All Solutions

1. Week 2

a) Write a program that accepts an integer **num** as command line argument. In your program create two processes. One process should report the **num!** (Factorial of **num**). The other process should report the summation of all factorials till **num**. [eg: if the argument is 5, one process calculates 5!=120, another process calculates 1!+2!+3!+4!+5!=153]. *Each process should report its pid, ppid, and child id.*

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

int num;

int fact(int n){
    if (!n)
        return 1;
    else
        return (n*fact(n-1));
}

int main(int argc, char **argv) {
    pid_t returnval;

    int childretval; // from child process: user provided return value
    int exitstatus, i=0;  // parent process : child's exit status
    int res=0, sum=0;

    if (argc != 2){
        printf("\n USAGE: <PRO2 arg1>\n");
        exit(1);
    }

    num=atoi(argv[1]);
```

```
    if ((returnval=vfork()) >= 0) {    // including actions to be
performed by both parent and child processes
    if (returnval == 0) { // inside child
        printf("CHILD:: fork() returnValue = %d, My PID = %d, my
Parent's ID = %d\n",returnval,getpid(), getppid());
//globalvble++; localvble++;
        res=fact(num);
        printf("\nThe factorial of %d is %d \n ",num, res);
        exit(childretval);
    }
    else if (returnval > 0) { // inside parent
        printf("PARENT:: My PID = %d, my Child's PID = %d, my parent's
PID = %d \n",getpid(), returnval, getppid());
        for (int i=1; i<=num; i++){
            sum+=fact(i);
    }
        printf("The Sum of factorials till %d is %d \n ",num, sum);
        wait(&exitstatus);
        printf("PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exitstatus));
        printf("PARENT:: Good Bye\n\n");
        exit(0); // parent process is terminating -- normally
    }
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
    }
}
```

b) Write a program to create **exactly 3 processes**. Each process should then display its own pid along with the pid of the other two processes and tis relation with the other two processes [grandparent/parent/child, parent/child-parent/child, parent/child/grandchild]. Take care so that you don't have any unwanted orphan/zombie process(es). *Each process should report its pid, ppid, and child id.*

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

```c
#include <unistd.h>
#include <sys/wait.h>

int num;

int main(int argc, char **argv) {
    pid_t ret1, ret2, gp;
    int exitstatus, exst, i=0;  // parent process : child's exit status
    gp=getpid();

    if ((ret1=fork()) >= 0) {    // including actions to be performed
by both parent and child processes
        if (ret1 == 0) { // inside child
            printf("2nd Gen Parent, 1st Gen Child:: fork() returnValue
= %d, My PID = %d, my Parent's ID = %d\n",ret1,getpid(), getppid());
            if ((ret2=fork()) >= 0) {
                if (ret2 == 0) { // inside grandchild
                    printf("2nd Gen Child:: fork() returnValue = %d, My
PID = %d, my Parent's ID = %d my Grand Parent's ID =
%d\n",ret2,getpid(), getppid(), gp);
                    exit(2);
                }
            }
        wait(&exst);
        printf("2nd Gen PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exst));
        printf("2nd Gen PARENT:: Good Bye\n\n");
        exit(1);
        }
        else if (ret1> 0) { // inside parent
            wait(&exitstatus);
            printf("1st Gen PARENT:: My PID = %d, my Child's PID = %d,
my parent's PID = %d \n",getpid(), ret1, getppid());
            printf("1st Gen PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exitstatus));
            printf("1st Gen PARENT:: Good Bye\n\n");
            exit(0); // parent process is terminating -- normally
        }
```

