Unix threads C++

Short introduction to threads (pthreads)

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>      include -pthread in g++
#include <unistd.h>

void * Runtime () {
        printf ("Hello from threads \n");
        Sleep (3);
        Printf ("Ending thread \n");
}

int main (int argc, char * argv[]) {

        pthread_t p1, p2;
        if (pthread_create (&p1, NULL, &Runtime, NULL) != 0) {
                return 1;
        }
        if (pthread_create (&p2, NULL, &Runtime, NULL) != 0) {
                return 2;
        }
        if (pthread_create
        if (pthread_join (p1, NULL) != 0) {
                return 3;
        }
        if (pthread_join (p2, NULL) != 0) {
                return 4;
        }
```

2 hr

thread vs process

thread

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

void* routine() {
    printf("Hello from threads\n");
}

int main(int argc, char* argv[]) {
    pthread_t t1;
    if (pthread_create(&t1, NULL,
                       &routine, NULL))
    { return 1;
    }
    if (pthread_join(t1, NULL)) {
        return 2;
    }
    return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char* argv[]) {
    int pid = fork();
    if (pid == -1) {
        return 1;
    }
    printf("Process id is %d\n", getpid());
    if (pid != 0) {
        wait(NULL);
    }
    return 0;
}
```

two diff Pid
= 25932
25938

if we get Pid in thread program
- same process id. [Multiple thread in same process]

Other difference :- [Address Space]

two diff Variables, Address Space

```
main {
    int x=2;
    int pid=fork();
    if (Pid==-1) {
        return 1;
    }
    if (pid==0) {
        x++;
    }
    Sleep (2);          // to make sure if change reflect it would
    Printf ("Value of x: %d \n", x);
    if (Pid != 0) {
        Wait (NULL);
    }
    return 0;
}
```

Value of x: 2
Value of x: 3 ← child process.

```
int x=2          ← global shared memory

Void * routine() {
    x++;
    Sleep(2);
    Printf ("Value of x: %d \n", x);
}

Void * routine2() {
    Sleep (2);
    Value ("Value of x : %d \n", x);
}
```

```
int main (int arg c, char* arg v [7]) {
    pthread_t t1, t2;
    if [pthread_Create [&t1, NULL, &routine, NULL])
    { return1;
    }
    if (Pthread_Create(&t2, NULL, &routine2, NULL))
    {
      return 2;
    }
    if (pthread_join (t1, NULL)) {
      return 3;
    }
    if (Pthreadjoin ( t2, NULL)) {
    } return 4;
    return 0;
}
```

Output

Value of x : 3.

Value of x : 3

Race Conditions

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

int mails = 0;

void* routine() {
    for (int i=0; i<1000000; i++) {
        mails++;
        // read mails
        // increment
    }   // write mails
}

int main (int argc, char* argv[]) {
    pthread_t p1, p2, p3, p4;
    if (pthread_create (&p1, NULL, &routine, NULL) != 0) {
        return 1;
    }
    if (pthread_create (&p2, NULL, &routine, NULL) != 0) {
        return 2;
    }
    if (pthread_join (p1, NULL) != 0) {
        return 3;
    }
    if (pthread_join (p2, NULL) != 0) {
        return 4;
    }
    printf ("Number of mails : %d \n", mails);
    return 0;
}
```

mutex in C

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

int mails = 0;
pthread_mutex_t mutex;

void* routine() {
    for (int i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&mutex);
        mails++;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char* argv[]) {
    pthread_t p1, p2, p3, p4;
    pthread_mutex_init(&mutex, NULL);
    if (pthread_create(&p1, NULL, &routine, NULL) != 0) {
        return 1;
    }
    if (pthread_create(&p2, NULL, &routine, NULL) != 0) {
        return 2;
    }
    if (pthread_create(&p3, NULL, &routine, NULL) != 0) {
        return 3;
    }
    if (pthread_create(&p4, NULL, &routine, NULL) != 0) {
        return 4;
    }
    if (pthread_join(p1, NULL) != 0) {
        return 5;
    }
```

```
if (pthread_join (p2, NULL) != 0) {
    return 6;
}
if (pthread_join (p3, NULL) != 0) {
    return 7;
}
if ( pthread_join (p4, NULL) != 0) {
    return 8;
}
pthread_mutex_destroy (&mutex);
Printf ("Number of mails : %d \n", mails);
return 0;
}
```

Create threads in a loop. (pthread_Create)

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int mails = 0;
pthread_mutex_t mutex;
Void * routine () {
    for (int i = 0; i < 10000000 ; i++) {
        pthread_mutex_lock (&mutex);
        mails++;
    } pthread_mutex_unlock (&mutex);
}
```

```
int main (int argc, char* argv[]) {
    ~~pthread_mutex_lock (&~~ pthread_t th[8];
    int i;
    pthread_mutex_init (&mutex, NULL);
    for (i=0; i<8; i++) {
        if (pthread_create (th+i, NULL, &routine, NULL) != 0) {
            perror ("failed to create thread"); return 1;
        }
        printf ("Thread %d has started \n", i);
    }
    for (i=0; i<8; i++) {
        if (pthread_join (th[i], NULL) != 0) {
            return 2;
        }
        printf ("Thread %d has finished execution \n", i);
    }
    pthread_mutex_destroy (&mutex);
    printf ("Number of mails = %d \n", mails);
    return 0;
}
```

How to return value from a thread [pthread_join]
How to pass arguments to threads in C.

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

int primes[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

void routine (void* arg) {
    sleep(1);
    int index = *(int*)arg;
    printf ("%d", primes[index]);
    free (arg);
}
```

```
int main (int argc, char* argv[]) {
    pthread_t th[10];
    int i;
    for (i=0; i<10; i++) {
        int* a = malloc (sizeof (int));      // Why cant we pass &i
        *a = i;
        if (pthread_create (&th[i], NULL, &routine, a) != 0)
            perror ("Failed to create a thread");
    }
    for (i=0; i<10; i++) {
        if (pthread_join (th[i], NULL) != 0) {
            perror ("Failed to Join thread");
        }
    }
    return 0;
}
```

