**THREADS**

* 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
* R.A.C.E Problem: The race problem occurs when two processes are accessing the same variable.
* The Loss/Update problem: occurs one of the update will be incorrect
* To fix this we use a lock & key
* Types of locking mechanisms
  + 1.Test-and-set
  + 2.WAIT and SIGNAL
  + 3.Semaphores
* **key** often referred as the **Mutex**
* if the mutex = 0 then its unlocked
* if the mutex is 1 then its locked

**TEST AND SET**

* the process repeatedly checks if the mutex is unlocked (equal to 0).
* Use a lock variable (*mutex*):

while (test\_and\_set(mutex) == 1) {

// do nothing

}

critical\_section();

mutex = 0;

* When the critical region of one program is finished the mutex is set back to 0 allowing the next program to run.
* A new program then enters the cpu and locks the mutex (sets it to 1)again when this process is finished it unlocks the mutex(sets it to 0 ) allowing the next program to run.
* Basically the mutex is just a bit getting changed from 1 to 0
* One bad thing about test and set is it can lead to starvation.
* **Creating a mutex**
* Data Type: pthread\_mutex\_t

Mutex functions:

* + int pthread\_mutex\_init(pthread\_mutex\_t \*mutex,

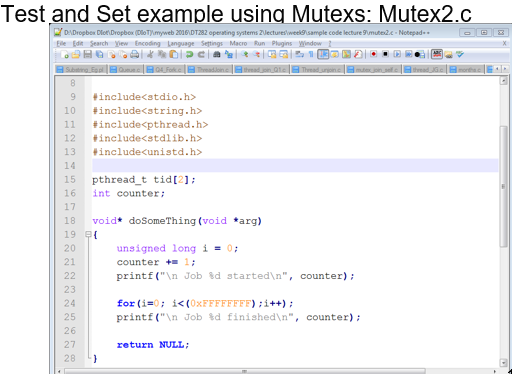
const pthread\_mutexattr\_t \*attr);

* + int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);
  + int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);
* **To compile a thread program**

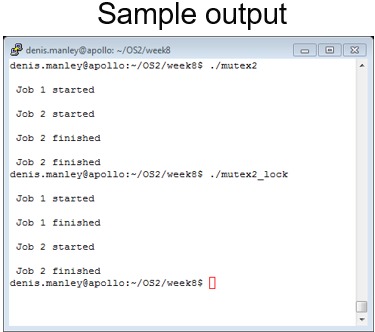
gcc program.c -o program -lpthread

* so why is it important to lock the mutex (look at the following example)

-mutex is never locked in this example so both threads are using the same counter variable leading to the loss/update problem







**Wait and Signal**

* better synchronization method then the test and set.block process/thread until explicitly unblocked (wait and set)
* It prevents a process from running until another thread signals the situation has changed(ie mutex has been unlocked)
* NB

🡪 1. we use the WAIT () fuction

-Activated when process/thread encounters “*busy” (ie mutex is 1)condition* code and puts thread on the *waiting queue* and changes the status of its PCB/TCB

🡪 2. We use the SIGNAL () function

-Activated when mutex has been set the open/unlocked (ie when mutex is changed from 1 to 0).This signals processes/threads in *waiting queue* to run on the CPU.

**“wait and set” Condition variables**

* Data Type pthread\_cond\_t
* Create/destroy 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);

* Signal (Waking)
  + int pthread\_cond\_singal(pthread\_cond\_t \*cond);

* OPEN CODE FOR WAIT\_SIGNAL.C

-T1 calls wait function and T2/T3 call signal function

-T1 must wait until counter = 12

-T2/T3 will each increment the counter 10 times

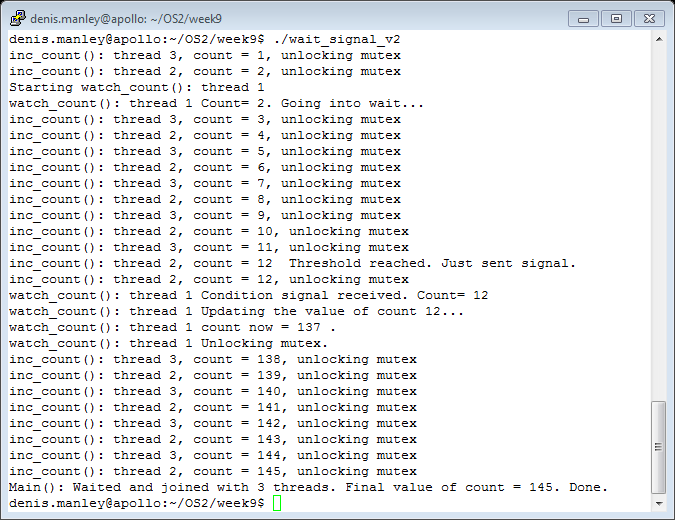
-The for loop in the signal function

-T2/T3 increment counter in an order determine by the process scheduler. (locking, incrementing, and unlocking)

-When counter = 12; T1 is woken up locks mutex and can continue its processing counter+=125 then unlocks mutex

-Then T2/T3 finish of their processing: ensuring that each one increments the counter 10 times

-Therefore, the final value will be: 125+10+10 = 145



**Semaphores**

* Is a generalization of wait and Signal .
* Similar to the wait and signal but the mutex can have any value
* There are 2 operations of semaphores

-P & V

* + *P : is equivalent to Wait operation (waiting for semaphore to give the all clear to enter critical region)*
  + *V : is like Signal indicating its free and selects next process to enter critical region*
  + NB ………..

