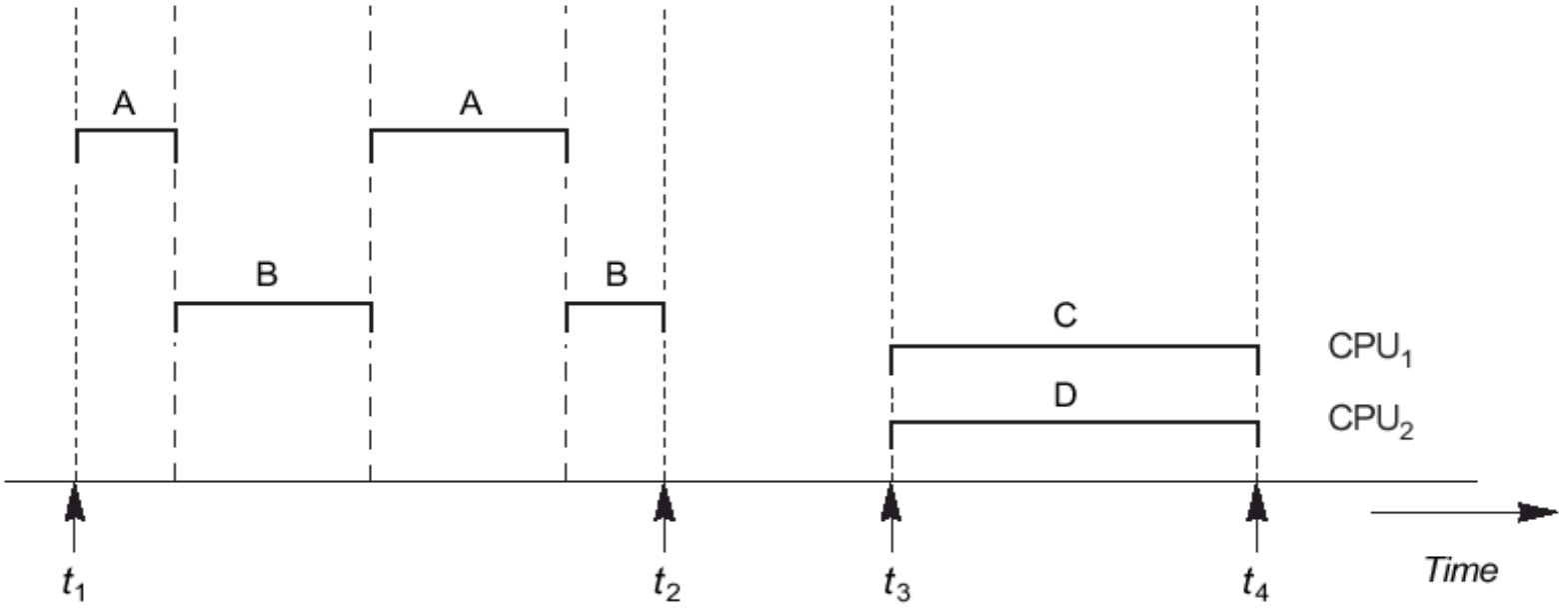
Inter-process synchronisation

Concurrent /Parallel Processes

concurrency processing Parallel processing

Figure 19.1 Interleaved processing versus parallel processing of concurrent transactions.



Why Concurrency Control?

- However, concurrent transaction need to be carefully synchronised to:
  - to ensure the non interference or isolation property of concurrently executing transactions/processes.
  - Which can result in the so called RACE (Lost update) problem
  - This is achieved by protocols that employ **locking**, in which transactions/ process get exclusive access to a data item (critical region),

What is Parallel Processing?

- Parallel processing
  - **Two or more processors** operate in one system at the same time
    - Work may or may not be related
  - Two or more CPUs execute instructions simultaneously (in parallel…)
  - Processor Manager
    - Coordinates activity of each processor
    - Synchronizes interaction among CPUs

Process Synchronization Software (cont'd.)

- Critical region
  - Part of a program that must be executed in “one action”. (All or Nothing)
  - Other processes must wait before accessing critical region resources