How to return Value from a thread

```
void* roll_dice () {
    int value = (rand() % 6) + 1;
    int* result = malloc (sizeof (int));
    *result = value;
    printf ("Thread result : %p\n", result);
    return (void*) result;
}

int main (int argc, char* argv[]) {
    int* res;
    srand (time (NULL));
    pthread_t th;
    if (pthread_join (th, (void**) &res) != 0) {
        return 2;
```

```
printf ("Main res: %p\n", res);
printf ("Result: %d\n", *res);
free (res);
return 0;
}
```

Practical example for using threads #1 [Summing numbers from an array]

```
#include

int primes[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

void * routine (void * arg) {
    int index = *(int *) arg;
    int sum = 0;
    for (int j = 0; j < 5; j++) {
        sum += primes[index + j];
    }
    printf ("Local sum: %d\n", sum);
    *(int *) arg = sum;
    return arg;
}
```

```
int main (int argc, char *argv[]) {
    pthread_t th[2];
    int i;
    for (i=0; i<2; i++) {
        int *a = malloc (sizeof (int));
        *a = i*5;
        if (pthread_create (&th[i], NULL, &routine, a)!=0) {
            perror ("Failed to create thread");
        }
    }
    int globalSum = 0;
    for (i=0; i<2; i++) {
        int *r;
        if (pthread_join (th[i], (void **) &r)!=0) {
            perror ("Failed to create thread");
        }
        globalSum += *r;
        free (r);
    }
    printf ("Global Sum: %d \n", globalSum);
    return 0;
}
```

Difference between trylock and lock in C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

pthread_mutex_t mutex;

void* routine_lock(void *arg) {
        pthread_mutex_lock(&mutex);
        printf("Got lock\n");
        sleep(1);
        pthread_mutex_unlock(&mutex);
}

void* routine_trylock(void *arg) {
        if (pthread_mutex_trylock(&mutex)==0) {
         printf("Got lock\n");
         sleep(1); pthread_mutex_unlock(&mutex);
        }else {
          printf("Didn't get lock\n");
        }
}

int main(int argc, char* argv[]){
        pthread_t th[4];
        pthread_mutex_init(&mutex, NULL);
        for(i=0; i<4; i++) {
                pthread_create(&th[i], NULL, &routine_lock, NULL);
                &routine_trylock
                { perror("Error at creating thread");
                }
        }
}
```

```
    for(int i=0;i<4;i++) {
        if(pthread_join(th[i], NULL)!=0) {
            perror("Error at joining thread");
        }
    }
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

#Conditional Variables in C (looks like wait, notify, notifyall)

```
Pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel=0;

void* fuel_filling (void *arg) {
    for(int i=0;i<5;i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel+=15;
        Printf("Filled fuel ... %d \n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_signal(&condFuel); sleep(1); //to keep control on
    }                                                Production
}

void* car (void *arg) {
    pthread_mutex_lock(&mutexFuel);
    while(fuel<40) {
        printf("No fuel. Waiting ... \n");
        pthread_cond_wait(&condFuel, &mutexFuel);
        // Equivalent to:
        pthread_mutex_unlock(&mutexFuel);
        // Wait for signal on condFuel
        pthread_mutex_lock(&mutexFuel);
```

```
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}

int main(int argc, char* argv[]) {
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for(int i=0; i<2; i++) {
        if(i==1) {
            if(pthread_create(&th[i], NULL, &fuel_filling, NULL)
               !=0) {
                perror("Failed to create pthread");
            }
        } else {
            if(pthread_create(&th[i], NULL, &car, NULL)!=0){
                perror("Failed to create thread");
            }
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_mutex_destroy(&condFuel);
    return 0;
}
```

Signaling for Condition Variables (pthread_cond_signal vs pthread_cond_broadcast)

```
pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;

void* fuel_filling(void* arg) {
    for(int i=0; i<5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 30;
        printf("Filled fuel ... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_broadcast(&condFuel); // Here check what happen if signal
    }
    sleep(1);
}
```

```
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}

int main(int argc, char* argv[]) {
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for(int i=0; i<2; i++) {
        if(i==1) {
            if(pthread_create(&th[i], NULL, &fuel_filling, NULL)
               !=0) {
                perror("Failed to create pthread");
            } else {
                if(pthread_create(&th[i], NULL, &car, NULL)!=0){
                    perror("Failed to create thread");
                }
            }
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_cond_destroy(&condFuel);
    return 0;
}
```

Signaling for Condition Variables (pthread_cond_signal vs pthread_cond_broadcast)

```
pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel=0;

void* fuel_filling(void* arg) {
    for(int i=0; i<5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 30;
        printf("Filled fuel ... %d\n", fuel);
        pthread_mutex_unlock(&mutexFuel);
        pthread_cond_broadcast(&condFuel); // here check what happen if signal
    }
    sleep(1);
```

```
void* car(void* arg) {
    pthread_mutex_lock(&mutexFuel);
    while(fuel < 40) {
        printf("No fuel. Waiting...\n");
        pthread_mutex_wait(&condFuel, &mutexFuel);
        //equivalent to pthread_mutex_unlock(&mutexFuel);
        //wait for signal on condFuel
        // pthread_mutex_lock(&mutexFuel);
    }
    fuel += 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}

int main(int argc, char* argv[]) {
    pthread_t th[6];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for(int i=0; i<6; i++) {
        if(i==4 || i==5) {
            if(pthread_create(&th[i], NULL, &fuel_filling, NULL)!=0) {
                perror("Failed to create thread");
            }
        } else {
            if(pthread_create(&th[i], NULL, &car, NULL)!=0) {
                perror("Failed to create thread");
            }
        }
    }
    for(int i=0; i<6; i++) {
        if(pthread_join(th[i], NULL)!=0) {
            perror("Failed to join thread");
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_cond_destroy(&condFuel);
    return 0;
}
```

