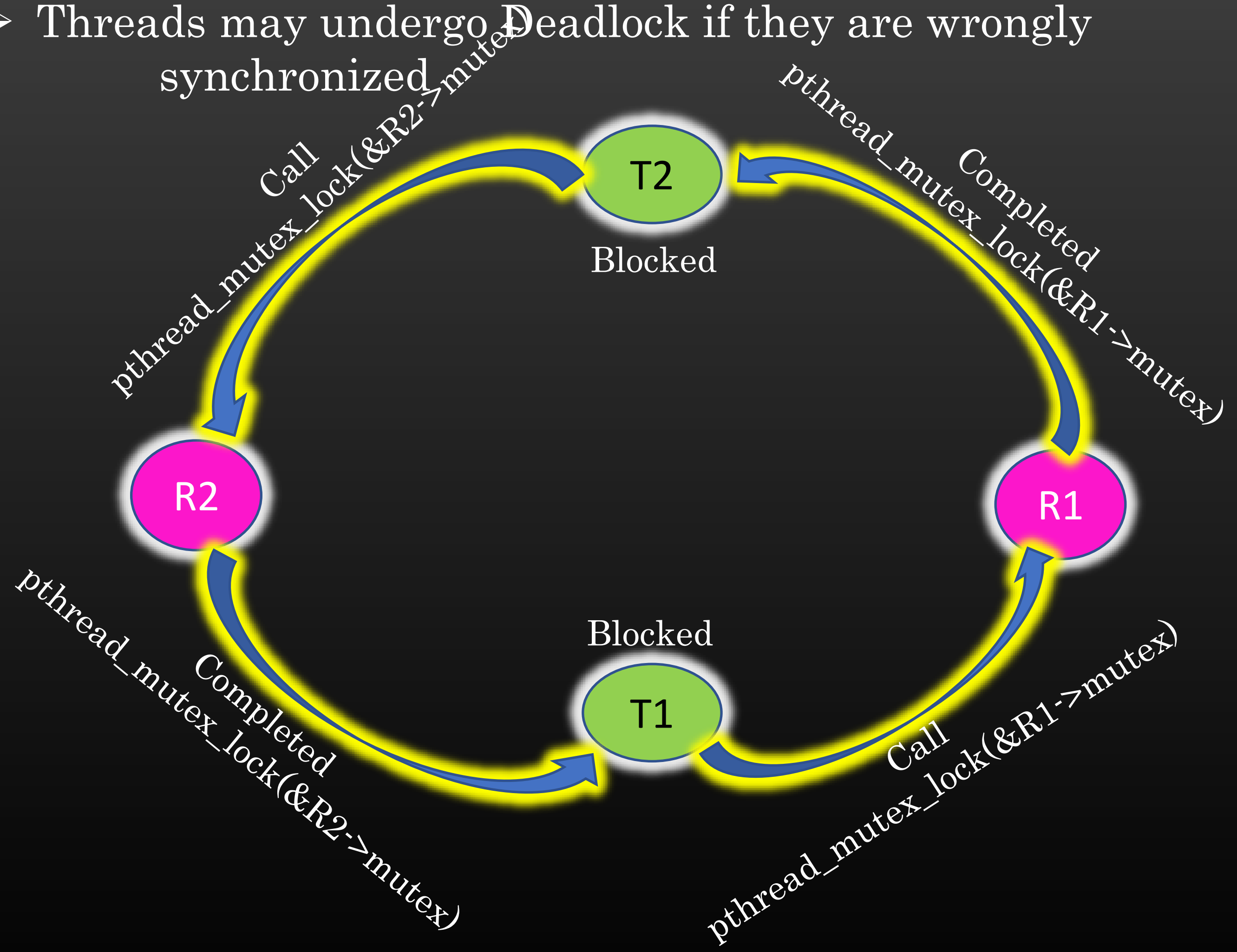
Thread Synchronization → Mutex Examples

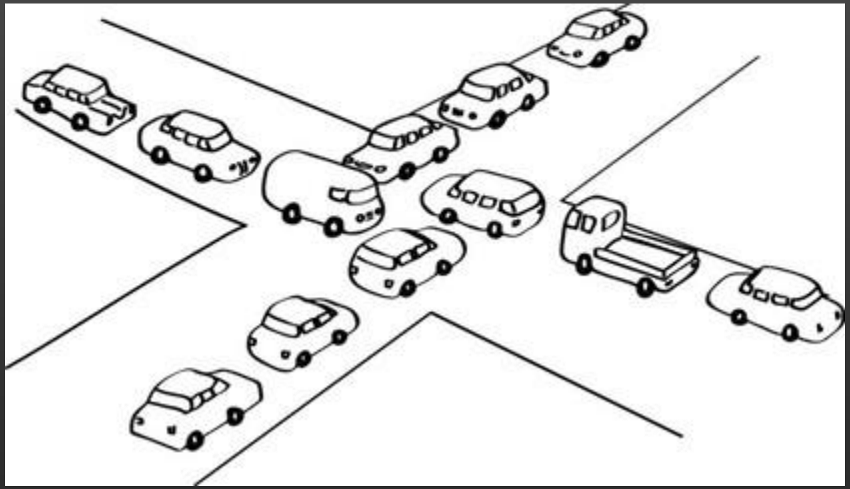
- Let us Write few programs demonstrating the use of Mutexes
  - strict_alternation.c

Thread Synchronization → Deadlock

- Deadlock is a situation where nobody makes a progress, and gets blocked for forever

- Threads may undergo Deadlock if they are wrongly synchronized


Call
pthread_mutex_lock(&R2->mutex)
T2
Blocked
pthread_mutex_lock(&R1->mutex)
Completed
R2
R1
Blocked
T1
pthread_mutex_lock(&R2->mutex)
Completed
Call
pthread_mutex_lock(&R1->mutex)



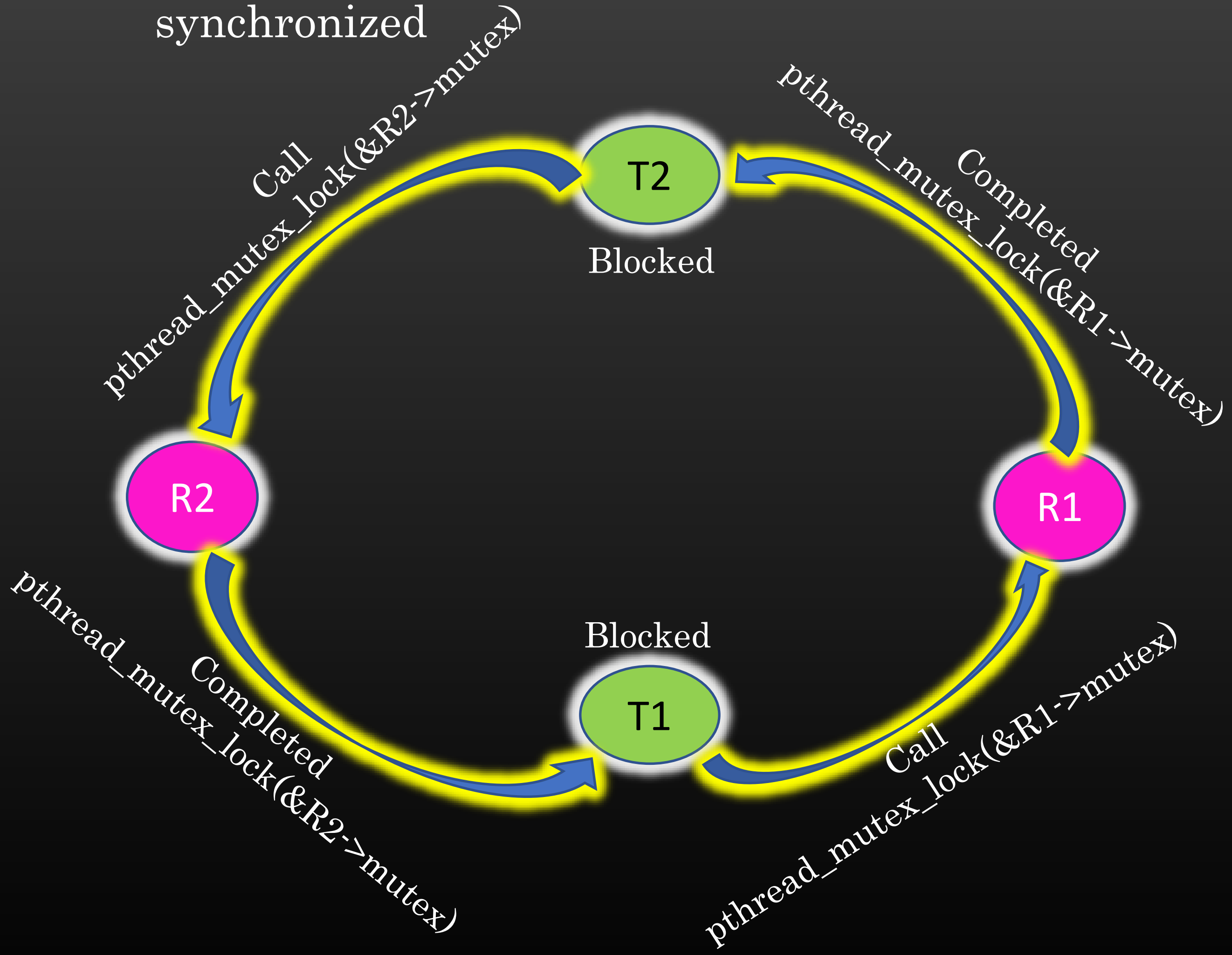
Deadlocked !

Each Is waiting other to release the Resource

They can never progress !

Thread Synchronization → Deadlock

- Deadlock is a situation where nobody makes a progress, and gets blocked for forever
- Threads may undergo Deadlock if they are wrongly synchronized


Call
pthread_mutex_lock(&R2->mutex)
T2
Blocked
pthread_mutex_lock(&R1->mutex)
Completed
R2
R1
Blocked
T1
pthread_mutex_lock(&R2->mutex)
Completed
Call
pthread_mutex_lock(&R1->mutex)

Four Necessary Conditions :

1. Mutual Exclusion: One or more than one resource are non-shareable (Only one thread can use at a time)

2. Hold and Wait: A thread is holding at least one resource and waiting for other resources.

3. No Preemption: A resource cannot be taken from a thread unless the thread releases the resource.

4. Circular Wait: A set of threads are waiting for each other in circular form.

Thread Synchronization → Deadlock → Inconsistent Locking Order

- There can be scenarios where a thread needs to lock multiple mutexes
- The order in which a given thread locks multiple Mutex Matter

T1

```
foo1( ) {
            . . I1. .
            pthread_mutex_lock(&mutex1);
            . . I2. .
            pthread_mutex_lock(&mutex2);
            . . I3. .
}
```

Deadlock !

T2

```
foo2( ) {
            . . I1. .
            pthread_mutex_lock(&mutex2);
            . . I2. .
            pthread_mutex_lock(&mutex1);
            . . I3. .
}
```

- The order of locking the mutexes must be preserved
- Order don’t matter, but whatever order is there, same order should be followed through out the code
- It is recommended to unlock the Mutexes in LIFO order
- All four necessary conditions for deadlock to occur are true in this example as well

Thread Synchronization → Locking Size -> example

➢ Suppose a thread T1 wants to perform some expensive operation OP(node) on a node of a linked list

```
foo( ) {

        pthread_mutex_lock(&list->mutex);
        node = list_node_search(list, a);
        if (!node){
                pthread_mutex_unlock(&list->mutex);
                return;
        }
        OP(node);
        pthread_mutex_unlock(&list->mutex);
}
```

```
foo( ) {

        pthread_mutex_lock(&list->mutex);
        node = list_node_search(list, a);
        if (!node){
                pthread_mutex_unlock(&list->mute
                return;
        }
        pthread_mutex_lock(&node->mutex);
        pthread_mutex_unlock(&list->mutex);
        /* Here list access become available to other threads*/
        OP(node);
        pthread_mutex_unlock(&node->mutex);
        }
```

Condition Variables

- You would want to wait from having a delicious meal, unless it is cooked fully
- You would not buy expensive headphones until your next salary
- We always wait in our day to day life until some condition is satisfied , right ? Its Natural.