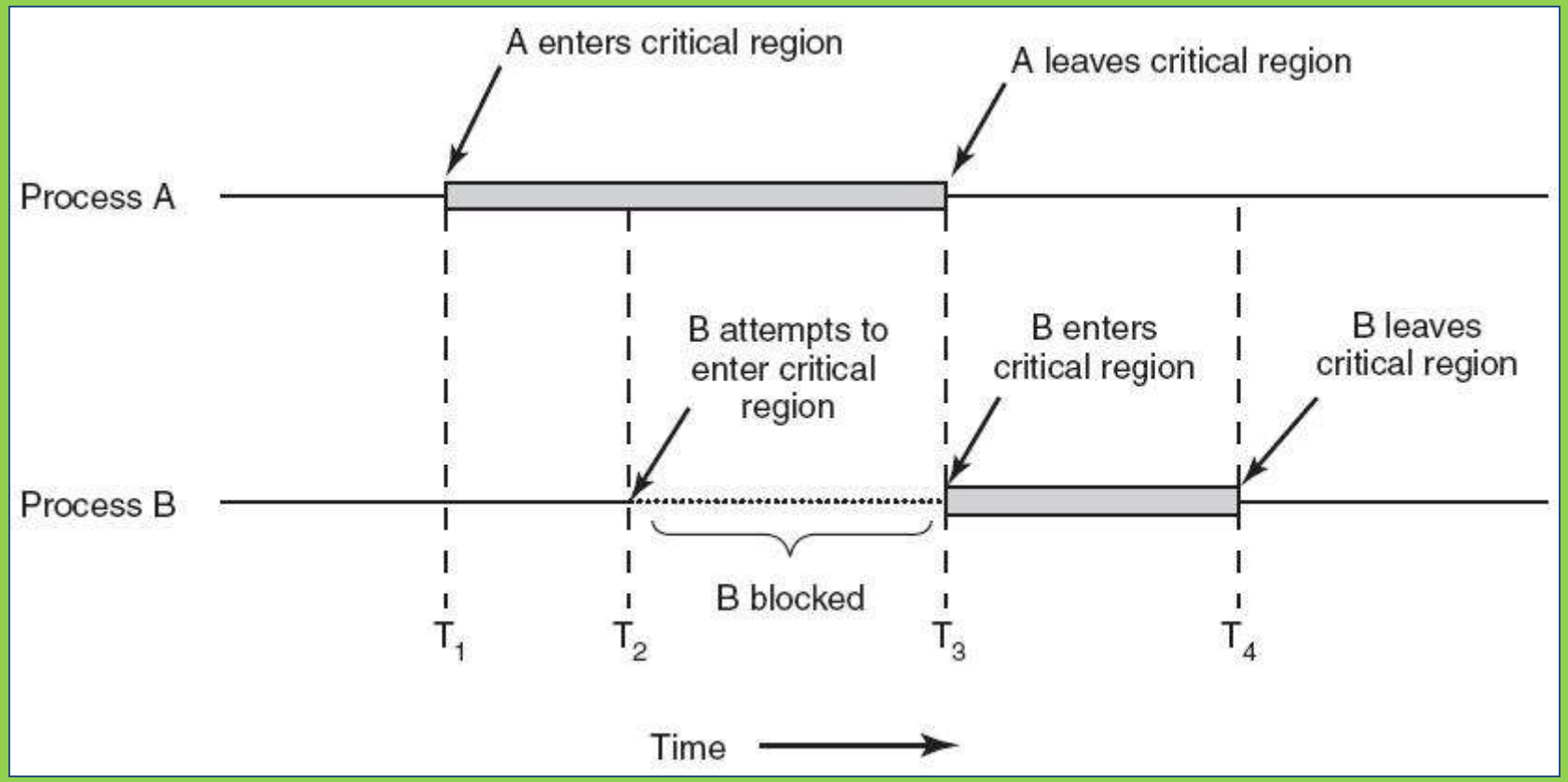
- Processes within the critical region
  - Cannot be interleaved otherwise
    - Threatens integrity of operation (all or nothing action)

Non Synchronisation example

- if two processes or threads/transactions try to update the same object can result in: **Race/lost update problem : The race problem also applies to databases but it is not the same as thread race problem.**

- Consider v = v + 1:
  1. Thread A fetches *v* = 5 into a processor register within processor A
  2. Thread B fetches *v* = 5 into a processor register within processor B
  3. Thread B increments value in its processor register to 6
  4. Thread B stores 6 into *v*
  5. Thread A increments value in its processor register to 6
  6. Thread A stores 6 into *v*

Critical regions of concurrent processes



Critical region (race problem)

- In the previous example:

- Step 1 occurs (fetch by process A)
- Step 2 -4 must be blocked (fetch… update by B)
- Step 5 and step 6 proceed. (update by process A)
- Then steps 2-4 can proceed.

- This *ensures* that the *integrity of critical region* (e.g. updating a register) *is maintained*.

Ensuring Process Synchronization

- Synchronization
  - Implemented as lock-and-key arrangement:
  - Process determines key availability
    - Process obtains key
    - Uses it to lock access to critical region
    - Makes it unavailable to other processes
- Types of locking mechanisms
  1. Test-and-set
  2. WAIT and SIGNAL
  3. Semaphores

1 Test-and-Set

- T.S Executed in single machine cycle
  - Test If key available: set to unavailable
- Actual **key** often referred as the **Mutex**
  - Single bit in storage location: zero (free) or one (busy)
- Before process enters critical region
  - Tests condition code using TS instruction
  - If No other process in region (key is zero)
    - Process proceeds
    - Condition code (key) changed from zero to one
    - P1 exits: code (key) reset to zero, allowing others to enter

Test and Set

- Use a lock variable (*mutex*):

```
while (test_and_set(mutex) == 1) {
    // do nothing
}
critical_section();
mutex = 0;
```

- Requires an *atomic* test-and-set operation
  - If mutex value is 0 resets mutex to value 1
  - Does not enter the loop; goes to critical section()
  - Otherwise stays witihin the while loop resulting in what is referred to as *busy waiting*

Test-and-Set (cont'd.)

- Advantages
  - Simple procedure to implement
  - Works well for small number of processes
- Drawbacks
  - Busy waiting
    - Waiting processes remain in unproductive, resource-consuming wait loops
  - Starvation (*will cover in more detail in next lecture*)
    - Many processes waiting to enter a critical region
    - Processes gain access in arbitrary fashion

Creating Mutex in linux

- Data Type: `pthread_mutex_t`

- **Mutex functions:**

  - `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`

  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

- Important: Mutex scope must be visible to all threads!