Practical example for pthread_mutex_trylock

```
//Chefs = threads
//Stove = Shared data (+ mutex)

pthread_mutex_t StoveMutex[4];
int StoveFuel[4] = {100, 100, 100, 100};

Void * routine (Void *args) {
    for (int i=0; i<4; i++) {
        if (pthread_mutex_trylock(&StoveMutex[i])==0){
            int fuelNeeded = (rand()%30);
            if (StoveFuel[i]-fuelNeeded<0) {
                Printf("No more fuel... going home\n");
            } else { StoveFuel[i] -= fuelNeeded;
                usleep(500000);
                Printf("Fuel left %d\n", StoveFuel[i]);
            }
        } else {
            if (i==3) {
                Printf("No Stove available yet, waiting...\n");
                Usleep(300000);
                i=0;
            }
        }
    }
}

int main (int argc, char * argv[]) {
    Srand(time(NULL));
    pthread_t th[10];
    for (int i=0; i<10; i++) {
        pthread_mutex_init(&StoveMutex[i], NULL);
    }
```

```
for (int i=0; i<10; i++) {
    if (pthread_join (th[i], NULL) != 0) {
        perror ("Failed to join thread");
    }
}

for (i=0; i<4; i++) {
    pthread_mutex_destroy (&storeMutex[i]);
}
return 0;
}
```

What is pthread_exit?

Used in main so that as return, kill all thread if not joined in main

{Another way then return take void*

and end process

pthread_return doesn't everything is controlled by process and it make sure all thread completes its work.

```
void * rollDice () {
    int value = (rand() % 6) + 1;
    int * result = malloc (sizeof (int));
    *result = value;
    sleep (2); // for slowness
    printf ("Thread result : %d \n", value);
    pthread_exit ((void *) result); // equivalent to return
}

int main (int argc, char * argv[]) {
    int * res;
    srand (time (NULL));
    pthread_t th;
    if (pthread_create (&th, NULL, &rollDice, NULL) != 0) {
        return 1;
    }
```

main

// pthread_exit(0); ← This will cause thread works to complete no further line be executed, but process will take care other thread does its work.

// return 0; ← Even this thread will end as well as other thread will be stopped as well

Memory leak baga bi

```
if (pthread_join(th, (void**)&res) != 0) {
    return 2;
}
printf("Result: %d\n", *res);
free(res);
return 0;
}
```

Introduction to barriers (pthread_barrier)

```
pthread_barrier_t barrier;
void* routine(void* args) {
    while(1) {
        Printf("Waiting at the barrier...\n");
        Sleep(1);
        pthread_barrier_wait(&barrier);
        printf("We passed the barrier\n");
        Sleep(1);
    }
}
int main(int argc, char* argv[]) {
    pthread_t th[10];
    int i;
    pthread_barrier_init(&barrier, NULL, 3);
    for(i=0; i<10; i++) {
        pthread_create(&th[i], NULL, &routine, NULL)!=0
        { perror("Failed to create thread");
```

→ | 3 thread

Only allowed when group of 3 is formed need to be multiple of 3.

```
for(i=0; i<10; i++) {
    if (pthread_join (th[i], NULL) != 0) {
        perror("Failed to join thread");
    }
}

pthread_barrier_destroy (&barrier);
return 0;
}
```

Practical example to barriers (pthread_barrier)

```
#define Thread_NUM 8

int dice_values[Thread_NUM];
int status[Thread_NUM] = {0};

pthread_barrier_t barrierRolledDice;
pthread_barrier_t barrierCalculated;

void* roll (void* args) {
    int index = *(int*) args;
    while(1) {
        dice_values[index] = rand()%32+1;
        pthread_barrier_wait (&barrierRolledDice);
        pthread_barrier_wait (&barrierRolledDice);
                              &barrierCalculated
        if (status[index] == 1) {
            printf ("(%d rolled %d) I won\n", index, dice_values[index]);
        } else {
            printf ("(%d rolled %d) I lost\n", index, dice_values[index]);
        }
    }
    free (args);
```

pthread.h
stdio.h
stdlib.h
unistd.h
string.h
time.h

Sundaram

```
int main (int argc, char * argv[]) {
    srand (time (NULL));
    pthread_t th [THREAD_NUM];
    int i;
    pthread_barrier_init ( &barrierRolledDice,
                           NULL, THREADNUM+1);
    pthread_barrier_init (&barrierCalculated,
                           NULL, THREADNUM+1);
    for (i=0; i<THREADNUM; i++) {
        int * a = malloc (sizeof (int));
        *a = i;
        if (pthread_create (&th[i], NULL, &roll,
                            (void*) a) != 0) {
            perror ("Failed to create thread");
        }
    }

    while (1) {
        pthread_barrier_wait (&barrierRolledDice);
        //Calculate winner.
        int max = 0;
        for (i=0; i< THREADNUM; i++) {
            if (dice_values [i] > max) {
                max = dice_values [i];
            }
        }

        for (i=0; i< THREAD_NUM; i++) {
            if (dice_values [i] == max) {
                status [i] = 1;
            } else {
                status [i] = 0;
            }
        }
        sleep (1);
```

```
        printf ("==New Round Starting ===\n");
        pthread_barrier_wait (&barrier_calculated);
    }

    for (i=0; i< THREAD_NUM; i++) {
        if (pthread_join (th[i], NULL) != 0) {
            perror ("Failed to Join thread");
        }
    }

    pthread_barrier_destroy (&barrier_RolledDice);
    pthread_barrier_destroy (&barrier_calculated);
    return 0;
}
```

What is pthread_t [id of a thread].