Thread Synchronization → Condition Variables

- **Condition Variables** allow us to have finer control over taking the decision on when and which competing thread to block/resume
- Eg : Thread T1 finds Queue Empty, it wants to wait until Queue has some element in it
- CV allows threads to get itself blocked or wake up when certain condition is met

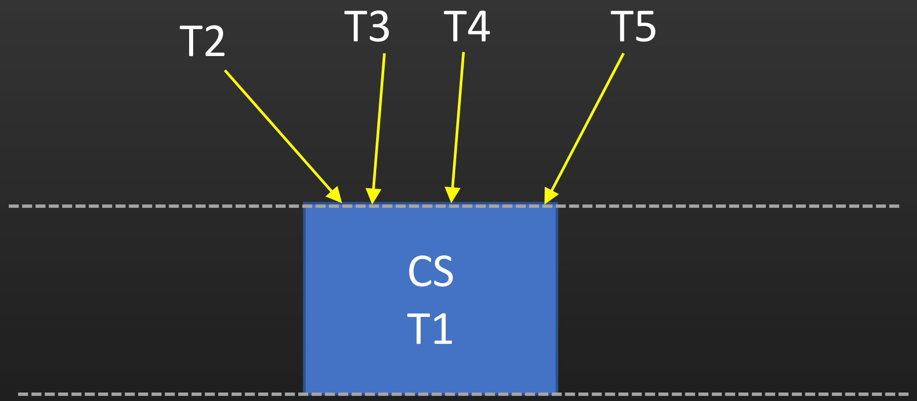
Mutexes only says –
Go ahead if you have access, wait if you don't

Like - *Yes Or No !*

Using Mutexes we can't implement below logic :

If Queue is empty, wait until the Queue has some Element in it

Mutex cannot implement condition-based blocking and wake up of threads, For this CV is required



When T1 leaves the CS, the lock is granted to the blocked thread chosen by the kernel's scheduling Policy.

Programmer has little control as to which thread should go next

CV + Mutex Combo is what it all takes to implement any advanced thread synchronization Scheme : Monitors, Producer-Consumer, Dining philosopher, Thread Scheduler, Semaphores, Wait Queues, Barriers etc

Thread Synchronization → Condition Variables Vs Mutex

- Mutex grant an access to the resource if it is not locked already,
- CVs allows threads to inspect the resource state and decide if it wants to wait for favorable resource state

Mutexes

- Access the laptop if it is not used by somebody else

Only Mutual Exclusion

```
pthread_mutex_lock(&laptop->mutex);

        /* Do whatever you want */

pthread_mutex_unlock(&laptop->mutex);
```

Condition Variables

- Access the laptop only if it not used by somebody else and if it has internet connection

Mutual Exclusion + Custom Condition

```
pthread_mutex_lock(&laptop->mutex);

        if (!laptop->internet_connection){
                wait( cv, &laptop->mutex);
        }
    // use laptop
pthread_mutex_unlock(&laptop->mutex);
```

Thread Synchronization → Condition Variables

- CV are not used for Mutual Exclusion, they are for Co-ordination (Signaling)
- First Let us understand the basics of CVs, how they are used and then we dive deep into advanced application of CVs
- To put it simple :
  - Using CVs, a thread can block itself (pthread_cond_wait)
  - Using CVs, a thread can signal already blocked thread (blocked by CV) to resume (pthread_cond_signal)
- Let us understand the concept of Wait and Signals – The end result of Invention of CVs

Thread Synchronization → Condition Variables → Wait and Signal Steps

Blocking a thread using CV

- A thread blocks itself using CV in 2 steps :
  - Step 1 : Lock a mutex
  - Step 2 : invoke pthread_cond_wait

T1

```
   pthread_mutex_t mutex;
   pthread_cond_t cv;

   pthread_mutex_init (&mutex, NULL);
   pthread_cond_init (&cv, NULL);

   . . .
   printf ("T1 is getting blocked");
S1 pthread_mutex_lock(&mutex);
S3 pthread_cond_wait(&cv, &mutex);
   printf ("T1 is awakened");
S5 pthread_mutex_unlock(&mutex);
   . . .
```

- Whoever invokes pthread_cond_wait( ) , gets blocked
- Romeo and Juliet, CV and Mutex Variable

- When thread invokes pthread_cond_wait( ) , two things happen :
  1. Thread gets blocked (job of CV)
  2. Mutex ownership is snatched from calling thread and is declared available

- When blocked Thread receives signal:
  1. It slips into *ready to execute state* and wait for mutex release
  2. It is given a lock on mutex as soon as mutex is released by signaling thread
  3. Thread resumes execution

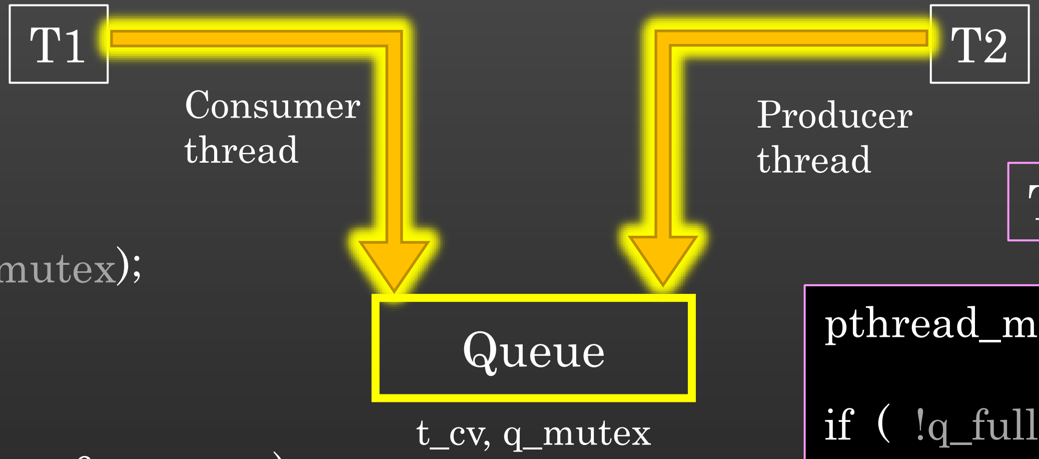
T2

```
   . . .
S6 pthread_mutex_lock(&mutex);
S8 pthread_cond_signal(&cv);
S9 pthread_mutex_lock(&mutex);
```

Signaling a blocked thread using CV involves 3 steps :

- Lock mutex
- Invoke pthread_cond_signal,
- unlock mutex

Thread Synchronization → Condition Variables -> Mutex + CV + Predicate



T1 (Consumer Threads)

```
S1  pthread_mutex_lock(&q_mutex);
S2    if (q_empty(q) ){
S3      pthread_cond_wait(&t_cv, &q_mutex);
      }
S4    /* perform Ops on Queue (C.S) */
S5    pthread_mutex_unlock(&q_mutex);
```

T2 (Producer Thread

```
pthread_mutex_lock(&mutex);          S6
if (  !q_full (q) ( ) ) {            S7
    produce_and_enque(q);
    pthread_cond_signal (&t_cv );    S8
}
pthread_mutex_lock(&mutex);          S9
```

- To exclusively perform S2 (predicate), step S1 (Mutex locking) is required
- Predicate is a condition which tells the thread whether it has to wait or not
- If the condition for wait is true (S2), thread blocks using CV (S3)
- T2 check the resource state (S7) exclusively (S6), and produce the new elem
- T2 sends signal to T1 (S8), unlock mutex (S9)
- Step S4 is the C.S which T1 executes when woken up
- Step S5, explicitly unlock the shared data when done

Spurious Wake Up

You wake up only to find that whatever you were promised (or told) has been broken (or is a lie)

Eg : You recvd a phone call from your dearest friend, and he invites you for a booze party at a common friend's house
While you are on the way, the common friend leaves the town out of some emergency, he did not inform you
You reach his home only to find your common friend no-where → Spurious Wake up
It takes finite amount of time for you to reach your friend's house, and within that time interval the situation changed such that it was futile for you to reach your friend's house

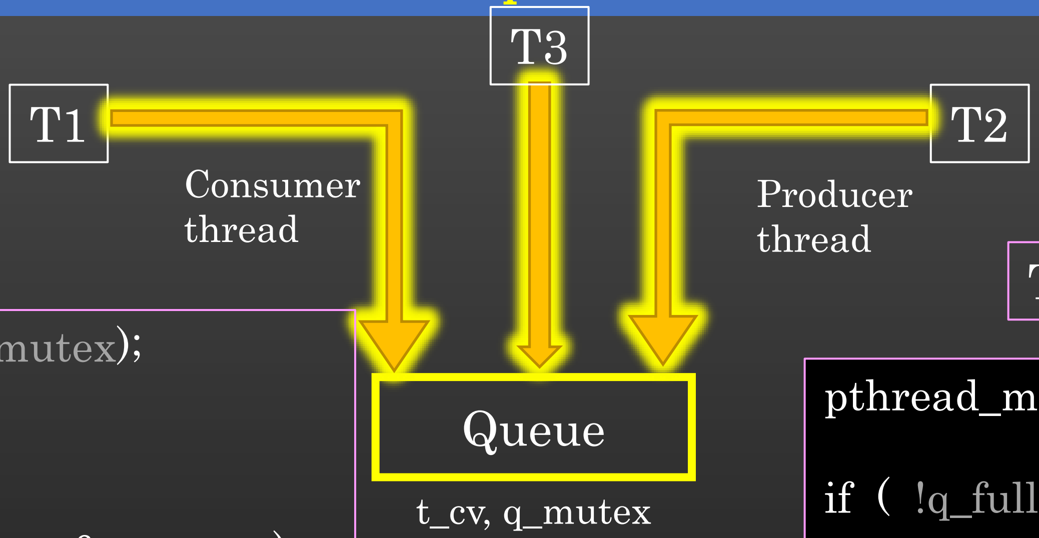
Eg : Your Father bought sweets exclusively for you, he advised to consume after having a bathe
You go for a bath
your sibling over-heard this conversation, and eat all the sweets in the box, leaving the box empty
You come out of the washroom, only to find sweet-box empty (against your expectation) → Spurious Wake up
You are sad and angry !

When you wake up Spuriously you find :

- The situation is against your expectations
- You think you have been given wrong information (by your dearest friend, or by your father)
- But, they did not give you wrong info, somebody else just cheated

➢ When a thread gets unblocked (bcoz it has recvd a signal), it can resume execution due to a reason which is no more valid

Thread Synchronization → Condition Variables → Loop Hole !



T1 (Consumer Threads)

```
S1  pthread_mutex_lock(&q_mutex);
S2   if (q_empty(q) ){
S3     pthread_cond_wait(&t_cv, &q_mutex);
     }
S4   /* perform Ops on Queue (C.S) */
S5   pthread_mutex_unlock(&q_mutex);
```

T2 (Producer Thread

```
pthread_mutex_lock(&mutex);  S6
if ( !q_full (q) ( ) ) {  S7
     produce_and_enque(q);
     pthread_cond_signal (&t_cv );  S8
}
pthread_mutex_unlock(&mutex);  S9
```

S94 [S9 – S4] Some other Consumer thread T3 of the process again empties the Queue, before T1 even get the chance

Our Consumer thread T1 ends up on consuming an empty Queue [S4] – undesirable situation !