- **Mutex1.c** is a simple example

Example 1 Mutex 1.c: a race problem example

denis.manley@soc-apollo-dk: ~/OS2/week8

```
*/

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main()
{
   int rc1, rc2;
   long t1 =1,t2=2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC,(void*)t1)))
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC,(void *)t2 )))
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC(void * threadID)
{
   long thread_num;
   thread_num = (long)threadID;
   printf("thread %ld is accessing before mutex\n",thread_num);
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
//  pthread_mutex_unlock( &mutex1 );
   printf("thread %ld is finished function\n",thread_num);
}
                                                    63,1          Bot
```

```
denis.manley@soc-apollo:~/OS2/week8$ ./mutex1
thread 1 is accessing before mutex
Counter value: 1
thread 1 is finished function
thread 2 is accessing before mutex
```

Uncomment mutex unlock

```
denis.manley@soc-apollo:~/OS2/week8$ ./mutex1
thread 1 is accessing before mutex
Counter value: 1
thread 1 is finished function
thread 2 is accessing before mutex
Counter value: 2
thread 2 is finished function
```

Test and Set example using Mutexs

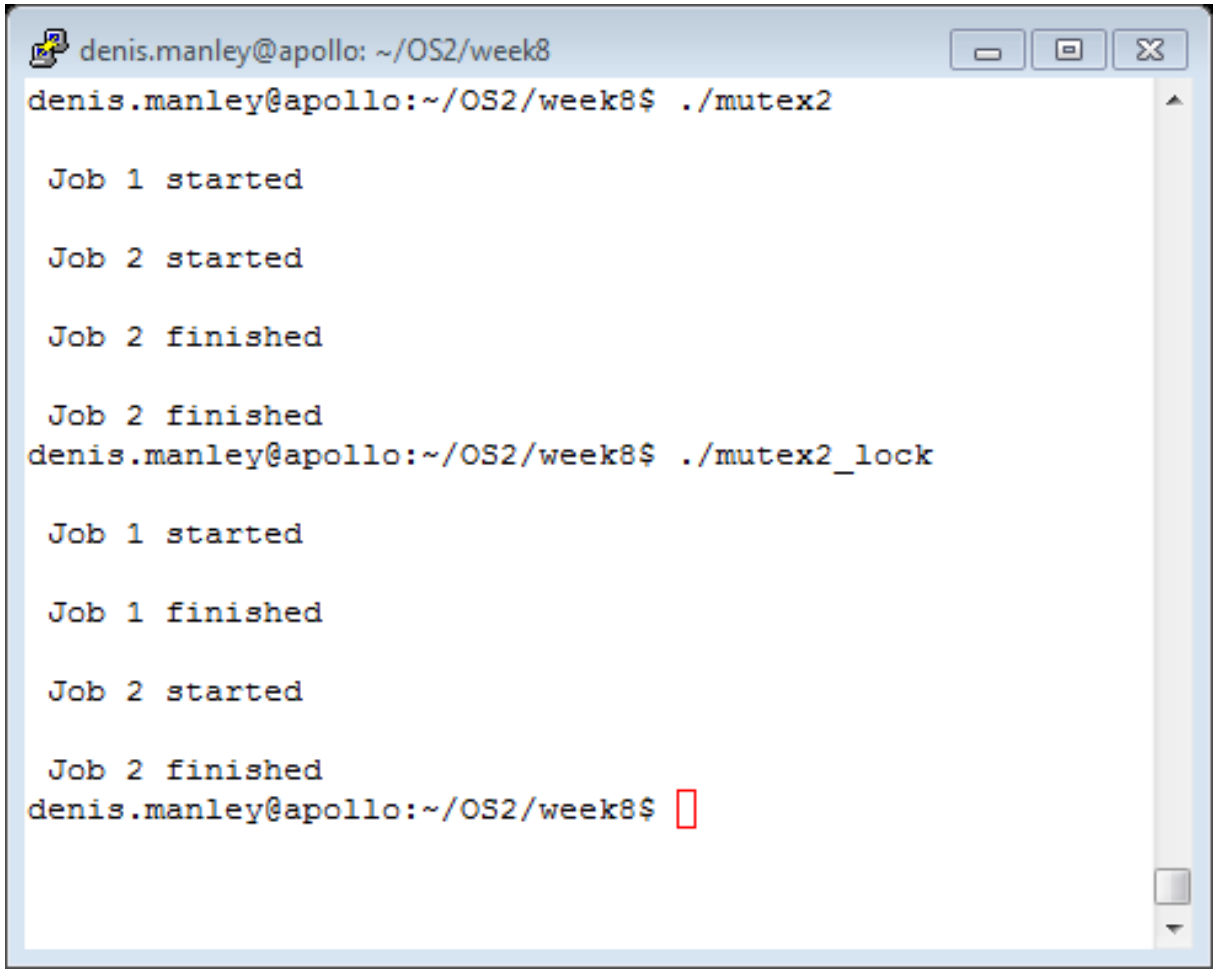
Mutex2.c

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

pthread_t tid[2];
int counter;

void* doSomeThing(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int err;

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

Mutex2_lock.c

```c
void* doSomeThing(void *arg)
{
    pthread_mutex_lock(&lock);       //lock the mutex other thread

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);  // unlock the mutex now other th

    return NULL;
}

int main(void)
{
    int i = 0;
    int err;


    if (pthread_mutex_init(&lock, NULL) != 0)      // declare the l
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);    //remove the lock variable (l

    return 0;
}
```