→ basically unsigned long in Sys ide

→ its id of a thread

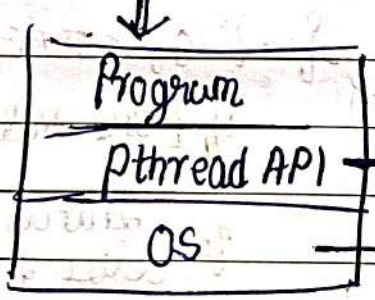
→ pthread_self() // get current thread

Treated as Opaque type
[Don't treat it as int, char etc]
Don't assume its of Specific type as done in below code.

Know how we get call to Know current process id. Same way we also have call to get current thread id

~~Pthread~~

Software Hierarchy



Read Documentation

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <sys/syscall.h>   (for other way gettid) os level

#define THREAD_NUM 2

void* routine (void* args) {
        pthread_t th = pthread_self();
        printf ("%ul\n", th);
        printf ("%d\n", (pid_t) syscall (SYS_gettid));
}

int main (int argc, char* argv[]) {
        pthread_t th = pthread_self();

        pthread_t th[THREAD_NUM];
        int i;
        for (i=0; i< THREAD_NUM; i++) {
                if (pthread_create (&th[i], NULL, &routine, NULL)!=0)
                { perror ("Failed to create thread");
                }  }

        for (i=0 ; i< THREAD_NUM; i++) {
                if (pthread_join (th[i], NULL)!=0) {
                        perror ("Failed to join thread");
                }  }

        return 0;
}
```

detached threads [detach from main thread]

```
#define THREAD_NUM 2
void * routine (void * args) {
        Sleep (1);
        printf ("Finished execution\n");
}

int main (int argc, char* argv[]) {
        pthread_t th [THREAD_NUM];
        pthread_attr_t detachedThread;
        pthread_attr_init (&detachedThread);
        pthread_attr_setdetachstate (&detachedThread, PTHREAD_CREATE_DETACHED);

        int i;
        for (i=0; i<THREAD_NUM; i++) {
            if (pthread_create (&th[i], &detachedThread, &routine, NULL) != 0) {
                perror ("Failed to create thread");
            }
        }  //pthread_detach (th[i]);  → to change to detach after creation as name suggest
        for (i=0; i<THREAD_NUM; i++) {
            if (pthread_join (th[i], NULL) != 0) {
                perror ("Failed to join thread");  //will print this as we cant join detach thread most time
            }
        }
        pthread_attr_destroy (&detachedThread);  // will deallocate own resources
        pthread_exit (0);  // know why... Process will wait for all other thread.
}
```

wrong

can land on join if one of the thread completes

attr when we have thread_detach its possible finish exec in between, and wait on join state. good doc.

Static Initializers in PTHREAD API.

→ Create detach while Creation

Pthread detach just means that you are never going to join thread again. This allows the pthread library to know whether it can immediately dispose of thread resources once the thread exits. (the detached case) or whether it just keep them around because you can later call pthread_join on the thread.

(It does not prevent the thread from being terminated if process terminate with exit or eq (main return 0)).

To allow other thread to continue execution, the main thread should rather than exit

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define THREAD_NUM 2
(If you are not customizing using pthread_mutex_init, destroy not necessary but good practice)
pthread_mutex_t mutexFuel = PTHREAD_MUTEX_INITIALIZER;   ← static initialize
pthread_cond_t condFuel = PTHREAD_COND_INITIALIZER;
                                  not for barrier as it need count.

void* routine (void* args) {

};
int main (int argc, char* argv[]) {
        pthread_t th[THREAD_NUM];
        int i;
        for (i=0; i<THREAD_NUM; i++) {
            if (pthread_create (&th[i], NULL, &routine, NULL) != 0) {
            } perror ("Failed to create thread");
        }
        for (i=0; i< THREAD_NUM; i++) {
            if (pthread_join (th[i], NULL) != 0) {
            } perror ("Failed to join thread");
        }
        return 0;
};
```

Deadlocks inc

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#include THREAD_NUM 8

pthread_mutex_t mutexFuel;
int fuel = 50;
pthread_mutex_t mutexWater;
int water = 10;

void *routine (void *args) {
    if (rand() % 2 == 0) {
        pthread_mutex_lock (&mutexFuel);
        sleep (1);
        pthread_mutex_lock (&mutexwater);
    } else {
        pthread_mutex_lock (&mutexwater);
        sleep(1);
        pthread_mutex_lock (&mutexfuel);
    }
    fuel += 50;
    water = fuel;
    printf ("Incremented fuel to: %d, Set water to
            %d \n", fuel, water);
    pthread_mutex_unlock (&mutexFuel);
    pthread_mutex_unlock (&mutexWater);
}
```

```
int main (int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_mutex_init(&mutexWater, NULL);
    int i;
    for (i=0; i< THREAD_NUM; i++) {
        if (pthread_create(&th[i], NULL, &routine, NULL)!=0)
        { perror("Failed to create thread");
    }  }
    Printf("Fuel : %d\n", fuel);
    Printf("water: %d\n", water);
    pthread_mutex_destroy(&mutexFuel);
    pthread_mutex_destroy(&mutexWater);
    return 0;
}
```

Requesting Same lock by same thread [Causes Deadlock in C++]
it will block