Soln : Consumer thread , when wake up, must check the predicate condition again before processing the Queue

Thread Synchronization → Condition Variables → General Pseudocodes

Consumer Thread :
General pseudo to block on CV

```
pthread_mutex_lock(&mutex);

while ( predicate( ) ) {

    pthread_cond_wait (&cv, &mutex);

}

execute_cs_on_wake_up ( );

pthread_mutex_unlock(&mutex);
```

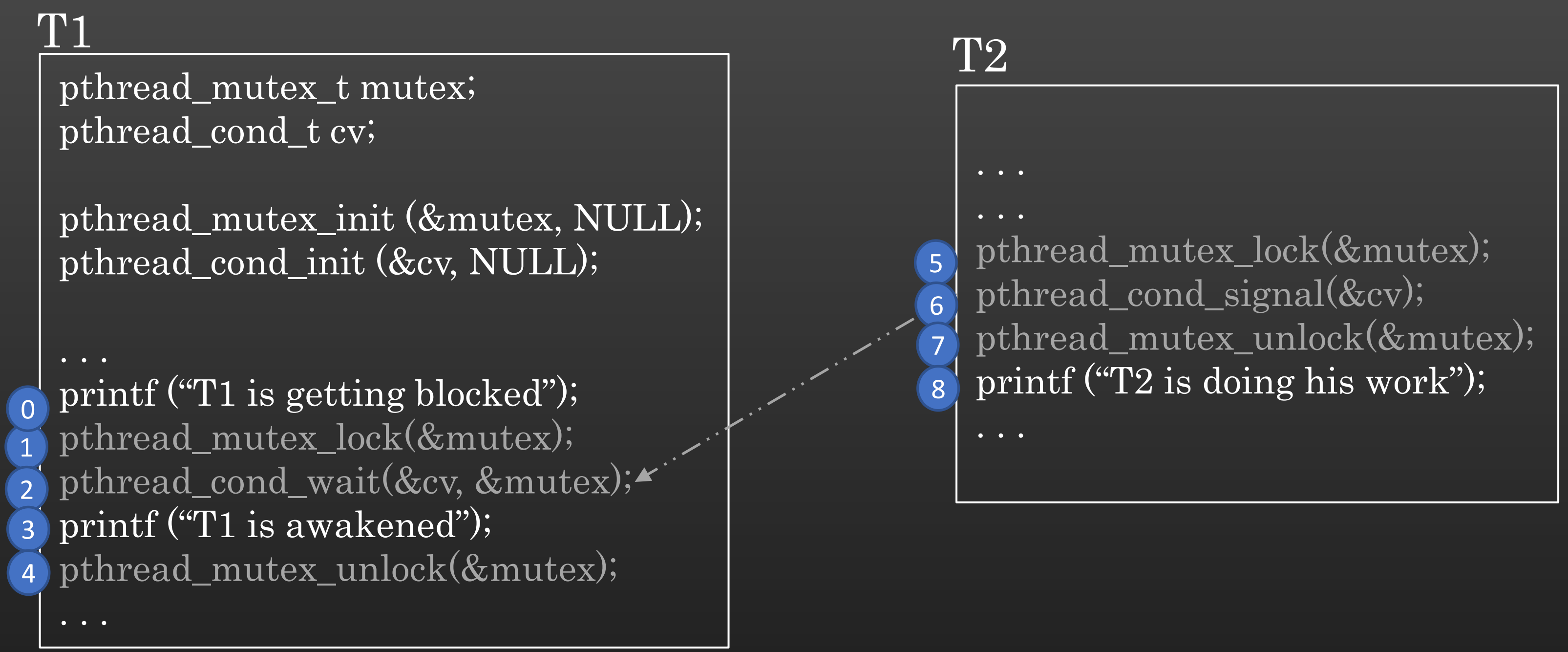
Producer Thread :
General pseudo to signal on CV

```
pthread_mutex_lock(&mutex);

if  ( !predicate( ) ) {

    pthread_cond_signal (&cv );

}
pthread_mutex_unlock(&mutex);
```

- Generic Pseudocodes, follow the same pattern while coding Thread Sync problem and solutions
- This is the most tight Thread synchronization between Producer and Consumer threads which is consistent and concrete
- Most thread synchronization problems can be decomposed into smaller Producer – Consumer Problems

Thread Synchronization → Using Wait and Signal - Steps

- At step 1, T1 locks the mutex (obviously this would make T2 blocked at step 5)
- At step 2, T2 gets blocked + mutex is unlocked (behind the scenes)
- At step 5, T2 locks the mutex, if it was already blocked on mutex, T2 will resume
- At step 6, T2 sends signal to T1
- T1 is awakened, it will try to grab a lock on mutex (behind the scenes)
- But mutex is locked by T2, therefore T1 stays blocked (but wanting to run)
- At step 7, T2 unlocks mutex, the mutex lock is given to T1 (behind the scenes)
- Now T1 can resume as normal, execute step 3
- T1 have to release the lock on mutex explicitly as in Step 4 (complimentary to step 1)
- T2 was never blocked, and can execute step 8



T1

```
  pthread_mutex_t mutex;
  pthread_cond_t cv;

  pthread_mutex_init (&mutex, NULL);
  pthread_cond_init (&cv, NULL);

  . . .
0 printf ("T1 is getting blocked");
1 pthread_mutex_lock(&mutex);
2 pthread_cond_wait(&cv, &mutex);
3 printf ("T1 is awakened");
4 pthread_mutex_unlock(&mutex);
  . . .
```

T2

```
  . . .
  . . .
5 pthread_mutex_lock(&mutex);
6 pthread_cond_signal(&cv);
7 pthread_mutex_unlock(&mutex);
8 printf ("T2 is doing his work");
  . . .
```

pthread_cond_wait( ) performs two operations atomically :

1. blocking the calling thread
2. unlocking the mutex (2nd arg)

What will happen if T2 signals T1 when T1 is not even blocked on CV ?

Let us do some stupid project to practice CV !

Thread Synchronization → Wait and Signal → Analogy

- let us try to understand through analogy

- Let's say, you Y and your friend F is playing a game
  - There is a box with some candies in it
  - The box is locked, and there is one key
  - You have to unlock the box using key, and take a candy out
  - Your Friend have to unlock the box using the key, and put a candy inside it
  - Only one who has a key can lock and unlock the box

- This is Typical Producer Consumer Scenario, when Y is consumer and F is producer

Thread Synchronization → Wait and Signal → Analogy

Y Steps

1. Y grab a key
S1 pthread_mutex_lock(&box->mutex);

2. Unlock the box

3. Y find the box empty
S2 if (empty(b)) {

Called "predicate"

4. Give up the key(lock the box) + Wait for the F to tell you to try again (from step s1)

S3 pthread_cond_wait(&box->cond_var, &box->mutex);

8.5 Y recv the notif that F is done with box, Y wants to access the box but cannot do so as F has not yet released the key to the box

10 Y get the key back

Called "predicate"

11 if (!empty(box)) {
12 Y take the candy out of the box S4

13 Y give up the key
pthread_mutex_unlock(&q->mut S5

F Steps

5. F grab a key
S6 pthread_mutex_lock(&box->mutex);

6. Unlock the box

S7 7. Place the candy in the Box
[enqueue(box, candy);]

8. Send a notif to Y to tell him – "I am done with box"
S8 pthread_cond_signal(&box->cond_var);

9. Give up the key

S9 pthread_mutex_unlock(&q->mutex);

Thread Synchronization → Multiple Condition Variables

> CV is associated with a Mutex Variable and a Predicate

- Many CVs can be associated with the same Mutex at the same time
- But 1 CV cannot be associated with more than 1 Mutex at the same time

Thread Synchronization → Multiple Condition Variables

pthread_cond_broadcast( )

```
pthread_mutex_lock(&mutex);
while ( predicate( ) ) {
    pthread_cond_wait (&cv, &mutex);
}
Print T1
pthread_mutex_unlock(&mutex);
Print "T1 is out of CS"
```

```
pthread_mutex_lock(&mutex);
while ( predicate( ) ) {
    pthread_cond_wait (&cv, &mutex);
}
Print T2;
pthread_mutex_unlock(&mutex);
Print "T2 is out of CS"
```

```
pthread_mutex_lock(&mutex);
while ( predicate( ) ) {
    pthread_cond_wait (&cv, &mutex);
}
Print T3
pthread_mutex_unlock(&mutex);
Print "T3 is out of CS"
```

Thread Synchronization → Condition Variables

- Points to remember about Condition Variables :

  > They must never be memcpy-ied (like Mutexes)

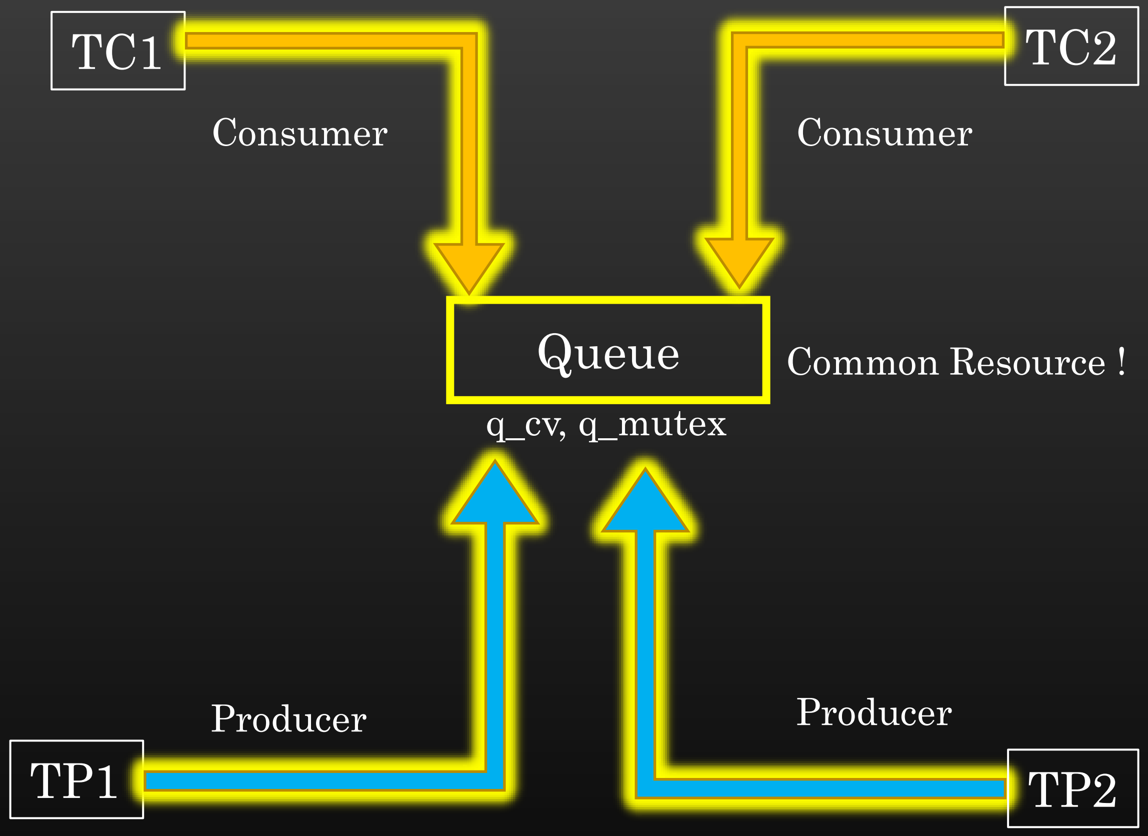
  > If you had initialized CV using pthread_cond_init( ) , then you must destroy CV using pthread_cond_destroy( )

  > Use PTHREAD_COND_INITIALIZER to statically initialize condition variable

  > CV are used for Co-ordination, not for Mutual Exclusion

Thread Synchronization → Implement Producer Consumer → Problem Statement

Code Dir : MultithreadingBible/ProducerConsumer
File : Assignment_prod_cons_on_Q.c
Supporting Files : Queue.c/.h, compile.sh



Problem Statement is attached in Resource Section

Thread Synchronization → Implement Producer Consumer → Problem Statement

Write a program which launches 4 threads - 2 consumer threads and 2 producer threads. Threads are created in JOINABLE Mode.