Sample output

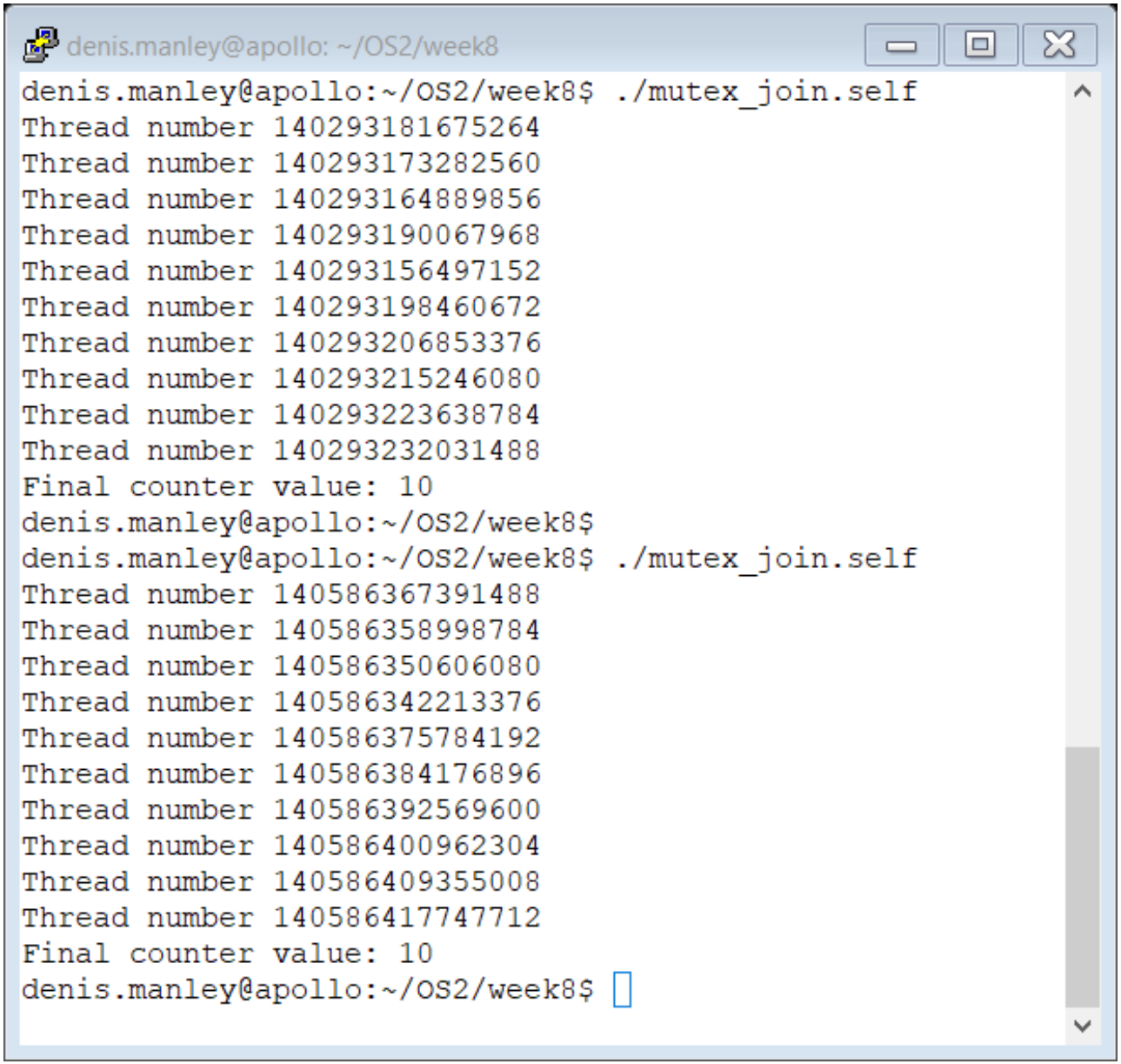


Explain the different outputs

Example 2: *mutex_join_self.c*

```
denis.manley@apollo:~/OS2/week8$ vi mutex_join.self.c
    #define NTHREADS 10
    void *thread_function(void *);
    pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;  // default declaration
    int  counter = 0;

    main()
    {
       pthread_t thread_id[NTHREADS];
       int i, j;

       for(i=0; i < NTHREADS; i++)
       {
          pthread_create( &thread_id[i], NULL, thread_function, NULL );
       }

       for(j=0; j < NTHREADS; j++)
       {
          pthread_join( thread_id[j], NULL);
       }

       /* Now that all threads are complete I can print the final result.     */
       /* Without the join I could be printing a value before all the threads */
       /* have been completed.                                                */

       printf("Final counter value: %d\n", counter);
    }

    void *thread_function(void *dummyPtr)
    {
       printf("Thread number %ld\n", pthread_self());
       pthread_mutex_lock( &mutex1 );
       counter++;
       pthread_mutex_unlock( &mutex1 );
    }
```