```
void *routine (void *args) {
    pthread_mutex_lock(&mutexFuel);
    pthread_mutex_lock(&mutexFuel); // block
}
```

If we take Water Fuel lock in same seq, no issue.

	Fuel Mutex	Water mutex
Thread 1	X (taken)	Wait
Thread 2	Wait	X

There can be other more mutex as well

~~Take~~ example.

Recursive Mutexes (Pritam asked something similar)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>          #include <string.h>
#include <sys/syscall.h>

# define THREAD_NUM 8

pthread_mutex_t mutexFuel;
int fuel = 50;

void * routine (void * args) {
        pthread_mutex_lock (&mutexFuel);
        pthread_mutex_lock (&mutexFuel);
        pthread_mutex_lock (&mutexFuel);
        pthread_mutex_lock (&mutexFuel);
        pthread_mutex_lock (&mutexFuel);
        fuel += 50;
        printf ("Incremented fuel to %d\n", fuel);
pthread_mutex_unlock (&mutexFuel); pthread_mutex_unlock (&mutexFuel);
}   ... x3

int main (int argc, char * argv[]) {

    pthread_t th [THREAD_NUM];
    pthread_mutexattr_t recursiveMutexAttributes;
    pthread_mutexattr_init (&recursiveMutexAttributes);
    pthread_mutexattr_settype (&recursiveMutexAttributes,
                               PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init (&mutexFuel, &recursiveMutexAttributes);
```

```
int i;
for (i=0; i< THREAD_NUM; i++) {
    if (pthread_create (&th[i], NULL, &routine, NULL) != 0) {
        perror ("Failed to create thread");
    }
}

for (i=0; i< THREAD_NUM; i++) {
    if (pthread_join (th[i], NULL) != 0) {
        perror ("Failed to join thread");
    }
}

printf ("Fuel: %d \n", fuel);
pthread_mutexattr_destroy (&recursiveMutexAttributes);
pthread_mutex_destroy (&mutexFuel);
return 0;
}
```

2
1 } if come back to 0 it considered unlock. (same time)
0

// You have to unlock same number of time.

(not same as semaphore can be lock and then unlock by other thread)

Introduction to Semaphores in C

(~~sem_wait~~ can not be used, sem_post) are used.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#include Thread_NUM 4;

sem_t semaphores;
```

— Mutex (Recursive mutex)
— Conditional
— Barrier (Detach thread)
— Semaphore

```
void * routine (void * args) {
    sem_wait (& semaphore);
    sleep(1);
    printf ("Hello from thread %d \n", *(int *)args);
    free(args);
}

int main (int argc, char * argv[]) {
    pthread_t th [THREAD_NUM];
    sem_init (& semaphore, 0, 4);
```

pShared argument indicates whether this semaphore is to be shared between the threads of the process, or between processes.

Value — Initial value for the semaphore

0 → shared between threads of a process, and located at some places visible to all threads. (global or Heap).

non-zero — semaphore shared between processes, and should be located at some address region of shared memory.

Shmget
mmap
Shm_open

(Since a child created by fork inherit its parent memory mappings, it can also access semaphore

Any process that can access the Sh memory region can operate on semaphore using sem_post, sem_wait and so on..

(Initializing a semaphore already initialized causes undefined behaviour)

```
    sem_destroy (& semaphore);
    return 0;
}
```

— Mutex
— Conditional
— Barrier
— Semaphore

(Recursive mutex
(Detach thread)

```
void * routine (void * args) {
    sem_wait (& semaphore);
    sleep(1);
    Printf ("Hello from thread %d \n", *(int *)args);
    free(args);
}

int main (int argc, char * argv[]) {
    pthread_t th [THREAD_NUM];
    sem_init (& semaphore, 0, 4);
```

pShared → argument indicates whether this semaphore is to be shared between the threads of the process, or between processes.

Value → Initial value for the semaphore

0 → shared between threads of a process, and located at some places visible to all threads. (global or Heap).