All 4 threads act on a shared resource - A Queue of integers. Producer threads produce a random integer and add it to Queue, Consumer threads remove an integer from the Queue. Maximum size of the Queue is 5.

<u>Following are the constraints applied :</u>

1. When producer threads produce an element and add it to the Queue, it does not release the Queue until the Queue is full i.e producer thread release the Queue only when it is Full

2. When consumer threads consume an element from the Queue, it consumes the entire Queue and do not release it until the Queue is empty.

3. Consumer Signals the Producers when Queue is Exhausted, Producers Signals the Consumers when Queue becomes full

Guidelines :
Use as many printfs as possible, so you can debug the program easily

Queue Operations

Queue.h

```
struct Queue_t* initQ(void);

bool
is_queue_empty(struct Queue_t *q);

bool
is_queue_full(struct Queue_t *q);

bool
enqueue(struct Queue_t *q, void *ptr);

void*
deque(struct Queue_t *q);

Q_COUNT(q)
```

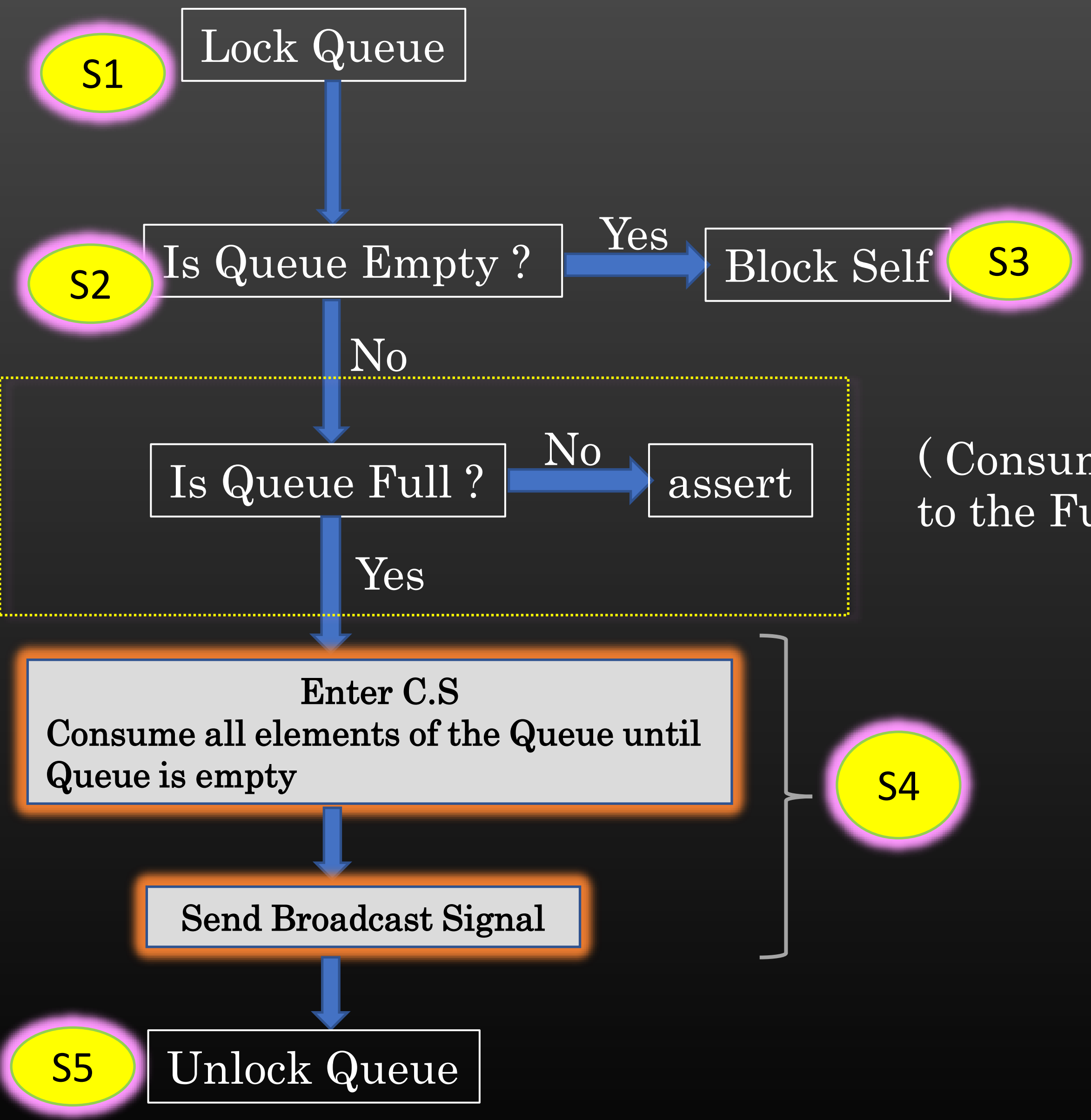
Example Usage

```
struct Queue_t *MyQ = initQ( );

bool  status = is_queue_empty(MyQ);

enqueue(myQ, (void *)5);

int a =  (int)deque(MyQ);

int count = Q_COUNT ( MyQ );
```

Next : Setting up the home-work program

Thread Synchronization → Implement Producer Consumer → Solution → Consumer Logic

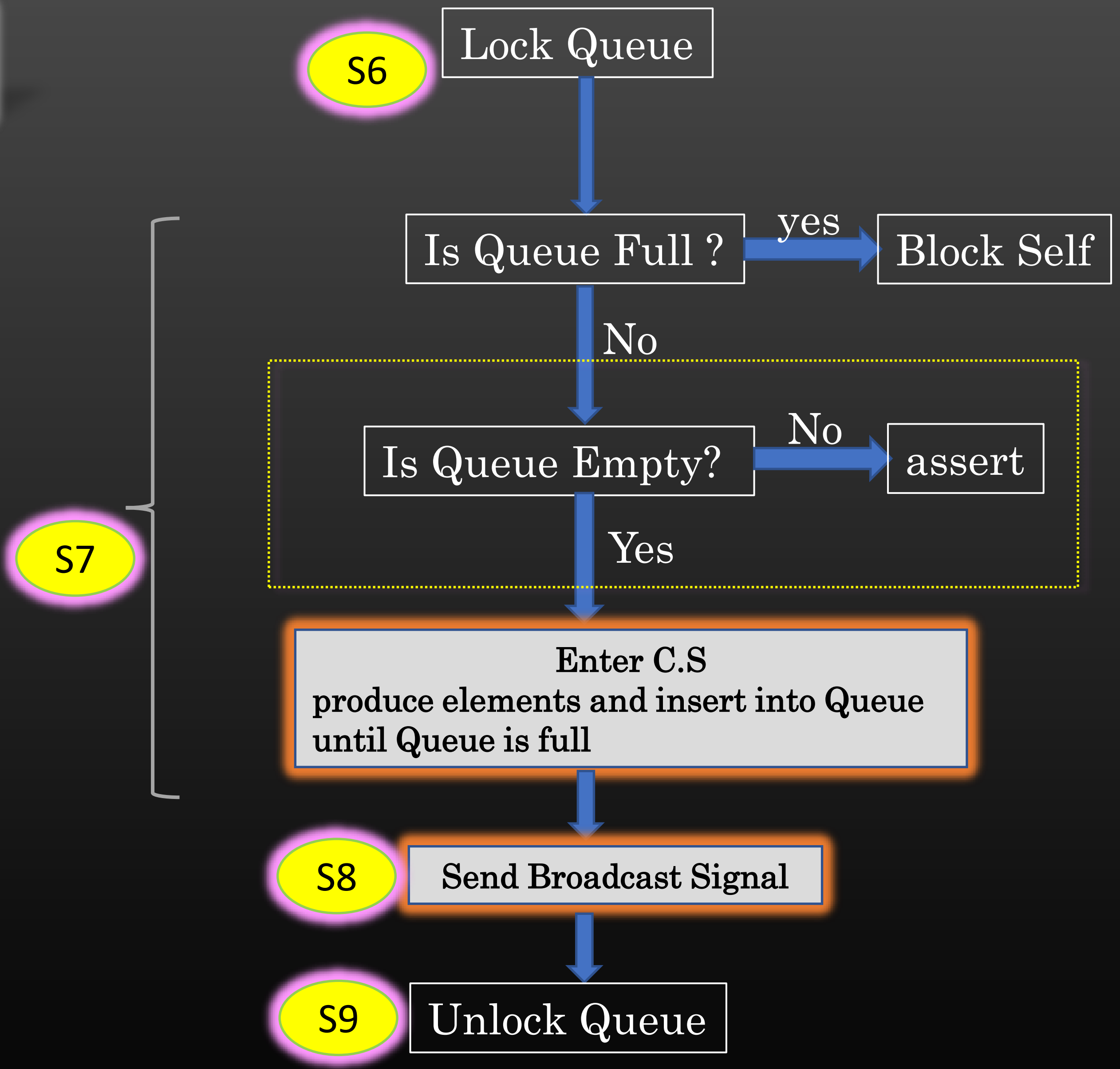
Consumer Thread
cons_fn( )



(Consumer must get an access to the Full Queue Only)

Thread Synchronization → Implement Producer Consumer → Solution → Producer Logic

Producer Thread
prod_fn( )



(Producer must get an access to the Empty Queue Only)

Dining Philosopher Problem

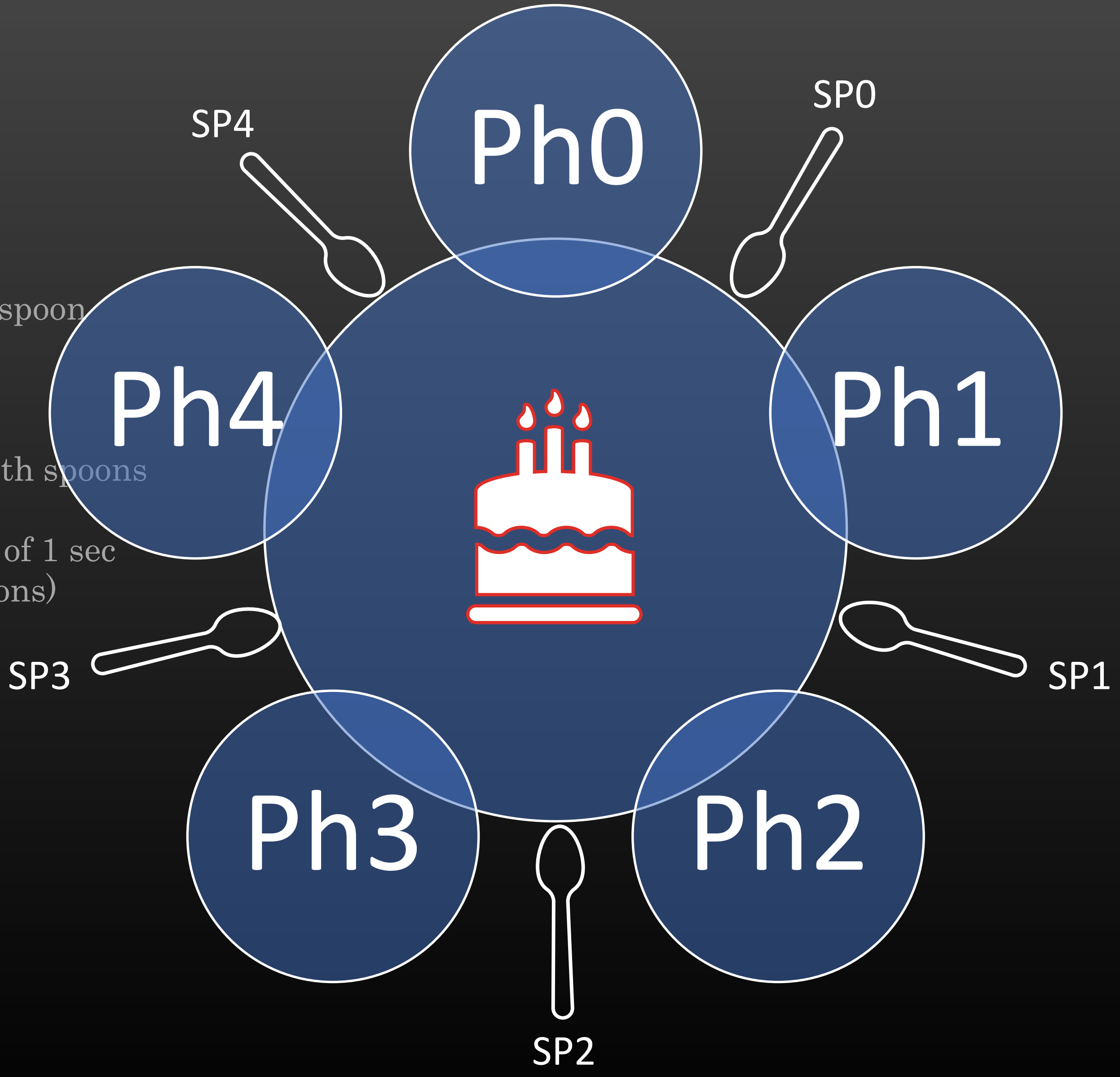
Thread Synchronization → Implement Dining Philosopher Problem → Problem Statement