```
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
    }
}
```

c) Write a program that accepts two integers (**low, high**) as command line argument. Create two processes in your program. The first process should calculate the summation of all integers between (**low, high**) as **sum_res**. The second process should evaluate whether **sum_res** is prime or not. *Each process should report its pid, ppid, and child id.*

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

int low, high, sumres;

int main(int argc, char **argv) {
    pid_t returnval;

    int childretval; // from child process: user provided return value
    int exitstatus, i=0;  // parent process : child's exit status
    int res=0, sum=0;

    if (argc != 3){
        printf("\n USAGE: <PRO2 arg1 arg2>\n");
        exit(1);
    }

    low=atoi(argv[1]);
    high=atoi(argv[2]);

    if ((returnval=vfork()) >= 0) {    // including actions to be
performed by both parent and child processes
    if (returnval == 0) { // inside child
```

```
        printf("CHILD:: fork() returnValue = %d, My PID = %d, my
Parent's ID = %d\n",returnval,getpid(), getppid());
//globalvble++; localvble++;
        for (int i=low; i<=high; i++){
            sumres+=i;
        }
        printf("The Summation of numbers from %d till %d is %d \n
",low, high, sumres);
        exit(childretval);
    }
    else if (returnval > 0) { // inside parent
        wait(&exitstatus);
        int fl=0;
        printf("PARENT:: My PID = %d, my Child's PID = %d, my parent's
PID = %d \n",getpid(), returnval, getppid());
        for (int i=2; i<=(sumres/2); i++){
            if (!(sumres % i)){
                printf("%d is not prime \n", sumres);
                fl=1;
                break;
            }
        }

        if (!fl)
            printf("%d is prime \n", sumres);

        printf("PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exitstatus));
        printf("PARENT:: Good Bye\n\n");
        exit(0); // parent process is terminating -- normally
    }
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
    }
}
```

d) Write a program to create **exac**
and max) as command line argum
within the range of min-max. The seco
within the range min-max. The third process should report the summation of even numbers and prime numbers within the range min-max.

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

int min, max, num;

void evod(){
    for(int i=min;i<=max;i++){
    if(!(i%2)){
        printf("%d is even \n",i);
    }
    else
    {
        printf("%d is odd \n",i);
    }
    }

}

int check(int n){
    int i;
    for(i=2;i<=n/2;i++){
        if(n%i)
            continue;
        else
            return 1;
    }
    return 0;
}

void prime(){
    for(int i=min;i<=max;i++){
```

```
        int flag=check(i);
        if (!flag) {
            printf("%d is : PRIME \n",i);
        }
        else {
            printf("%d is : NOT PRIME \n ",i);
        }
    }
}

void sum(){
    int sumprime=0, sumev=0;
    for(int i=min; i<=max; i++){
    int flag=check(i);
    if (!flag)
        sumprime+=i;
    else
        continue;
    }

    for(int i=min; i<=max; i++){
        if(!(i%2))
            sumev+=i;
    }

    printf("The sum of even numbers is %d \n",sumev);
    printf("The sum of prime and even numbers is %d
\n",sumprime+sumev);
}
int main(int argc, char **argv) {
    pid_t ret1, ret2, gp;
    int exitstatus, exst, i=0;  // parent process : child's exit status
    gp=getpid();

    if (argc != 3){
        printf("\n USAGE: <PRO2 arg1 arg2>\n");
        exit(1);
    }
```

```
    min=atoi(argv[1]);
    max=atoi(argv[2]);

    if ((ret1=fork()) >= 0) {    // including actions to be performed
by both parent and child processes
        if (ret1 == 0) { // inside child
            printf("2nd Gen Parent, 1st Gen Child:: fork() returnValue
= %d, My PID = %d, my Parent's ID = %d\n",ret1,getpid(), getppid());
            if ((ret2=fork()) >= 0) {
                if (ret2 == 0) { // inside grandchild
                    printf("2nd Gen Child:: fork() returnValue = %d, My
PID = %d, my Parent's ID = %d my Grand Parent's ID =
%d\n",ret2,getpid(), getppid(), gp);
                    evod();
                    exit(2);
                }
            }
        wait(&exst);
        printf("2nd Gen PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exst));
        prime();
        printf("2nd Gen PARENT:: Good Bye\n\n");
        exit(1);
        }
        else if (ret1> 0) { // inside parent
            wait(&exitstatus);
            printf("1st Gen PARENT:: My PID = %d, my Child's PID = %d,
my parent's PID = %d \n",getpid(), ret1, getppid());
            printf("1st Gen PARENT:: Child's exit code is -> %d
\n",WEXITSTATUS(exitstatus));
            sum();
            printf("1st Gen PARENT:: Good Bye\n\n");
            exit(0); // parent process is terminating -- normally
        }
    } // outer-if complete....
    else {
        printf ("Child creation error....");
        exit(0);
```