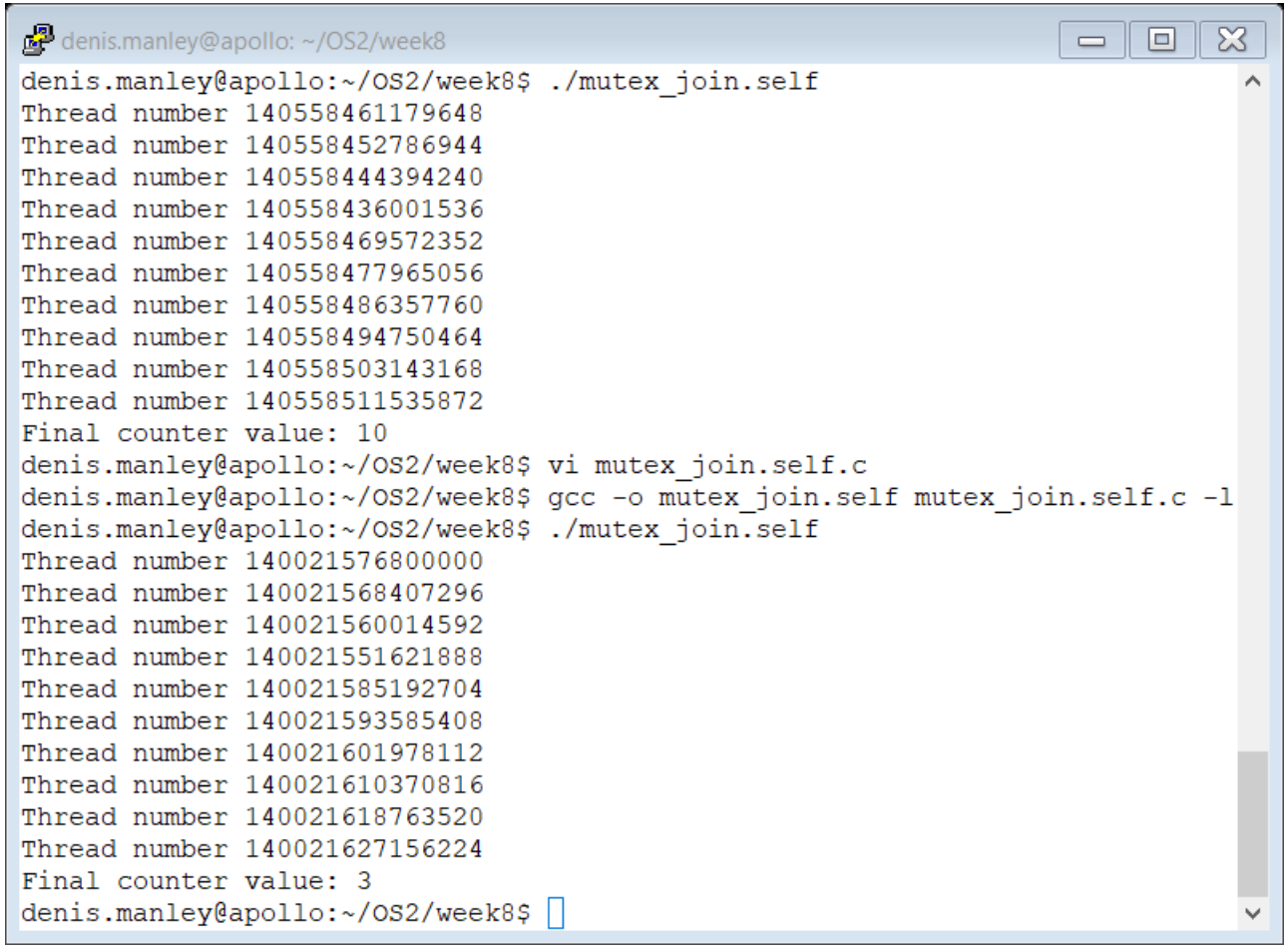
Question: Explain Sample output ?



Modify the “self” code

- Count++; is equivalent to counter = counter +1;
- Alternatively in 3 steps:
  - temp = counter;
  - Temp = temp +1;
  - Counter = temp;
- Insert some processing between the steps: for (i=0…)
  - What do you think would happen if:
  - We kept lock/unclock
  - We removed lock/unlock
  - We remove unlock

Sample output with no mutex lock



2 WAIT and SIGNAL

- **A better synchronisation method: *block process/thread until explicitly unblocked (wait and set)***
  - This prevents a process/thread from running again until another thread signals that the situation (access to the critical regions) has changed

- ***Two* new mutually exclusive operations:**
- **WAIT**
  - Activated when process/thread encounters "*busy" condition* code and puts thread on the *waiting queue* and changes the status of its PCB/TCB (e.g. wait while counter is < 6)
- **SIGNAL**
  - Activated if a process/thread encounters a condition "code" and sets mutex to "free"; signals processes/threads in *waiting queue* to run on the CPU. (signal if counter = 6)

“wait and set” Condition variables

- Data Type `pthread_cond_t`

- **Create Functions:**
  - `int` **`pthread_cond_init`**`(pthread_cond_t *cond,`
  - `const pthread_condattr_t *attr);`
  - `int` **`pthread_cond_destroy`**`(pthread_cond_t *cond);`
- **Waiting on condition**
  - `int` **`pthread_cond_wait`**`(pthread_cond_t *cond,`
  - `pthread_mutex_t *mutex);`
- **Waking (set)**
  - `int` **`pthread_cond_singal`**`(pthread_cond_t *cond);`
  - `int` **`pthread_cond_broadcast`**`(pthread_cond_t *cond); (waking multiple threads)`

int main .c (Wait_Signal.c)

```
int main(int argc, char *argv[])
{
  int i, rc;
  long t1=7, t2=2, t3=10;
  pthread_t threads[3];
  pthread_attr_t attr;

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);


  pthread_create(&threads[0], NULL, wait_fnt, (void *)t1);
  pthread_create(&threads[1], NULL, signal_fnt, (void *)t2);
  pthread_create(&threads[2], NULL, signal_fnt, (void *)t3);

  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Wait and signal theads finished Final value of count = %d!!!\n",
          count);

  /* Clean up and exit */

  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);

  pthread_exit (NULL);

}
```