Code Dir : MultithreadingBible/DiningPhilosopherProblem
File : assignment_din_ph.c
assignment_din_ph_soln.c

Constraints :

1. Philosopher can eat only when he has access to both spoons
2. Philosopher tries to get access to left spoon first, and then right spoon
3. If after getting access to left spoon, right spoon is not available,
   Philosopher has to release the left spoon also and
   restart from beginning after a wait of 1 sec
4. Philosopher enjoys the cake for 1 sec after it has got access to both spoons
5. Philosopher releases both spoons after enjoying cake for 1 sec
6. Philosopher makes an attempt for IInd slice of cake after a wait of 1 sec
7. Philosophers are threads, competing for adjacent resources (spoons)
8. All philosophers threads runs in infinite loop
9. Non-Adjacent Philosophers can eat in parallel

- Data Structures
- Program Structure
- Step by Step Solution
- Assignment



😆 **Practice , Practice and Just Practice !**

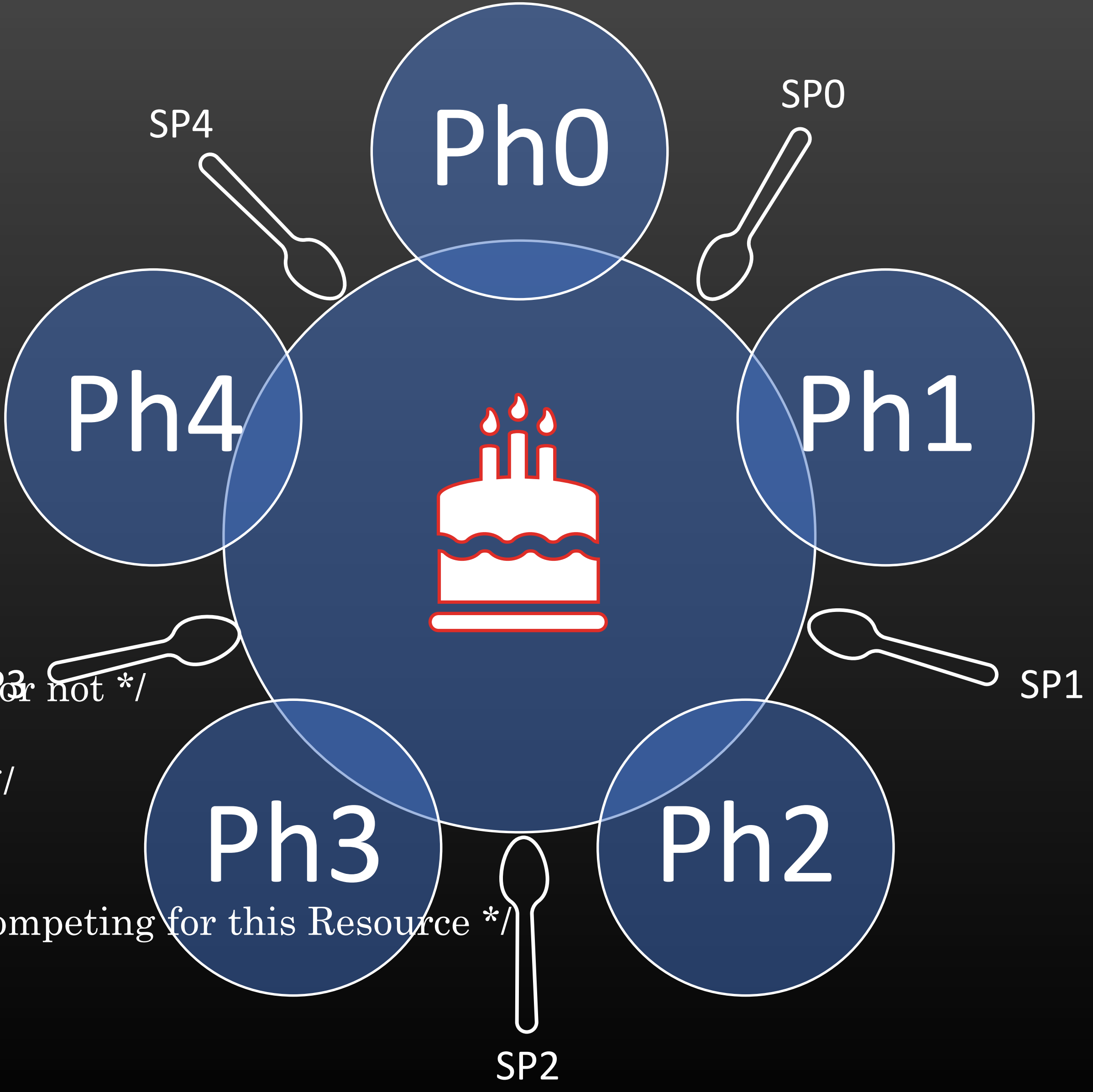
Problem Statement is attached in Resource Section

Thread Synchronization → Implement Dining Philosopher Problem → Data Structures Setup

Data Structure

din_ph.h

```
typedef struct phil_ {

    int phil_id;
    pthread_t thread_handle;
    int eat_count;
} phil_t;


typedef struct spoon_ {

    int spoon_id;
    bool is_used;   /* bool to indicate if the spoon is
                                                    being used or not */

    phil_t *phil;    /* If used, then which philosopher
                                                    is using it */

    pthread_mutex_t mutex;  /* For Mutual Exclusion */
    pthread_cond_t cv;   /* For thread Co-ordination
                                                            competing for this Resource */

} spoon_t;
```

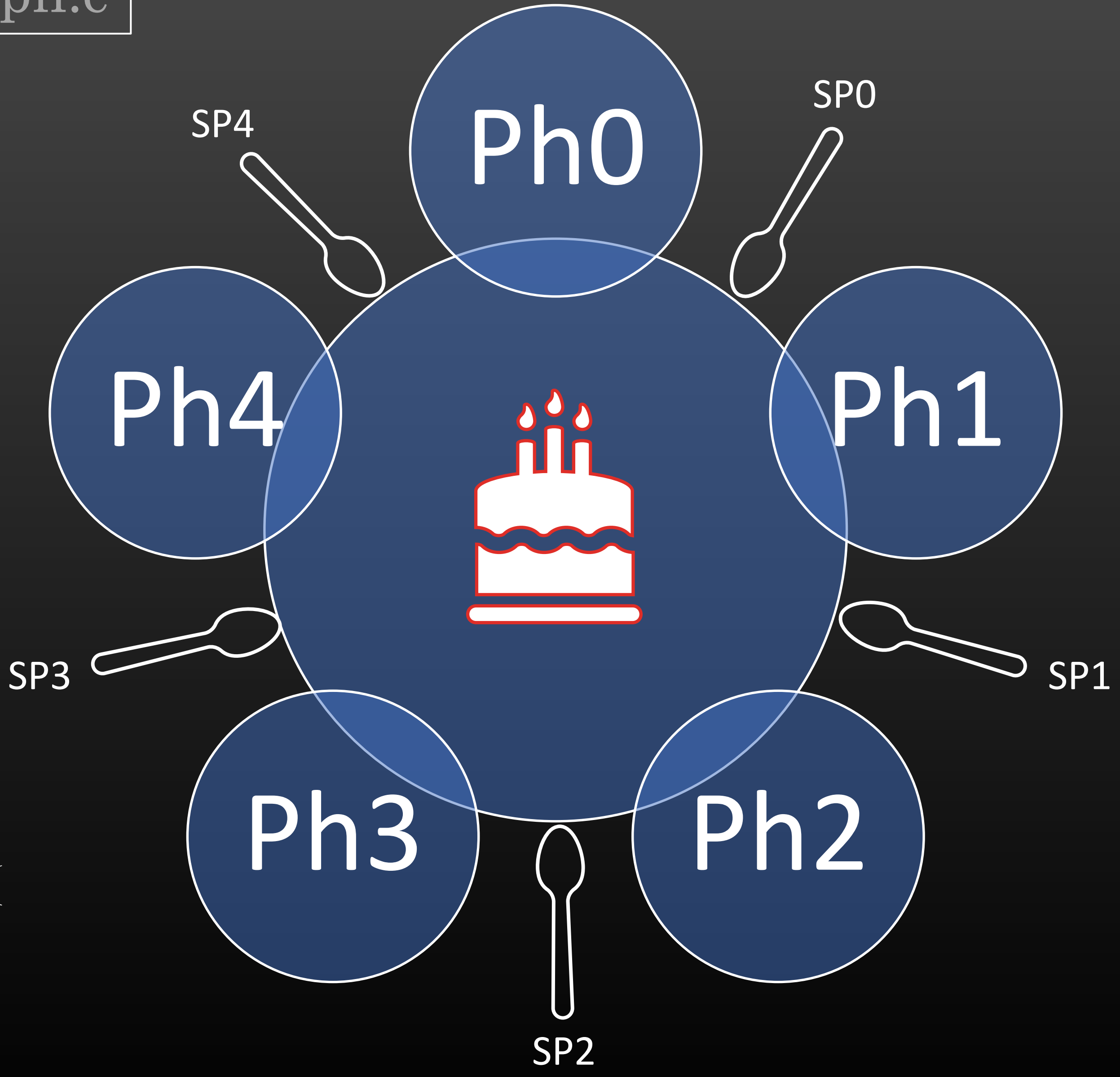


Thread Synchronization → Implement Dining Philosopher Problem → Common APIs

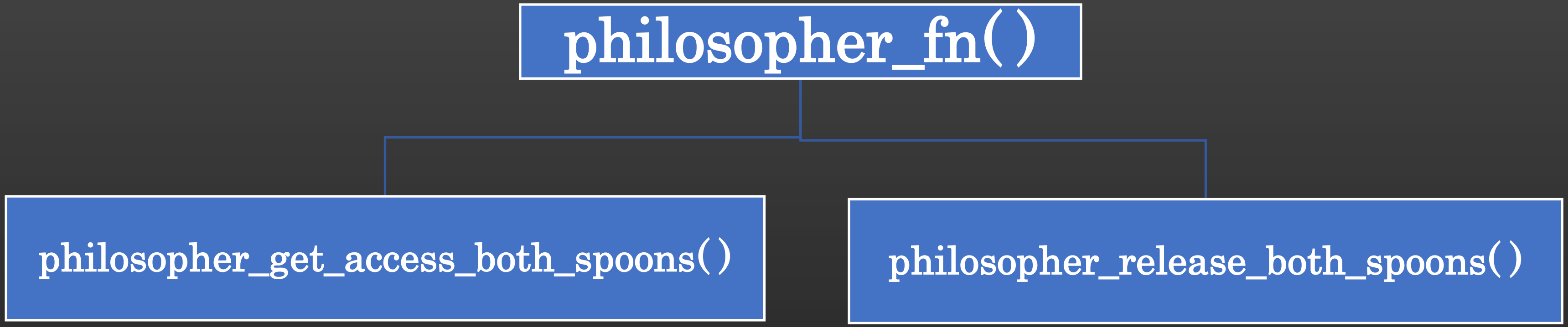
Common APIs

assignment_din_ph.c

```
static
spoon_t *phil_get_right_spoon(phil_t *phil);

static
spoon_t *phil_get_left_spoon(phil_t *phil);

static void
phil_eat(phil_t *phil);

static void
philosopher_release_both_spoons(phil_t *phil) {

        /* Core logic */
}

static bool
philosopher_get_access_both_spoons(phil_t *phil) {

        /* Core logic */
}
```