```
    }
}
```

2. Week 3

a) Write a program that accepts an integer **num** as command line argument. In your program create two threads. One thread should report the **num!** (Factorial of **num**). The other thread should report the summation of all factorials till **num**. [eg: if the argument is 5, one thread calculates 5!=120, another thread calculates 1!+2!+3!+4!+5!=153]. *Each process should report its own id and caller's id. When a thread ends, it should display meaningful message.*

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

int num;

int fact(int n){
    if (!n)
        return 1;
    else
        return (n*fact(n-1));
}

//factorial thread
void *thdfact(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;
    sid=pthread_self();
    printf("\nFactorial Thread with ID: %lu called by : %lu", sid,
thd_id);
    int res = fact(num);
    printf("The factorial of %d is %d \n ",num, res);
    pthread_exit(NULL);
}
```

```c
//sum thread
void *thdsum(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;
    sid=pthread_self();
    printf("\nSummation Thread with ID: %lu called by : %lu",
sid,thd_id );
    int sum = 0;
    for (int i=1; i<=num; i++){
        sum+=fact(i);
    }
    printf("The Sum of factorials till %d is %d \n ",num, sum);
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int t1, t2;
    pthread_t sid1, tid1, tid2;

    num=atoi(argv[1]);
    sid1=pthread_self();

    if ((t1=pthread_create(&tid1, NULL, thdfact, (void *)&sid1))){
        printf("ERROR %d in creating Thread!\n", t1);
        exit(1);
    }

    if ((t2=pthread_create(&tid2, NULL, thdsum, (void *)&sid1))){
        printf("ERROR %d in creating Thread!\n", t2);
        exit(1);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

b) Write a program that accepts tw
threads in your program. The first th
(**low, high**) as **sum_res**. The second thread should evaluate whether **sum_res** is prime or not.
Each process should report its own id and caller's id. When a thread ends, it should display meaningful message.

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

int low, high, sumres;


//prime thread
void *thdprime(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;
    int flag=0;

    sid=pthread_self();
    printf("\nPrime Checker Thread with ID: %lu called by : %lu", sid,
thd_id);

    for (int i=2; i<=(sumres/2); i++){
        if (!(sumres % i)){
            printf("%d is not prime \n", sumres);
            flag=1;
            break;
        }
    }

    if (!flag)
        printf("%d is prime \n", sumres);

    pthread_exit(NULL);
```

```
}

//sum thread
void *thdsum(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;
    sid=pthread_self();
    printf("\nSummation Thread with ID: %lu called by : %lu",
sid,thd_id );

    for (int i=low; i<=high; i++){
        sumres+=i;
    }
    printf("The Summation of numbers from %d till %d is %d \n ",low,
high, sumres);
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int t1, t2;
    pthread_t sid1, tid1, tid2;

    if (argc != 3){
        printf("\n USAGE: <PRO2 arg1 arg2>\n");
        exit(1);
    }
    low=atoi(argv[1]);
    high=atoi(argv[2]);
    sid1=pthread_self();

    if ((t1=pthread_create(&tid1, NULL, thdsum, (void *)&sid1))){
        printf("ERROR %d in creating Thread!\n", t1);
        exit(1);
    }
    pthread_join(tid1, NULL);

    if ((t2=pthread_create(&tid2, NULL, thdprime, (void *)&sid1))){
        printf("ERROR %d in creating Thread!\n", t2);
```

```
        exit(1);
    }
    pthread_join(tid2, NULL);
    return 0;
}
```

3. Week 4

a) Write a program [named **demo.c**] that accepts two integers (**low, high**) as command line argument. **demo.c** in turn should call two other programs **pro1** and **pro2**.

pro1 should calculate the summation of all integers between (**low, high**) as **sum_res**.

pro2 should evaluate whether **sum_res** is prime or not.