```
    sem_destroy (& semaphore);
    return 0;
}
```

Non-zero — semaphore shared between processes, and should be located at some address region of shared memory.

Shmget
mmap
Shm_open

(Since a child created by fork inherit its parent memory mappings, it can also access semaphore

Any process that can access the Sh memory region can operate on semaphore using sem_post, sem_wait and so on...

(Initializing a semaphore already initialized causes undefined behaviour

Sem_wait → S>0 → S-- [Continue execution]

Sem_post → S++

Now Thread 2

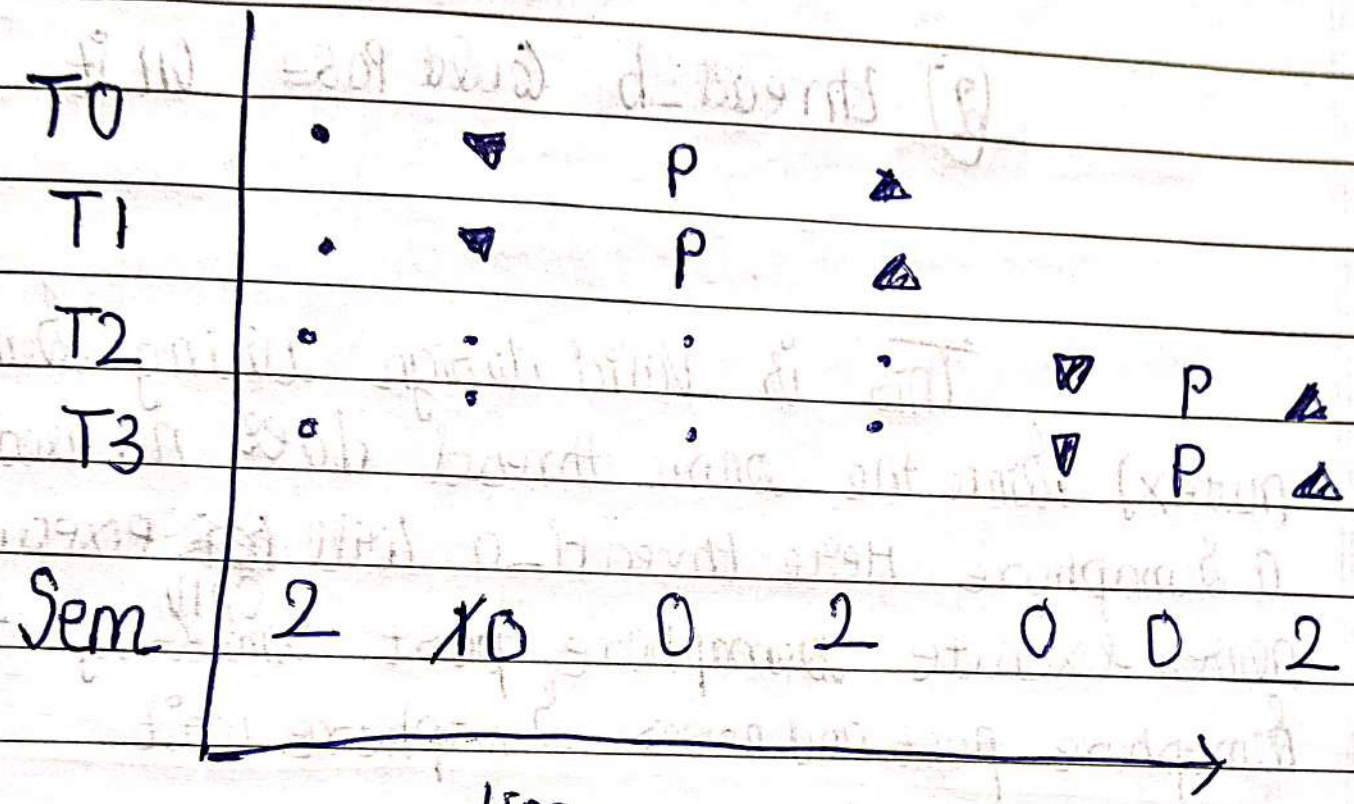
T0	▼	▼	P	▲	.	▼ = Sem_wait = P [S]	.	.	▼	P	▲
T1	.	.	.	.	.	▲ = Sem_post = V [S]	P	▲	.	.	.
T2	.	.	.	-	▲▼	P = Print	.	.			
T3	.	.	.	.	.						
Sem	1	0	0	1	0		0	1	0	0	

Same for Thread 3
Only one thread

```
Void * routine (void * args) {
    Sem_wait (& semaphore);
    Sleep (1);
    Printf ("Hello from thread %d \n", *(int *) args);
    Sem_post (& semaphore);
    free (args);
}

int main (int argc, char * argv[]) {
    pthread_t th [THREAD_NUM];
    Sem_init (& semaphore, 0, 1);   <-- What if we change it to 2
    int i;
    for (i=0; i < THREADNUM; i++) {
        int * a = malloc (sizeof (int));
        *a = i;
        if (pthread_create (& th[i], NULL, & routine, a) != 0) {
            perror ("Failed to create thread");
        }
    }
    loop
    Join
    Sem_destroy (& semaphore);
    return 0;
}
```

If 2 how its behaviour,



▼ = Sem wait = P(S)

▲ = Sem - post = V[S]

P - Print

Semaphore — mutex with counter. VS Recursive mutex
[across thread lock] [multiple lock at same mutex]

Difference:- between Multiple thread

* Comment Section pinned. [Important point) ✡✡✡

Semaphore VS mutex.

* Since Semaphores are almost always discussed along with mutex, another good difference to mention is that Semaphores are not necessarily owned by a thread. Let me take an example

Consider a Semaphore_a which has been initialized to have a value of 0.

(1) thread_a could wait on it (i.e it could decrement the Semaphore count)

(2) thread_b could post on it (i.e it will only increment the Semaphore count)

This is valid design using Semaphores (but not with mutex) since the same thread does no need to wait and also post a Semaphore. Here thread_a will only execute Semaphore wait & never execute Semaphore post. Similarly thread_b will only execute Semaphore post and never Semaphore wait.

(Above example can be mistaken to suggest a thread which execute Semaphore wait (& decrement Semaphore count), also need to post the Semaphore too (& increment Semaphore count).

)

However its not true with mutex. if a thread locks a mutex, the same thread need to unlock the mutex.

Practical Example using Semaphore

main.c

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define THREAD_NUM 16
sem_t semaphore;
void* routine(void* args) {
    printf("(%d) Waiting in the login queue\n", *(int*)args);
    sem_wait(&semaphore);
    printf("(%d) Logged in\n", *(int*)args); sleep(rand()%5+1);
    printf("(%d) Logged out\n", *(int*)args); sem_post(&semaphore);
    free(args);
}
int main(int argc, char* argv[]) {
    pthread_t th[THREAD_NUM];
    sem_init(&semaphore, 0, 5);
    int i;
    for(i=0; i<THREAD_NUM; i++) {
        int* a = malloc(sizeof(int));
        *a = i;
        if(pthread_create(&th[i], NULL, &routine, a)
        { perror("Failed to Create thread"); }
    }
```

Want to // Set limit

daram

```
for (i = 0; i < THREAD_NUM; i++) {
    if (pthread_join (th[i], NULL) != 0) {
        perror ("Failed to join thread");
    }
}
sem_destroy (&semaphore);
return 0;
}
```

ChatGPT

Understood the meaning of the function should not return an error code of [EINT

for instance, this might happen if program makes use of alarm() to run some code Asynchronously when a timer run out. If timeout occur while program is calling write() we must just retry the System call read/write etc.

many System calls will report the EINTR error code if a Signal error occured while System call was in progress.