Signal thread code

denis.manley@soc-apollo-dk: ~/OS2/week9

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

#define NUM_THREADS  3
#define TCOUNT 6
#define COUNT_LIMIT 10

int  count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *signal_fnt(void *t)
{
  int i;
  long my_id = (long)t;
  printf("Signal thread id %ld is starting...", my_id);
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condi
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal to waiting thread.\n");
      }
    printf("thread %ld, count = %d, unlocking mutex\n",
           my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}
```

Wait Thread Code

```
void *wait_fnt(void *t)
{
  long my_id = (long)t;

  printf("Starting wait thread: thread ID is:  %ld\n", my_id);

  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
        printf("In wait thread %ld Count= %d. Going into wait state...\n", my_id,count);

        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("thread %ld Condition signal received. Count= %d\n", my_id,count);
        printf("thread %ld Updating the value of count %d...\n", my_id,count);

    count += 200;

        printf("thread %ld count now = %d .\n", my_id, count);
    }

  printf("thread %ld Unlocking mutex.\n", my_id);

  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Explain output ***wait_signal.c*.**
Remove **wait and signal functions** and explain output

denis.manley@soc-apollo-dk: ~/OS2/week9

```
denis.manley@soc-apollo-dk:~/OS2/week9$ vi wait_signal_2025.c
denis.manley@soc-apollo-dk:~/OS2/week9$ ./wait_signal_2025
Starting wait thread: thread ID is:  7
In wait thread 7 Count= 0. Going into wait state...
Signal thread id 10 is starting...thread 10, count = 1, unlocking mutex
Signal thread id 2 is starting...thread 2, count = 2, unlocking mutex
thread 10, count = 3, unlocking mutex
thread 2, count = 4, unlocking mutex
thread 10, count = 5, unlocking mutex
thread 2, count = 6, unlocking mutex
thread 10, count = 7, unlocking mutex
thread 2, count = 8, unlocking mutex
thread 10, count = 9, unlocking mutex
thread 2, count = 10  Threshold reached. Just sent signal to waiting thread.
thread 2, count = 10, unlocking mutex
thread 7 Condition signal received. Count= 10
thread 7 Updating the value of count 10...
thread 7 count now = 210 .
thread 7 Unlocking mutex.
thread 10, count = 211, unlocking mutex
thread 2, count = 212, unlocking mutex
Wait and signal theads finished Final value of count = 212!!!
denis.manley@soc-apollo-dk:~/OS2/week9$
```

Semaphores

- The semaphore methods is a generalisation of the mutex (*wait and signa*l)
  - has an integer value ≥ 0 (wait/signal it is 0 or 1)
  -
  - **wait operation**: suspend if = 0, otherwise decrement by 1 and continue.
  - **signal operation**: increment (and notify any threads blocked on this semaphore)
  - Signals if/when a **resource/critical region is free**
    - Resource can be used by another process

3 Semaphores

- Original terminology
- Two operations of semaphore: proposed by Dijkstra (1965)
  - P (proberen means “to test”)
  - V (verhogen means “to increment”)
  - *Where **P** is equivalent to **Wait** operation (waiting for semaphore t**o give the all clear** to enter critical region)*
  - ***V** is like **Signal** indicating its free and selects next process **to enter critical region***

Semaphores (cont'd.)

- Let *s* be a semaphore (mutual exclusive) variable
  - P(*s*): **If *s* > 0, then *s*: = *s* – 1**
    - Test, fetch, decrement, store sequence (test and set/wait signal)
  - **V(*s*): *s*: = *s* + 1**
    - Fetch, increment, store sequence (enter signal)
- *If s* = 0 implies *busy a process in* critical region
  - A Process/thread calling on P operation must wait until *s* > 0
- One leaving the critical region the V function is called
  - The V signals to “one” of the process in the waiting queue that the critical region is free and increment semaphore by 1

Semaphores in posix

- **Initalise semaphore:**
- **sem_int (sem_t *sem, 0, unsigned int value)**
  - sem_int(&sem_name, 0, 1)
  - sem)_int(&sem_name, 0, 0)
  - The 3rd parameter can be a number > 0:
  - Note 2nd number relates to shared mutexes and in Linux must be set to 0

- **P (wait) function is Sem_wait(sem_t *sem) ;**
  - sem_wait(&sem_name)
  - If functions returns a negative value it blocks the process and must wait for the sem_post (V or signal to wake up)

Semaphores in posix

- **V : signal function or sem_post (sem_t *sem)** ;
  - sem_post(&sem_name)
  - Increments the value of the semaphore and wakes up any process (thread) that is waiting

- Other functions associated with semaphores are:
  - int sem_**getvalue**(&sem_name, &value);
    - printf(“the value of the semaphore is %d \n”, value);
  - sem_**destroy** (&sem_name);