Demo.c

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

int main(int argc, char *argv[]){
    pid_t pid;
    int sumres;
    char arg_sumres[8];
    if ((pid = fork()) == 0){ // child process
        printf("\nI am Child, my ID %d, MY PARENT ID = %d", getpid(),
getppid());
        printf("\n");
        execl("w4pro1", "w4pro1", argv[1], argv[2], (char *)NULL);
        exit(0);
    }

    else if (pid > 0){ // parent process
        int sumres;
        //sleep(5);
        waitpid(pid, &sumres, 0);
        sumres = WEXITSTATUS(sumres);
        printf("\nResult from PRO1: %d", sumres);
```

```
        printf("\n\nI am Parent, my ID %d, I have child with ID: %d, My
ParentID = %d", getpid(), pid, getppid());
        printf("\n");
        sprintf(arg_sumres, "%d", sumres);  // equivalent to
itoa(sumres,arg_sumres,10);
        execl("w4pro2", "w4pro2", arg_sumres, (char *)NULL);
        exit(0);
    }

    else
        printf("Fork() ERROR!!\n");
    return 0;
}
```

Pro1.c

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

int main(int argc, char *argv[]){
    int low, high, sumres = 0;
    if (argc != 3){
        printf("\n USAGE: <PRO1 arg1 arg2>\n");
        exit(1);
    }

    low = atoi(argv[1]);
    high = atoi(argv[2]);

    printf("\nReceived %d AND %d from Caller", low, high);
    printf("\n");
    for (int i = low; i <= high; i++)
        sumres = sumres + i;

    printf("Sum of series is: %d\n",sumres);
    printf("\n");
    return sumres;
}
```

Pro2.c

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

int main(int argc, char *argv[]){
    int sumres = 0, flag = 0;

    if (argc != 2){
        printf("\n USAGE: <PRO2 arg1>\n");
        exit(1);
    }

    sumres = atoi(argv[1]);

    for (int i=2; i<=(sumres/2); i++){
        if (!(sumres % i)){
            printf("%d is not prime \n", sumres);
            flag = 1;
            break;
        }
    }