/No error ~~code~~ actually occured it was just reported this way because the System is not able to resume call automatically.

Producer - Consumer problem in Multi-Threading

```
#define Thread_NUM 8

sem_t semEmpty;
sem_t semFull;
pthread_mutex_t mutexBuffer;

int buffer[10];
int count = 0;
```

```
void* producer (void *args) {
    while (1) {
        // Produce
        int x = rand() % 100;
        sleep(1);
        // Add to the buffer
        sem_wait (&semEmpty);
        pthread_mutex_lock (&mutexBuffer);
        buffer [count] = x;
        count++;
        pthread_mutex_unlock (&mutexBuffer);
        sem_post (&semFull);
    }
}

void * consumer (void *args) {
    while (1) {
        int y;
        // Remove from the buffer
        sem_wait (&semFull);
        pthread_mutex_lock (&mutexBuffer);
        y = buffer [count - 1];
        count--;
        pthread_mutex_unlock (&mutexBuffer);
        sem_post (&semEmpty);
        // Consume
        printf ("Got %d\n", y); sleep(1);
    }
}

int main (int argc, char *argv []) {
    srand (time (NULL));
    pthread_t th [THREAD_NUM];
    pthread_mutex_init (&mutexBuffer, NULL);
    sem_init (&semEmpty, 0, 10);
    sem_init (&semFull, 0, 0);
    int i;
```

```
for (i=0; i< THREAD_NUM; i++) {
        if (i>0) {
            if (pthread_create (&th[i], NULL, &producer, NULL) !=0) {
                perror ("Failed to create thread");
            }
        } else {
            if (pthread_create (&th[i], NULL, &consumer, NULL) !=0) {
                perror ("Failed to create thread");
            }
        }
}

for (i=0; i< Thread_NUM; i++) {
        if (pthread_join (th[i], NULL ) !=0) {
            perror ("Failed to join thread");
        }
}
Sem_destroy (&SemEmpty);
Sem_destroy (&SemFull);
pthread_mutex_destroy (&MutexBuffer);
return 0;
}
```

What are binary Semaphores

```
#define THREAD_NUM 1

Sem_t SemFuel;
pthread_mutex_t mutexFuel;
int* fuel;

void* routine(void* args) {
    *fuel += 50;
    printf("Current value is %d\n", *fuel);
    sem_post(&SemFuel);
}

int main(int argc, char* argv[]) {
    pthread_t th[THREAD_NUM];
    fuel = malloc(sizeof(int));
    *fuel = 50;
    pthread_mutex_init(&mutexFuel, NULL);
    sem_init(&SemFuel, 0, 0);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_create(&th[i], NULL, &routine,
                           NULL) != 0) {
            perror("Failed to create thread");
        }
    }
    sem_wait(&SemFuel);
    printf("Deallocating Fuel\n");
    free(fuel);
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    sem_destroy(&SemFuel);
    return 0;
}
```

Difference between Binary Semaphores and Mutexes

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>
#define THREAD_NUM 4
```

Code on website

(Whi things which were discussed)

↓

undefined behaviour

Mutex → Semaphore

(undef) ~~Under~~ Across thread

Getting the value of a Semaphore

```
#define THREAD_NUM 4
sem_t sem;

void* routine (void* args) {
    int index = *(int*) args;
    int semVal;
    sem_wait (&sem);
    sleep (index+1);
    sem_getvalue (&sem, &semVal);
    printf ("(%d) Current Semaphore value after wait is
            %d \n", index, semVal);
    sem_post (&sem);
    sem_getvalue (&sem, &semval);
    printf ("(%d) Current Semaphore value after post is %d \n",
                index, semVal);
    free (args);
}
int main (int argc, char* argv[]) {
    pthread_t th [THREAD_NUM];
    sem_init (&sem, 0, 4);
    int i;
    for (i=0; i< THREAD_NUM; i++) {
        int *a = malloc (sizeof (int));
        *a = i;
        if (pthread_create (&th[i], NULL, &routine, a) != 0
        {
            perror ("Failed to create thread");
        }
    }
    for (i= 0; i< THREAD_NUM; i++) {
        if (pthread_join (th[i], NULL) != 0) {
            perror ("Failed to join thread");
        }
    }
    sem_destroy (&sem);
    return 0;
}
```

Parallelism vs. Concurrency

If enough CPU Power its Parall

parallelism.c

```
#define THREAD_NUM 2
```


```
int primes[10] = {2,3,5,7,11,13,17,19,23,29};
```

Concurrency, Context Switching may be costly.

```
void* routine (void* arg) {
    int index = *(int*) arg;
    sleep(1);
    for (int j = 0; j<5; j++) {
        sum += primes[index+j];
    }
    printf ("Local Sum : %d \n", sum);
    *(int*) arg = sum;
    return arg;
}

int main (int argc, char* argv[]) {
    pthread_t th[thread_NUM];
    int i;
    for (i=0; i < THREAD_NUM; i++) {
        int* a = malloc (sizeof (int));
        *a = i*5;
        if (pthread_create (&th[i], NULL, &routine, a) != 0) {
            perror ("Failed to join thread");
        }
    }
    int globalSum = 0;
    for (i=0; i<THREAD_NUM; i++) {
```

```
int *r;
if (pthread_join(th[i], (void**)&r) != 0){
    perror("Failed to join thread");
}
globalSum += *r;
free(r);
}
printf("Global Sum = %d\n", globalSum);
return 0;
}
```

Concurrency.c