A program using semaphores

- To Run a semaphore program use:
- gcc -o example example.c **-lpthread –lrt**

- Link to semaphore basics

Semaphore_example.c

denis.manley@soc-apollo-dk: ~/OS2/week9

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem1, sem2;

void *thread1( void *arg ) {

    int value;
    sem_getvalue(&sem1, &value);
    printf("Start Thread1: The value of semaphore 1 is %d\n", value);
    sem_getvalue(&sem2,&value);
    printf("Start Thread 1: The value of semaphore 2 is %d\n",value);

    sem_wait( &sem1 );
    printf( "I'm in thread 1,signal sem2\n" );
    sem_post( &sem2 );

    sem_getvalue(&sem1, &value);
    printf("End Thread 1: The value of semaphore 1 is now %d\n", value);
    sem_getvalue(&sem2, &value);
    printf("End Thread 1: The value of semaphore 2 is now %d\n", value);



    pthread_exit( NULL );
    return NULL;
}

 void *thread2( void *arg ) {
    int value;
    sem_getvalue(&sem1, &value);
    printf("Start Thread 2: The value of semaphore 1 is %d\n", value);
    sem_getvalue(&sem2,&value);
    printf("Start Thread 2: The value of semaphore 2 is %d\n",value);

    sem_wait( &sem2 );
    printf( "I'm in thread 2, awoken by thread 1\n" );

    sem_getvalue(&sem1, &value);
    printf("End Thread 1: The value of semaphore 1 is now %d\n", value);
    sem_getvalue(&sem2, &value);
    printf("End Thread 1: The value of semaphore 2 is now %d\n", value);

    pthread_exit( NULL );
    return NULL;
}
```

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

    /* initalise the semaphores (mutex) the first is free
    the second (set to 0) is busy: ensure thread 1 is executed before thread 2*/
    sem_init( &sem1, 0, 1 );
    sem_init( &sem2, 0, 0 );

    pthread_t threads[ 2 ];

    pthread_create( &threads[ 0 ], NULL, thread2, NULL );
    sleep( 1 );
    pthread_create( &threads[ 1 ], NULL, thread1, NULL );


   /* prevents process from closing before threads are executed */
    pthread_join( threads[ 0 ], NULL );
    pthread_join( threads[ 1 ], NULL );

   /* remove semapores */
    sem_destroy( &sem2 );
    sem_destroy( &sem1 );

    return 0;
}
```

Semaphore_example .c: note semaphore values

```
denis.manley@soc-apollo-dk:~/OS2/week9$ ./Semaphore_Example
Start Thread 2: The value of semaphore 1 is 1
Start Thread 2: The value of semaphore 2 is 0
Start Thread1: The value of semaphore 1 is 1
Start Thread 1: The value of semaphore 2 is 0
I'm in thread 1,signal sem2
End Thread 1: The value of semaphore 1 is now 0
I'm in thread 2, awoken by thread 1
End Thread 1: The value of semaphore 1 is now 0
End Thread 1: The value of semaphore 2 is now 0
End Thread 1: The value of semaphore 2 is now 0
denis.manley@soc-apollo-dk:~/OS2/week9$
```

- Comment out
- // sem_post(&sem2)

- Program stops with following output:

```
denis.manley@soc-apollo-dk:~/OS2/week9$ ./Semaphore_Example
Start Thread 2: The value of semaphore 1 is 1
Start Thread 2: The value of semaphore 2 is 0
Start Thread1: The value of semaphore 1 is 1
Start Thread 1: The value of semaphore 2 is 0
I'm in thread 1,signal sem2
End Thread 1: The value of semaphore 1 is now 0
End Thread 1: The value of semaphore 2 is now 0
```

Semaphore examples with/without mutex:

- A program using two threads to increment a global variable 10000.
- The **badcnt.c** does not use semaphores and the result is not always 10000 (This is due to the race problem)

- The **goodcnt.c** uses semaphores to lock the variable while being update by either thread and thus prevents the race problem. In this case the result is always 10000
- Run these two programs in the lab and ensure you understand what is happening

Prevent race problem with semaphores

Badcnt.c

```
#define NITER 1000000

int cnt = 0;

void * Count(void * a)
{
    int i, tmp;

    int t = (int)a;
    for(i = 0; i < NITER; i++)
    {
        tmp = cnt;      /* copy the global cnt locally */
        tmp = tmp+1;    /* increment the local copy */
        cnt = tmp;      /* store the local value into the global cnt */
        printf("incremented by thread %d\n",t);
    }
}
```

Goodcnt**.c**

```
void * Count(void * a)
{
    int i, tmp;

    int t = (int)a;

    /* lock the mutex */

    printf("thread %d about to block the mutex\n", t);

    sem_wait(&mutex);

    printf(" the mutex is now locked by thread %d \n", t);
    for(i = 0; i < NITER; i++)
    {


        tmp = cnt;      /* copy the global cnt locally */
        tmp = tmp+1;    /* increment the local copy */
        cnt = tmp;      /* store the local value into the global cnt */

      printf( "the thread is number %d\n", t);

    }

        printf("unlocking the mutex\n");
        sem_post(&mutex);

        printf(" the mutex is unlocked by thread %d\n",t);


        /* code to exit a thread correctly */
        pthread_exit( NULL );

        return NULL;
}
```