    if (!flag)
        printf("%d is prime \n", sumres);
}
```

```
I am Child, my ID 1358, MY PARENT ID = 1357

Received 1 AND 10 from Caller
Sum of series is: 55


Result from PRO1: 55

I am Parent, my ID 1357, I have child with ID: 1358, My ParentID = 10
55 is not prime
```

b) Write a program to illustrate the **Bounded Buffer** or **Producer/Consumer** problem.

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int sem4=1;

int buff[10];
int num_item=0, max_item=10;

int b, p=10, c=10;


//PRODUCER
void *producer(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;

    while(p--){ sleep((rand()%2));
    if (num_item!=max_item && sem4==1){//start CS
        pthread_mutex_lock(&mutex1);
        sem4--;
        sid=pthread_self();
        buff[num_item]=num_item+1;
        printf("\nProducer with ID: %lu", sid);
        num_item++;
        printf("\nStack after producing : ");
        for (b=0; b<num_item; b++)
            printf("%d ", buff[b]);
        printf("\n");
        sem4++;
        pthread_mutex_unlock(&mutex1);
    //end CS
    }
```

```
    else{
        if (!sem4){// == 0)
            printf("\n Wait for Consumer to finish CS!");
            printf("\n");
        }
        else{
            printf("\nOVERFLOW!! Consumer not taking Items!");
            printf("\n");
        }
    }
    }
    pthread_exit(NULL);
}

//CONSUMER
void *consumer(void *arg){
    pthread_t thd_id=(pthread_t)arg;
    pthread_t sid;

    while(c--){ sleep((rand()%2));
    if (num_item>0 && sem4==1){ //start CS
        pthread_mutex_lock(&mutex1);
        sem4--;
        sid=pthread_self();
        printf("\nConsumer with ID: %lu", sid);
        printf("\n");
        num_item--;
        buff[num_item] = 0;
        printf("\nStack after consuming : ");
        for (b=0; b<num_item; b++)
            printf("%d ", buff[b]);
        printf("\n");
        sem4++;
        pthread_mutex_unlock(&mutex1);
    //end CS
    }
    else{
        if (!sem4){//==0)
```

```
            printf("\nWait for Producer to finish CS!");
            printf("\n");
        }
        else{
            printf("\nUNDERFLOW!! Producer Stack Empty!");
            printf("\n");
        }
    }
    }
    pthread_exit(NULL);
}

int main(){
    int t1, t2;
    pthread_t tid1, tid2;

    if ((t1=pthread_create(&tid1, NULL, consumer, (void *)&tid1))){
        printf("ERROR %d in creating Thread!\n", t1);
        exit(1);
    }
    if ((t2=pthread_create(&tid2, NULL, producer, (void *)&tid2))){
        printf("ERROR %d in creating Thread!\n", t2);
        exit(1);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

```
Producer with ID: 140682471028480
Stack after producing : 1

Consumer with ID: 140682479421184

Stack after consuming :
```

```
Producer with ID: 140682471028480
Stack after producing : 1

Producer with ID: 140682471028480
Stack after producing : 1 2

Consumer with ID: 140682479421184

Stack after consuming : 1

Consumer with ID: 140682479421184

Stack after consuming :

UNDERFLOW!! Producer Stack Empty!

Producer with ID: 140682471028480
Stack after producing : 1

Consumer with ID: 140682479421184

Stack after consuming :

UNDERFLOW!! Producer Stack Empty!

Producer with ID: 140682471028480
Stack after producing : 1

Producer with ID: 140682471028480
Stack after producing : 1 2

Consumer with ID: 140682479421184

Stack after consuming : 1

Producer with ID: 140682471028480
Stack after producing : 1 2
```

```
Consumer with ID: 140682479421184

Stack after consuming : 1

Producer with ID: 140682471028480
Stack after producing : 1 2

Consumer with ID: 140682479421184

Stack after consuming : 1

Producer with ID: 140682471028480
Stack after producing : 1 2

Wait for Producer to finish CS!

Producer with ID: 140682471028480
Stack after producing : 1 2 3
```

c) Write a multi-threaded program such that it creates 10 more threads. Each thread should print ***Hello from nth thread*** along with the argument received from the main thread. Each thread should return its own id and a unique value to main using **`pthread_exit()`**. Main thread should be able to print this returned message identifying which thread ended within its own thread. Use mutex for synchronization and compare the results with the previous week's assignment.

i. Synchronize it such that all 10 threads are created first and then all of them are terminated.

Eg:
```
0 MAIN
    1 created
    2 created
    3 created
    .
    .
    .
    .
    10 created
    1 ended
    2 ended
    3 ended
    .
    .
    .
    .
    10 ended
MAIN ended
```

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