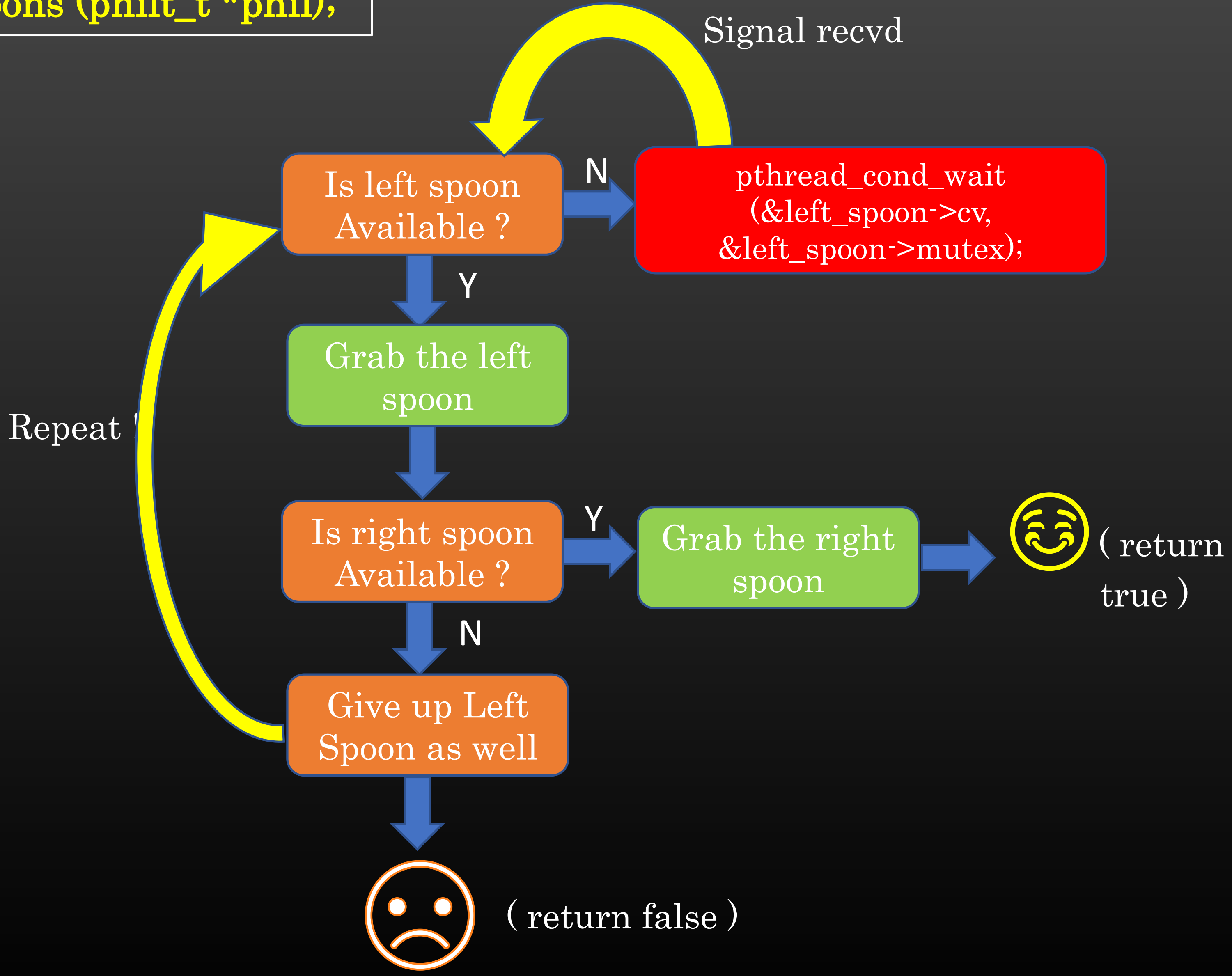
Thread Synchronization → Implement Dining Philosopher Problem → Solution


philosopher_fn( )
philosopher_get_access_both_spoons( )
philosopher_release_both_spoons( )

```
// Assigning a spoon S to phil P :
pthread_mutex_lock(&S->mutex);

S->is_used = true;
S->phil = P;

pthread_mutex_unlock(&S->mutex);
```

```
// Making the Spoon Available:
pthread_mutex_unlock(&S->mutex);

S->is_used = false;
S->phil = NULL;

pthread_mutex_unlock(&S->mutex);
```

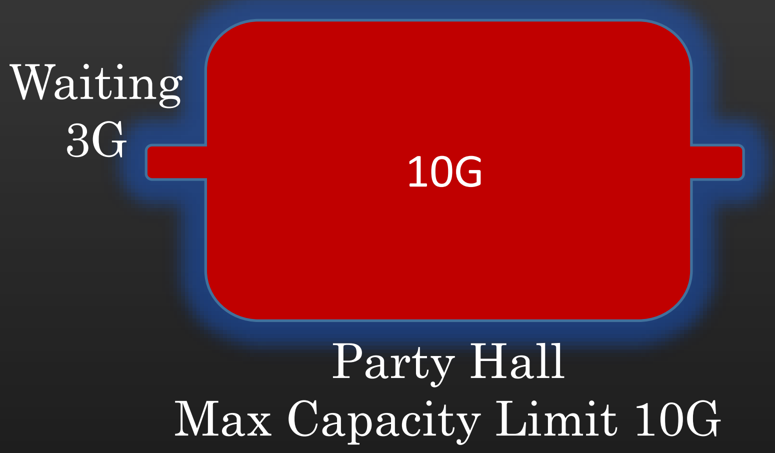
```
bool
philosopher_get_access_both_spoons (philt_t *phil);
```


Signal recvd
Is left spoon Available ?
N
pthread_cond_wait (&left_spoon->cv, &left_spoon->mutex);
Y
Grab the left spoon
Repeat !
Is right spoon Available ?
Y
Grab the right spoon
(return true)
N
Give up Left Spoon as well
(return false)

Semaphores
(Restrict the number of Resource Users)

Semaphores Analogy

- You are throwing your Bday party and only 10 guests can accommodate in a party hall
- But 13 Guests arrived ! You were bad organizer indeed !



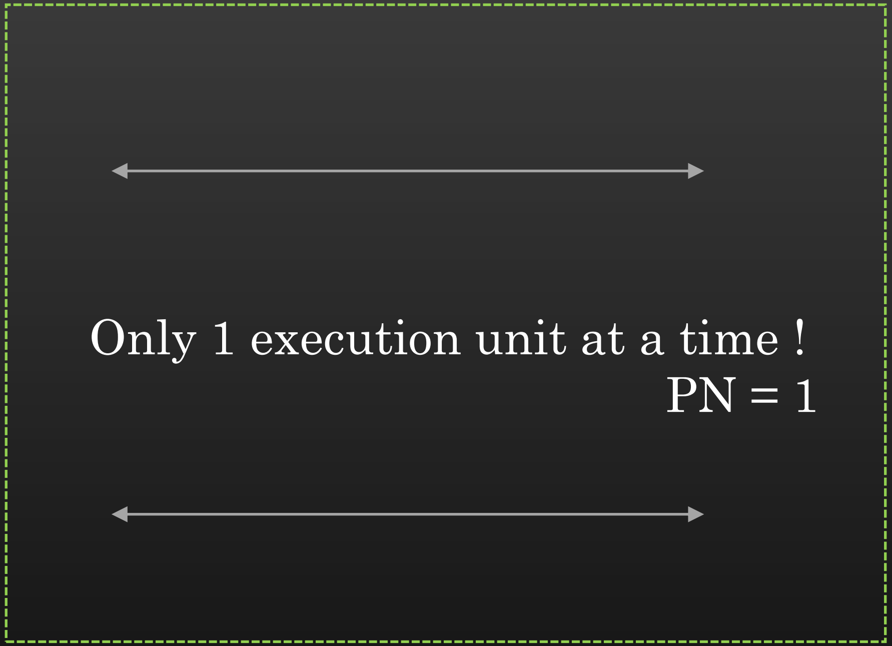
Place an upper bound on a no. of users of a resource

- Permit # is 10
- When one guest leaves, 1 Guest from Waiting list can enter into hall !

- This is where Semaphores comes into picture
  - Guests - Execution Units
  - Party Hall – C.S, Shared Resources

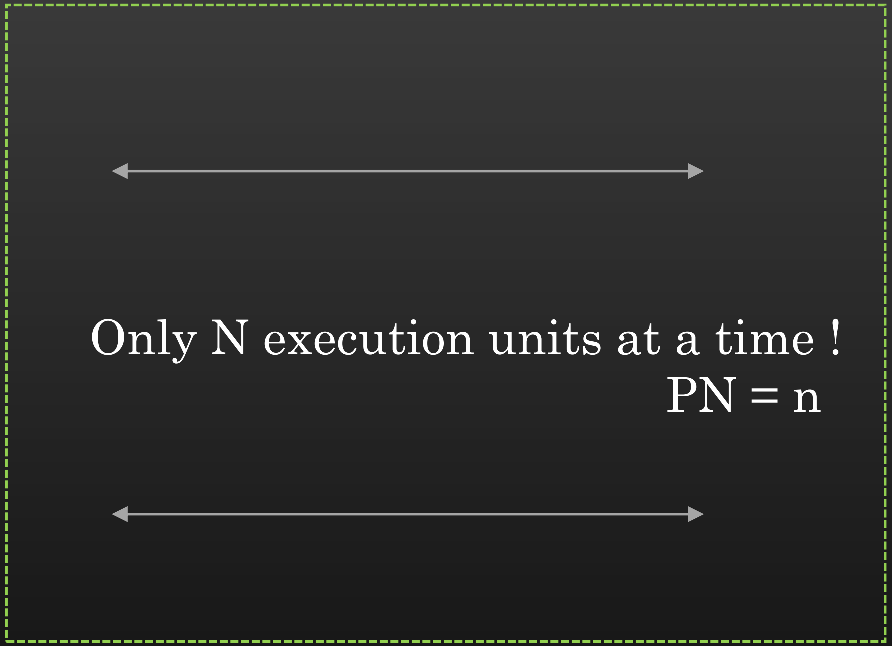
Semaphores Vs Mutex

Mutexes



Only 1 execution unit at a time !
PN = 1

Semaphores (n)



Only N execution units at a time !
PN = n

- If Semaphore S is initialized with n = 1, then Semaphore = Mutex in terms of functionality
- Therefore, Mutexes are also called binary semaphores