P (Wait) command

V (signal) command

Sample output bad and good

```
denis.manley@apollo:~/OS2/week9$ ./badcnt

 OK! cnt is [2000000]
denis.manley@apollo:~/OS2/week9$ ./badcnt

 BOOM! cnt is [1480657], should be 2000000
denis.manley@apollo:~/OS2/week9$ ./badcnt

 OK! cnt is [2000000]
denis.manley@apollo:~/OS2/week9$ ./badcnt

 OK! cnt is [2000000]
denis.manley@apollo:~/OS2/week9$ ./badcnt

 BOOM! cnt is [1471883], should be 2000000
```

```
denis.manley@apollo:~/OS2/week9$ ./goodcnt
thread 2 about to block the mutex
 the mutex is now locked by thread 2
thread 1 about to block the mutex
unlocking the mutex
 the mutex is now locked by thread 1
unlocking the mutex
 the mutex is unlocked by thread 2
 the mutex is unlocked by thread 1

 OK! cnt is [2000000]
denis.manley@apollo:~/OS2/week9$ ./goodcnt
thread 2 about to block the mutex
 the mutex is now locked by thread 2
thread 1 about to block the mutex
unlocking the mutex
 the mutex is now locked by thread 1
 the mutex is unlocked by thread 2
unlocking the mutex
 the mutex is unlocked by thread 1

 OK! cnt is [2000000]
denis.manley@apollo:~/OS2/week9$ ./goodcnt
thread 2 about to block the mutex
 the mutex is now locked by thread 2
thread 1 about to block the mutex
unlocking the mutex
 the mutex is now locked by thread 1
unlocking the mutex
 the mutex is unlocked by thread 2
 the mutex is unlocked by thread 1

 OK! cnt is [2000000]
```

Process Cooperation

- Several processes work together to complete common task
- Each case requires
  - *Mutual exclusion and synchronization*

- Classical Example
  1. Producers and consumers problem

- Is implemented using semaphores
- Used to “even out” speed differences between producers (e.g. PC) and consumers (e.g. printers)

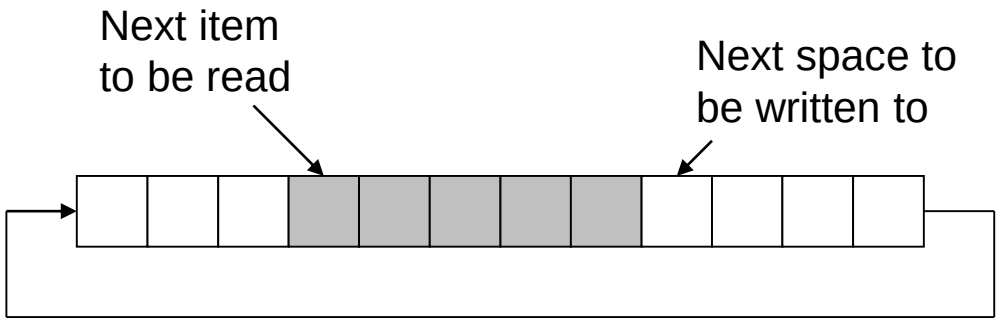
Producers and Consumers

- One process **produces data**
- Another process “later” **consumes data**
- Requires a **bounded buffer** to control differences in “speed” between both processes.

- Example: CPU (producer) and printer buffer (consumer)
  - Delay producer if the “printer” buffer is full
  - Delay consumer: if buffer empty
  - Mutual exclusion to bounded buffer (can only add or remove from buffer at any one time)

Bounded buffers

- Producer threads produce values, consumer threads consume them
- Bounded buffer can hold up to N items
  - used to even out speed differences between producers and consumers



Producers and Consumers (cont'd.)

- Initialise Variables

  `empty: = n; full: = 0; mutex: = 1`

- Producers Algorithm:

- **{P x > 0; x = x - 1); (V x = x + 1}**

Producer algorithm	Consumer Algorithm
Produce data	P(full)
P(empty)	P(mutex)
P(mutex)	# BEGIN CRITICAL REGION
#begin critical region	*read item from buffer*
write item into buffer	V(mutex)
V(mutex)	#END CRITICAL REGION
#END CRITICAL REGION	V(empty)
V(full)	Consume data

Example of producer and consumer

- Using a single “bounder” buffer;
- number of elements N = 10. mutex = 1;

- *If empty =10 ; full = 0 (empty buffer)*
  - What happens in producer algorithm?
  - What happens to consumer algorithm?
- *If empty = 0; full = 10 (full buffer)*
  - What happens in producer algorithm?
  - What happens to consumer algorithm?
-

A Bounded buffer code with **bug** which can cause deadlock

Empty Buffer: **itemsAvailable = 0, spaceAvailable = N**

```
insert(Item i) {
    p(spaceAvailable);
    p(mutex);
    doInsert(i);
    v(mutex);
    v(itemsAvailable);
}
```

```
Item remove() {
 p(mutex);
p(itemsAvailable);
Item i = doRemove();
v(mutex);
 v(spaceAvailable);
return i;                }
```

Explain why will this code cause a problem or end up in deadlock.

Sample Question

- Explain why it is critical to ensure that concurrency is carefully controlled for processes accessing the same data item; in other words the race problem **(6marks)**
-
- The test and set is an algorithm to prevent the race problem. What is the main problem with this algorithm and explain how the wait and signal approach overcomes this problem **(6 marks)**

Sample question

- Threads use two functions to implement correct inter-process communication. What are the two functions and illustrate, with a suitable example how they work to prevent interference between threads.
- Or

- Explain in detail how the wait and signal example works (note you will be given the code)**(12 marks)**

- Give an example of the expected output explaining your answer **(6 marks)**

Sample question

Producer algorithm	Consumer Algorithm
Produce data	P(full)
P(mutex)	P(mutex)
P(empty)	# BEGIN CRITICAL REGION
#begin critical region	*read item from buffer*
write item into buffer	V(mutex)
V(mutex)	#END CRITICAL REGION
#END CRITICAL REGION	V(empty)
V(full)	Consume data

Given the above pseudocode for the producer/consumer problem. Explain does this code prevent/result in deadlock **(8 marks)**