```c
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
void *thd_ret[10];
int ret;
int ct=1;

void *thdfunc(void *arg){
    char c[4];
    pthread_t *thd_id = (pthread_t *)arg;
    pthread_mutex_lock(&mutex1);//enter CS
        pthread_t id = pthread_self();
        printf("HELLO from Thread %d with ID: %lu | Received: %lu \n",
ct, id, *thd_id);
        ret=ct;
        ct++;
    pthread_mutex_unlock(&mutex1);//exit CS
    pthread_exit((void*)ret);
}

int main(){
    int i;
    pthread_t sid, tid[10];
    sid = pthread_self();

    printf("HELLO from Parent with ID = %lu\n", sid);

    for (i=0; i<10; i++){
        if (pthread_create(&(tid[i]), NULL, &thdfunc, (void
*)&tid[i])){
            perror("pthread_create() error\n");
            exit(1);
        }
        sleep(1);
```

```
    }

    for (i = 0; i < 10; i++){
        pthread_join(tid[i], &thd_ret[i]);
        printf("Thread %d with ID %lu has ENDED \n", thd_ret[i],
tid[i]);
    }

    printf("Parent with ID: %lu exitting..\n", sid);
    return 0;
}
```

ii. Synchronize it such that one is created and ends before the next one is created (till all 10 are created and terminated)

Eg:
```
0 MAIN
    1 created
    1 ended
    2 created
    2 ended
    3 created
    3 ended
    .
    .
    .
    .
    10 created
    10 ended
MAIN ended
```

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
void *thd_ret[10];
int ret;
int ct=1;
```

```c
void *thdfunc(void *arg){
    char c[4];
    pthread_t *thd_id = (pthread_t *)arg;
    pthread_mutex_lock(&mutex1);//enter CS
        pthread_t id = pthread_self();
        printf("HELLO from Thread %d with ID: %lu | Received: %lu \n",
ct, id, *thd_id);
        ret=ct;
        ct++;
    pthread_mutex_unlock(&mutex1);//exit CS
    pthread_exit((void*)ret);
}

int main(){
    int i;
    pthread_t sid, tid[10];
    sid = pthread_self();

    printf("HELLO from Parent with ID = %lu\n", sid);

    for (i=0; i<10; i++){
        if (pthread_create(&(tid[i]), NULL, &thdfunc, (void
*)&tid[i])){
            perror("pthread_create() error\n");
            exit(1);
        }

        else{
            pthread_join(tid[i], &thd_ret[i]);
            printf("Thread %d with ID %lu has ENDED \n", thd_ret[i],
tid[i]);
        }
    }

    printf("Parent with ID: %lu exitting..\n", sid);
    return 0;
}
```

iii. Synchronize it such tha
how they were created.

Eg:
```
0 MAIN
        1 created
        2 created
        2 ended
        1 ended
        3 created
        4 created
        4 ended
        3 ended
        .
        .
        .
        .
        9 created
        10 created
        10 ended
        9 ended
MAIN ended
```

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
void *thd_ret[10];
int ret;
int ct=1;

void *thdfunc(void *arg){
    char c[4];
    pthread_t *thd_id = (pthread_t *)arg;
    pthread_mutex_lock(&mutex1);//enter CS
        pthread_t id = pthread_self();
        printf("HELLO from Thread %d with ID: %lu | Received: %lu \n",
ct, id, *thd_id);
        ret=ct;
        ct++;
    pthread_mutex_unlock(&mutex1);//exit CS
```

```
    pthread_exit((void*)ret);
}

int main(){
    int i;
    pthread_t sid, tid[10];
    sid = pthread_self();

    printf("HELLO from Parent with ID = %lu\n", sid);

    for (i=0; i<10; i++){
        if (pthread_create(&(tid[i]), NULL, &thdfunc, (void
*)&tid[i])){
            perror("pthread_create() error\n");
            exit(1);
        }

        else{
            if (i%2){
                pthread_join(tid[i], &thd_ret[i]);
                printf("Thread %d with ID %lu has ENDED \n",
thd_ret[i], tid[i]);
                pthread_join(tid[i-1], &thd_ret[i-1]);
                printf("Thread %d with ID %lu has ENDED \n", thd_ret[i-
1], tid[i-1]);
            }
        }
    }

    printf("Parent with ID: %lu exitting..\n", sid);
    return 0;
}
```