- Let us discuss next how semaphores work …

Semaphores APIs

```
#include <semaphore.h>

sem_t sem;  /* Declare Semaphore Variable */

sem_init (sem_t *sem, int pshared, int permit counter);
                              /* Initialize the Semaphore */

sem_wait (sem_t *sem);
                              /* Block the calling thread if Semaphore is not available */

sem_post (sem_t *sem);
                              /* Signal the blocked thread on semaphore */

sem_destroy (sem_t *sem);
                              /* Destroy the semaphore after use */
```

Semaphores Wait and Post API

sem_wait (sem_t *sem);

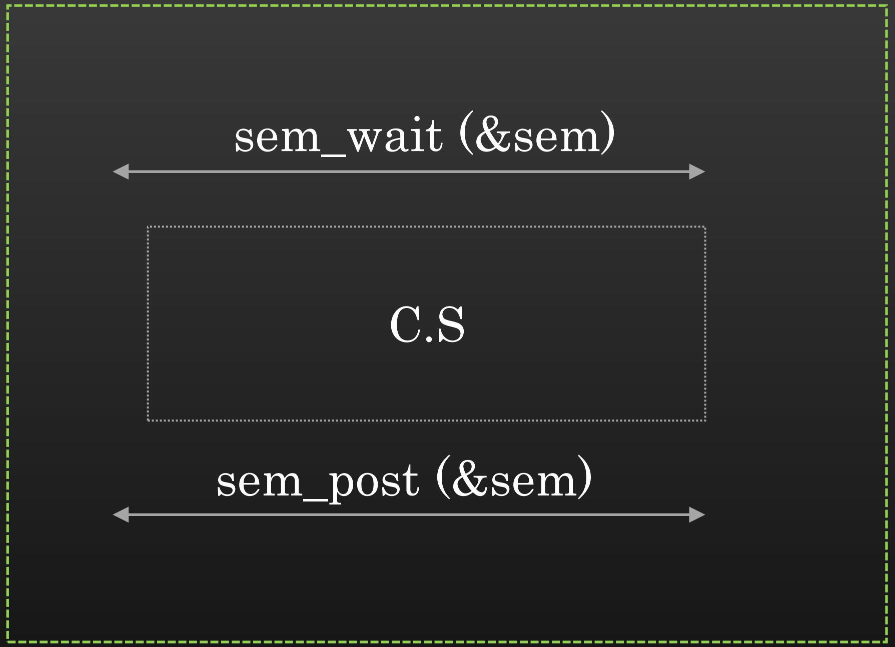
- Unconditionally decrease the PC of the Semaphore
- If PC < 0 after decrement, block the calling thread

sem_post (sem_t *sem);

- Unconditionally Increase the PC of the semaphore
- Send signal to threads blocked on sem_wait , if any

- End Goal : Allow at-most ‘n’ threads to execute concurrently in C.S
- If n = 1 , Binary semaphore which is same as mutex, but only difference is, semaphore can be unblocked (= sem_post) by a different thread

sem_init(&sem, n)



Semaphores
(Simple Example)

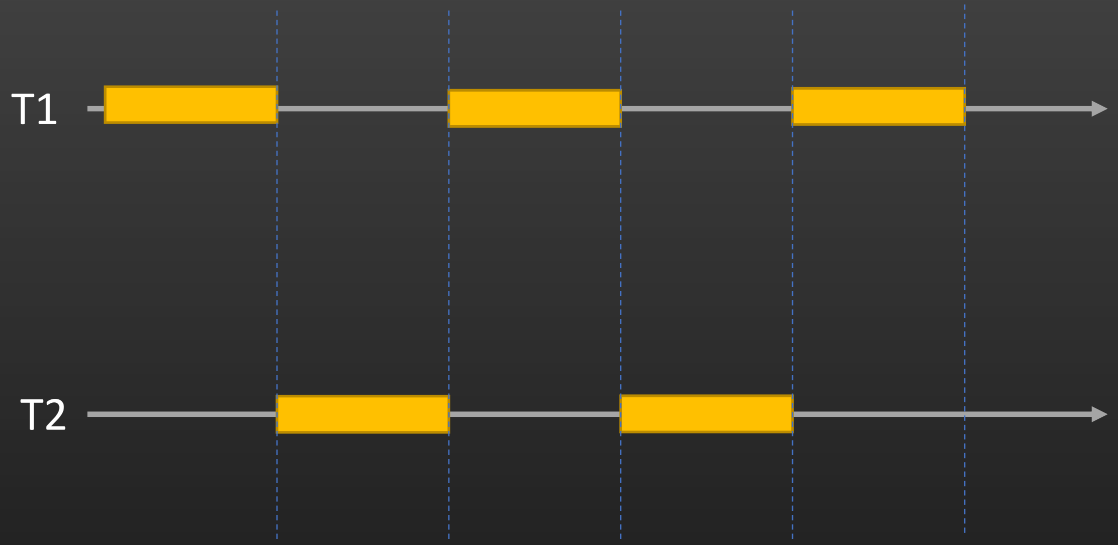
- Create 5 threads
- But allow only max 2 threads to execute inside C.S concurrently at a time
- You can FAKE C.S code by printing some msg by the executing thread
- MultithreadingBible/Semaphores/semaphore_hello_world.c

Strict Alternation

(Using Semaphore)

Semaphores → Strict Alternation

- It is of practical significance to make a pair of threads to execute in strict alternation manner



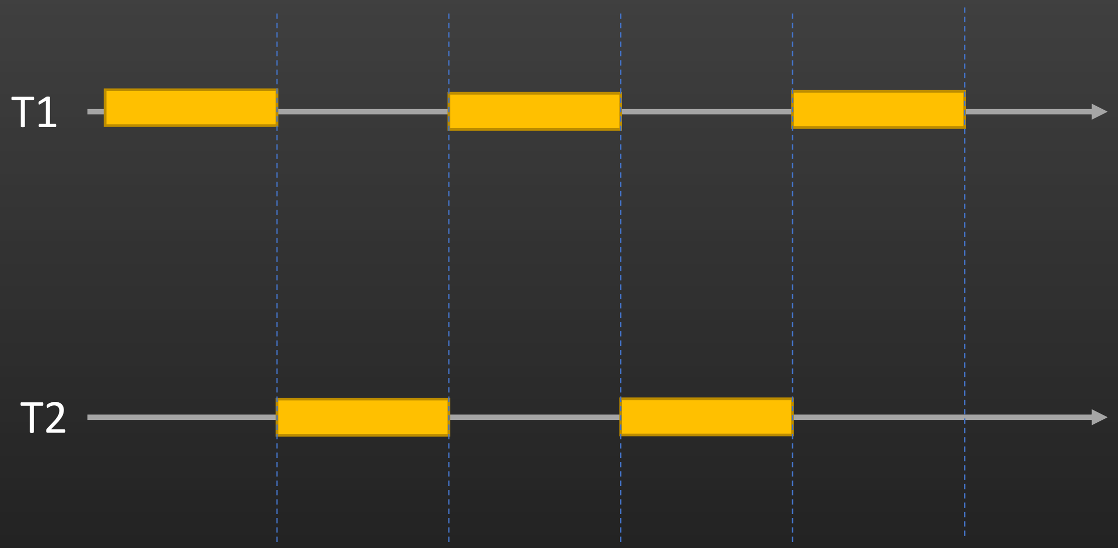
Zero Semaphores

sem_init(&sem, 0, 0);

sem_wait(&sem); -- Calling thread is immediately blocked
sem_post(&sem); -- blocked thread , if any, wakes up

Semaphores → Strict Alternation

- It is of practical significance to make a pair of threads to execute in strict alternation manner



?

Write a program which create Two Threads T1 and T2. T1 thread prints odd numbers between 1 to 15. T2 prints even numbers between 2 to 15. Output should be sequential.

T1 :

```
for( i = 1; i < 15; i+=2) {


   print i;
   sem_post (sem0_1);
   sem_wait (sem0_2);
}
```

T2 :

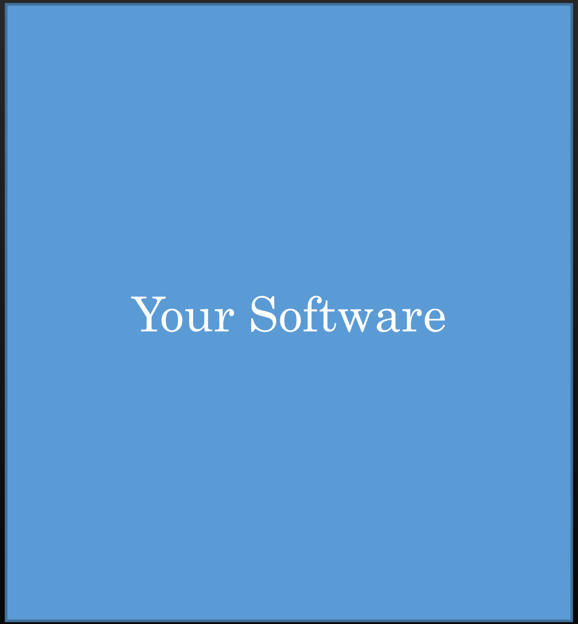
```
for( i = 2; i < 15; i+=2) {


   sem_wait (sem0_1);
   print i;
   sem_post (sem0_2);
}
```

Semaphores → Strict Alternation

- You ask your servant to go and do some piece of work, and you wait until he finishes the work and comes back to notify you that work has been done

- You again assign new work to the servant, and repeat ..

You have written some software
And want to write a test cases for it



Testing the software by executing its code in the context of outsider
Thread, no need to run or deploy the application !
For ex, your application is a routing protocol, using this approach you can Test the routing protocol without having to deploy the network in the lab
In other words, you don't have to run the routing protocol as a process on the Router
= Testing the strength of the fighter jet engine in the Lab !

Semaphores
(Internal Implementation)

- Mutex and CV are building blocks
- In-fact, Mutex and CV are building blocks for all Thread synch Data structures

Semaphores Counters Definitions

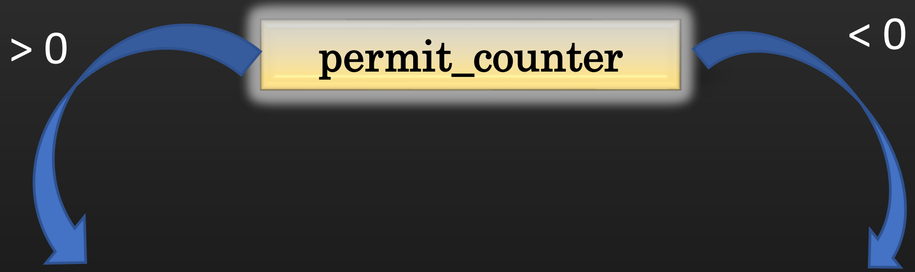
- Semaphores Permit Counter
  - int permit_counter

Party Hall
Max Capacity Limit 10

Semaphores Counters Definitions Obsolete

- Semaphores work with two counters –
  - int permit_counter
  - uint pending_signals

Party Hall
Max Capacity Limit 10



Current # of Guests who can freely enter the party hall

# of threads which are allowed to enter C.S without any wait

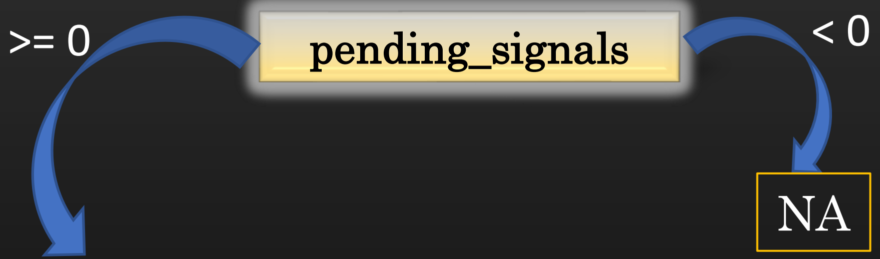
Current # of Guests who are waiting outside the party hall (bcoz party hall is full)