```
#define THREAD_NUM 16
int mails = 0;
pthread_mutex_t mutex;

void* routine(){
    for (int i=0; i<10; i++){
        pthread_mutex_lock(&mutex);
        mails++;
    } printf("Processing mail %d\n", mails); sleep(1);
        pthread_mutex_unlock(&mutex);
}

int main(int argc, char* argv[]){
    pthread_t th[THREAD_NUM];
    pthread_mutex_init(&mutex, NULL);
    int i;
    for ____________
        pthread_create
    for ____________
        pthread_join
    pthread_mutex_destroy(&mutex);
    printf("Number of mails: %d\n", mails);
    return 0;
}
```

Thread Pools in C [using pthread API]

```
#define THREAD_NUM 4

typedef struct Task {
    int a, b;
} Task;

Task taskQueue[256];
int taskCount = 0;

pthread_mutex_t mutexQueue;
pthread_cond_t condQueue;

void executeTask(Task* task) {
    usleep(50000);
    int result = task->a + task->b;
    printf("The sum of %d and %d is %d\n", task->a, task->b,
           result);
}

void submitTask(Task task) {
    pthread_mutex_lock(&mutexQueue);
    taskQueue[taskCount] = task;
    taskCount++;
    pthread_mutex_unlock(&mutexQueue);
    pthread_cond_signal(&condQueue);
}
```

Thread Pools in C [using pthread API]

Condition variable
- wait
- signal

mutex
- lock
- unlock

semaphore
- wait
- post

```
#define THREAD_NUM 4

typedef struct Task {
    int a, b;
} Task;

Task taskQueue[256];
int taskCount = 0;

pthread_mutex_t mutexQueue;
pthread_cond_t condQueue;

void executeTask(Task* task) {
    usleep(50000);
    int result = task->a + task->b;
    printf("The sum of %d and %d is %d\n", task->a, task->b,
           result);
}

void submitTask(Task task) {
    pthread_mutex_lock(&mutexQueue);
    taskQueue[taskCount] = task;
    taskCount++;
    pthread_mutex_unlock(&mutexQueue);
    pthread_cond_signal(&condQueue);
}
```

```
void * StartThread (void* args) {
    while (1) {
        Task task;
        pthread_mutex_lock (&mutexQueue);
        while (taskCount == 0) {
            pthread_cond_wait (&condQueue, &mutexQueue);
        }
        task = taskQueue[0];
        int i;
        for (i = 0; i < TaskCount - 1; i++) {
            taskQueue[i] = taskQueue[i+1];
        }
        taskCount--;
        pthread_mutex_unlock (&mutexQueue);
        executeTask (&task);
    }
}

int main (int argc, char* argv[]) {
    pthread_t th[threadNUM];
    pthread_mutex_init (&mutexQueue, NULL);
    pthread_cond_init (&condQueue, NULL);
    int i;
    for (i=0; i < ThreadNUM; i++) {
        if (pthread_create (&th[i], NULL, &StartThread, NULL)
            != 0) {
            perror ("Failed to Create the thread");
        }
    }
    Srand time (NULL);
    for (i=0; i<100; i++) {
        Task t = {
            .a = rand()%100;
            .b = rand()%100;
        };
        SubmitTask (t);
    }
    for (i=0; i < THREAD_NUM; i++) {
        if (pthread_join (th[i], NULL) != 0) {
            perror ("Failed to Join the thread");
        }
    }
    pthread_mutex_destroy (&mutexQueue);
    pthread_cond_destroy (&condQueue);
    return 0;
}
```

Cond t ← to release lock or wait
diff with Semaphore

Thread pool with Function pointers in C

```
#define THREAD_NUM 4

typedef struct Task {
    void (*taskFunction) (int, int);
    int arg1, arg2;
} Task;

Task taskQueue [256];
int taskCount = 0;

Pthread_mutex_t mutexQueue;
Pthread_Cond_t condQueue;

Void Sum (int a, int b) {
    usleep (50000)
    int Sum = a+b;
    Printf ("The Sum of %d and %d is %d \n", a, b,
                                        Sum);
}

Void Product (int a, int b) {
    usleep (50,000);
    int prod = a*b;
    printf ("Product of %d and %d is %d \n", a, b,
                                        Prod);
}
```

```
void executeTask (Task *task) {
    task->taskFunction (task->arg1, task->arg2);
}
void SubmitTask (Task task) {
    pthread_mutex_lock (&mutexQueue);
    taskQueue [taskCount] = task;
    taskCount++;
    pthread_mutex_unlock (&mutexQueue);
    pthread_cond_signal (&condQueue);
}

void* startThread (void *args) {
    while (1) {
        Task task;
        pthread_mutex_lock (&mutexQueue);
        while (taskCount == 0) {
            pthread_cond_wait (&condQueue, &mutexQueue);
        }
        task = taskQueue [0];
        int i;
        for (i=0; i<taskCount-1; i++) {
            taskQueue [i] = taskQueue [i+1];
        }
        taskCount--;
        pthread_mutex_unlock (&mutexQueue); executeTask (&t
    }
}
                    ↑ As less as possible in Critical Section

int main (int argc, char *argv[]) {
    pthread_t th [threadNum];
    pthread_mutex_init (&mutexQueue, NULL);
    pthread_cond_init (&condQueue, NULL);
    int i;
```

```
for (i=0; i<THREAD_NUM; i++) {
    if (pthread_create(&th[i], NULL, &startThread, NULL) != 0) {
        perror("Failed to create thread");
    }
}
srand(time(NULL));
for (i=0; i<100; i++) {
    if (pthread_join(th[i], NULL) != 0) {
        perror("Failed to join the thread");
    }
}
pthread_mutex_destroy(&mutexQueue);
pthread_cond_destroy(&condQueue);
return 0;
}

for (i=0; i<100; i++) {
    Task t = {
        .taskFunction = i%2==0 ? &sum : &product;
        .arg1 = rand() % 100;
        .arg2 = rand() % 100;
    };
    submitTask(t);
}
```