-int sem\_wait(sem\_t \*s) { //starting critical section

Decrement the value of the semaphore s by 1 and

Wait if value of semaphore s is negative

}

Carry out processing in middle

-int sem\_post(sem\_t \*s) {//signal process is done

Increment the value of semaphore s by 1 and

If there are one or more threads waiting ,wake one

}

//wait decremnts semaphore not allow thread on cpu

//post increments semaphore indicating a new thread in allowed on the cpu

FOR EXAMPLE

* Let S be a semaphore
  + P(*s*): **If *s* > 0, then *s*: = *s* – 1**
  + **V(*s*): *s*: = *s* + 1**
  + *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 ,therefor allowing the new process onto the cpu
* So basically if s <= 0 cpu is busy.

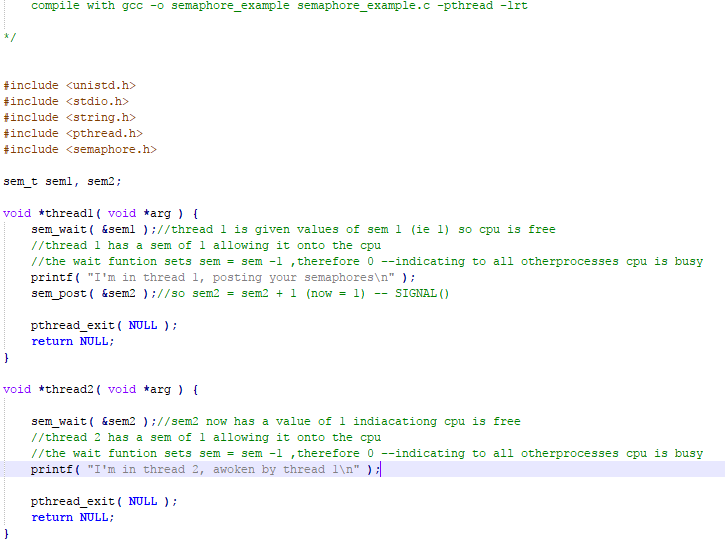
If s > 0 cpu is free.

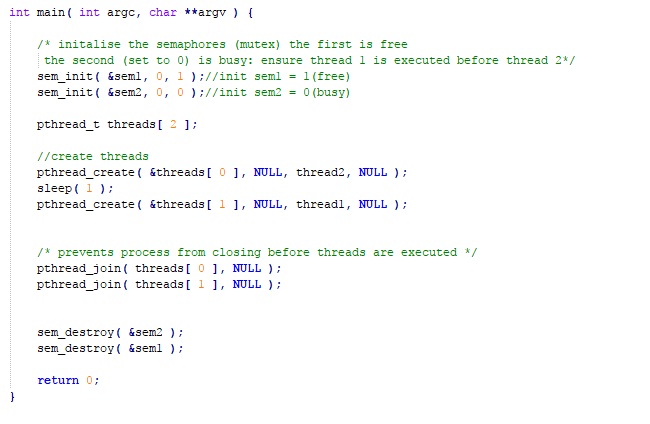
* So basically if sem is 0 or less its not allowed on cpu .

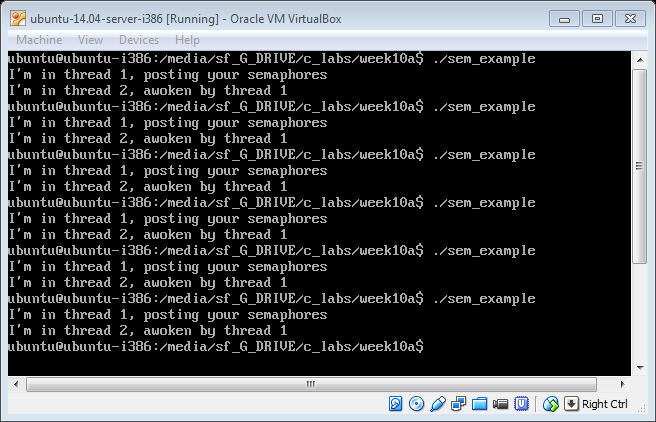
If sem is 1 its allowed on cpu.

* **To compile semaphoes**

**- gcc -o semaphore\_example semaphore\_example.c -pthread -lrt**







**MY EXPLANATION OF SEMAPHORES**

Say I have 4 ready Process A,B,C,D and I start off with the semaphore s = 1

Process A will call wait() decrementing s = 0 and then it will enter its critical region.

Process A will then be kicked off the cpu and put to the back of the ready queue and process B is next is line so he is called,

Process B will then execute the wait function decrementing s = -1, now because s is a negative number B will be put into a waiting Queue.

Process C is next is line so he is called, Process C will then execute the wait function decrementing s = -2,now because s is still a negative number C will be put into a waiting Queue.

Process D is next is line so he is called, Process D will then execute the wait function decrementing s = -3,now because s is still a negative number C will be put into a waiting Queue.

Now its Process A turn to run again ,as it is half way through execution it will just continue from where it left off, Eventually process A will execute the POST function .

This will increment s = -2 and now B will get access to the CPU .since it has already executed WAIT(),it will continue from the line after until Process B itself calls the POST function

This will increment s = -1 and now C will get access to the CPU .since it has already executed WAIT(),it will continue from the line after until Process C itself calls the POST function

This will increment s = 0 and now D will get access to the CPU .since it has already executed WAIT(),it will continue from the line after until Process D itself calls the POST function

S now = 1 again