# of threads which are blocked from entering C.S because max allowed limit has reached

Semaphores Counters Obsolete

- Semaphores work with two counters –
  - int permit_counter
  - uint pending_signals

Party Hall
Max Capacity Limit 10



# of Guests who have left the party hall
provided there is at-least one Guest Waiting
and as many guests can now enter party hall

# No of threads which have left the C.S provided
there is at-least one thread blocked from entering the C.S
and as many threads can now enter into C.S

Semaphores → Implement Yourself

sema.h/sema.c

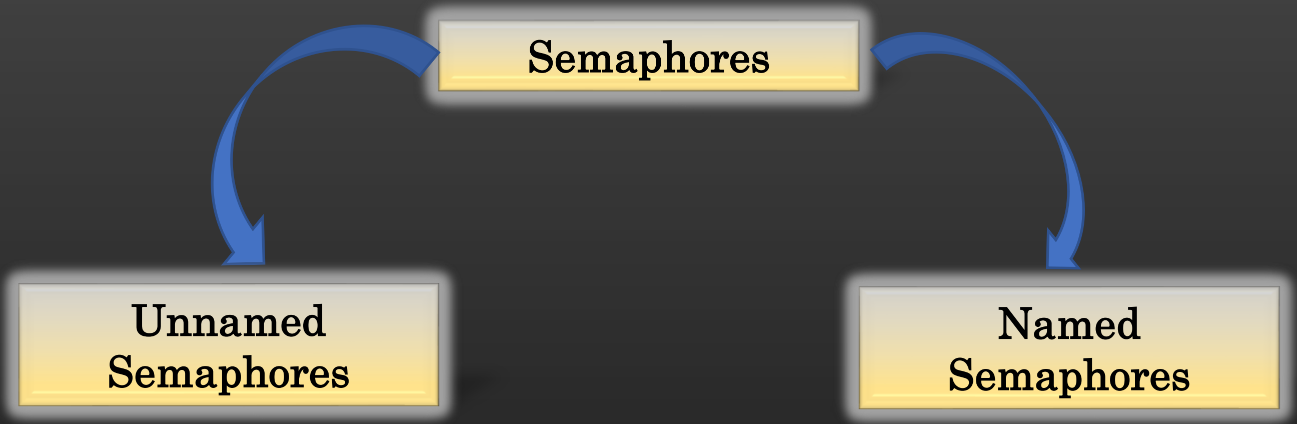
```
typedef struct sema_ sema_t;

struct sema_ {

    int permit_counter;
    pthread_cond_t cv;
    pthread_mutex_t mutex;
};
```

```
sema_t *
sema_get_new_semaphore();

void
sema_init(sema_t *sema, int count);

void
sema_wait(sema_t *sema);

void
sema_post(sema_t *sema);

void
sema_destroy(sema_t *sema);

int
sema_getvalue(sema_t *sema);
```

Locn : MultithreadingBible/Semaphores

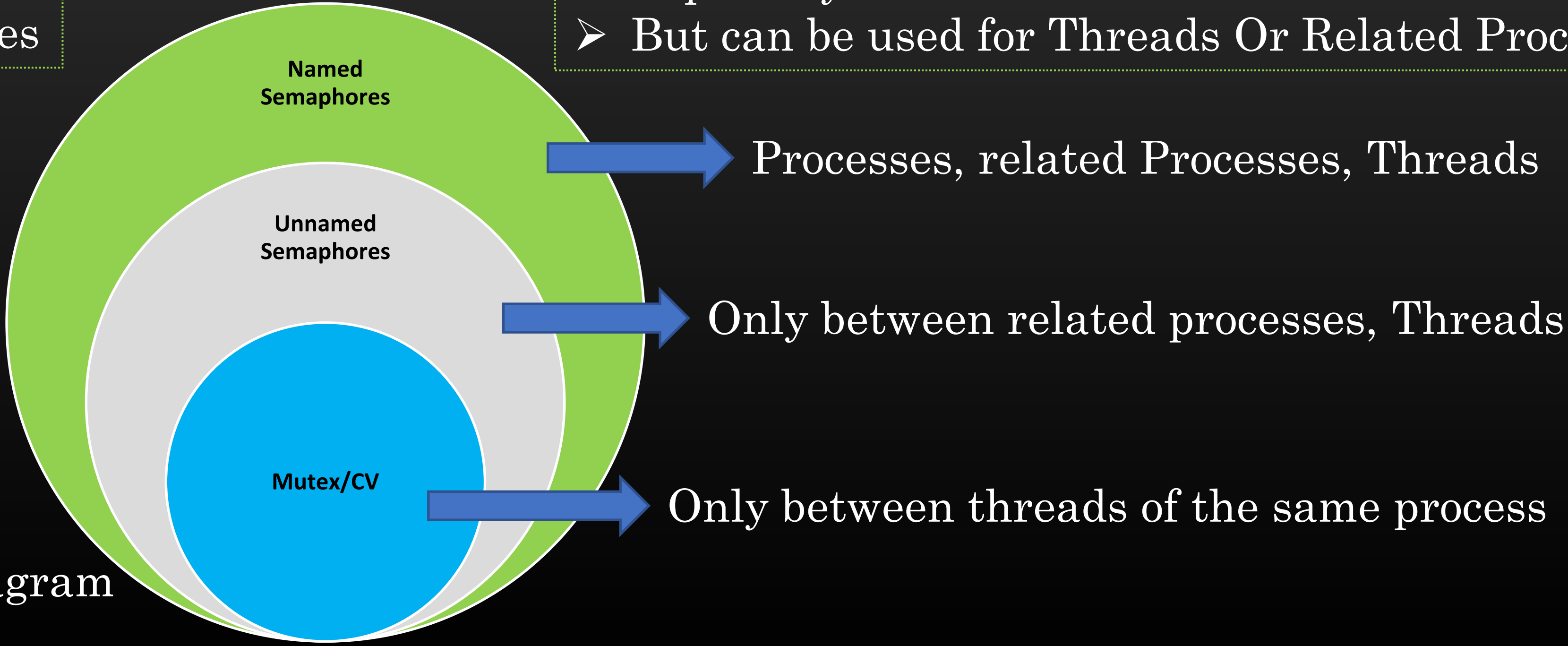
Semaphores

- Type of Semaphores :



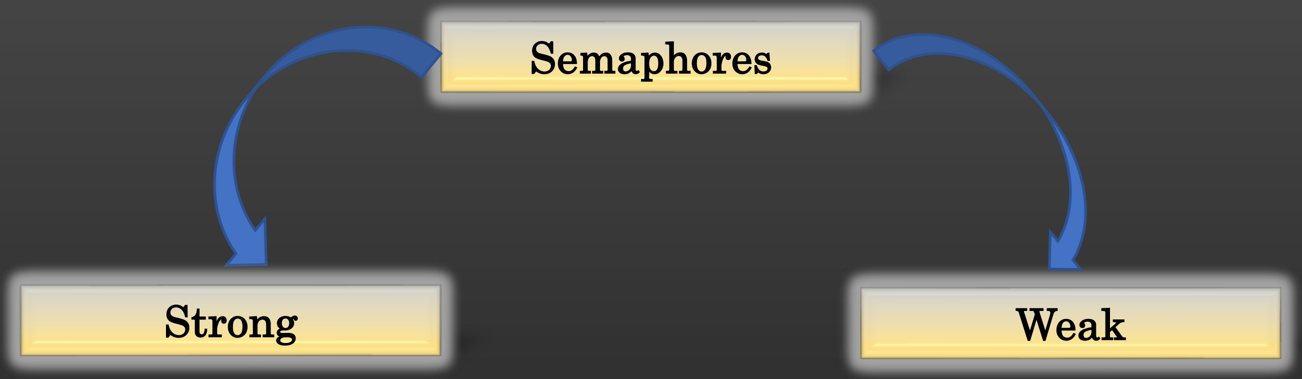
- Threads
- Related Processes

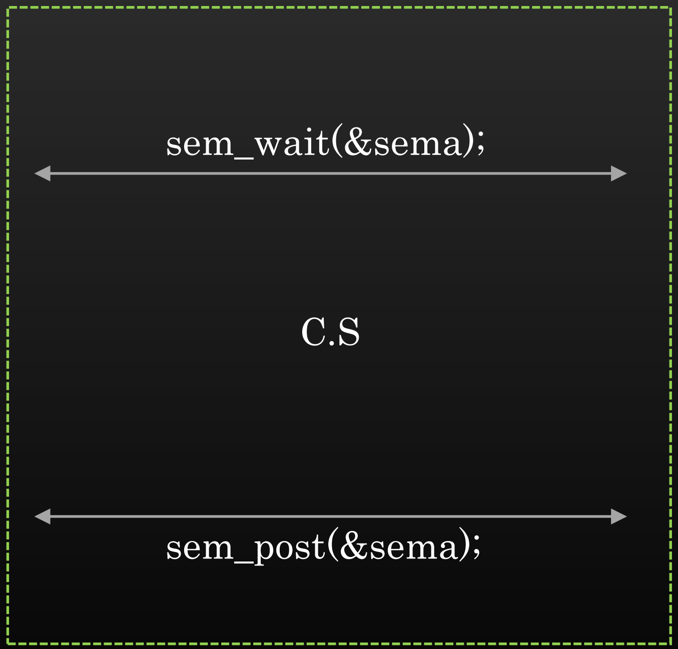
- Especially used for unrelated Processes
- But can be used for Threads Or Related Processes


Named Semaphores
Unnamed Semaphores
Mutex/CV
Processes, related Processes, Threads
Only between related processes, Threads
Only between threads of the same process

Scope Diagram

Semaphores → Strong and Weak Semaphores





- Whenever theoretically you can show in your solution that some thread blocked on mutex or semaphore may never get chance to resume execution (Starvation), then we say that solution is lacking the property of Bounded Waiting and such a semaphore is called Weak Semaphore
- We can deploy a way such that blocked threads are unblocked in a FIFO way per signal, then such a semaphore is called Strong Semaphore
- Bounded Waiting is a desirable property that any synchronization solution must have
- Weak Semaphores can be converted into strong semaphores by changing the policy of blocke thread selection from random to FIFO (Sequel Course)

Semaphores → Permit Parameter

sema_init(&sema, 0, n= 0)

sema_wait(&sema);

Only N execution units at a time !
N = n

sema_post(&sema);

```
S1   pthread_mutex_lock(&q_mutex);

S2     while (q_empty(q) ) {

S3       pthread_cond_wait(&t_cv, &q_mutex);
       }

S4      /* perform Ops on Queue (C.S) */

S5     pthread_mutex_unlock(&q_mutex);
```

- Semaphore can be initialized with non –ve integer
- Zero Semaphore (S0)
  - sem_init(&sem, 0, 0) : S0
  - Any call to sem_wait(&sem) will block the EU
  - sem_wait(&S0) = pthread_cond_wait(&cv, NULL)
  - sem_post(&S0) = acts as pthread_cond_signal(cv)
- Like Mutexes, Semaphores too do not have a provision for Conditional Critical Section Access, they allow entry into C.S only based on permit value

```
S13  sem_wait(&S);

S4   /* perform Ops on Queue (C.S) */

S5    sem_post(&S);
```

Semaphores → Mutex Vs Semaphores Vs Condition Variable

Mutexes	Semaphores	CV
Used for Mutual Exclusion	Used for Mutual Exclusion respecting the permit limit	Used for Thread Coordination
CS can be executed by at-most 1 EU	CS can be executed by at-most N EU	NA
No Provision for User defined predicate	No Provision for User defined predicate	Mutex + CV Provision for User defined predicate
Only threads	Threads Related Processes Un-related Processes (Named Semaphores) (Every where)	Only threads