CMP325
Operating Systems
Lecture 13, 14

S/W Based and H/W Based CSP Solutions

Fall 2021

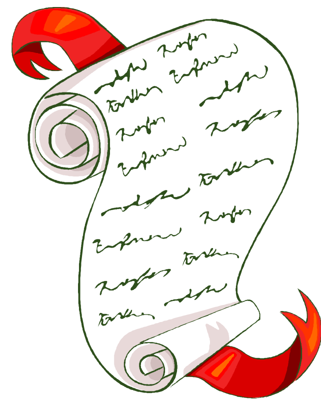
Arif Butt (PUCIT)

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

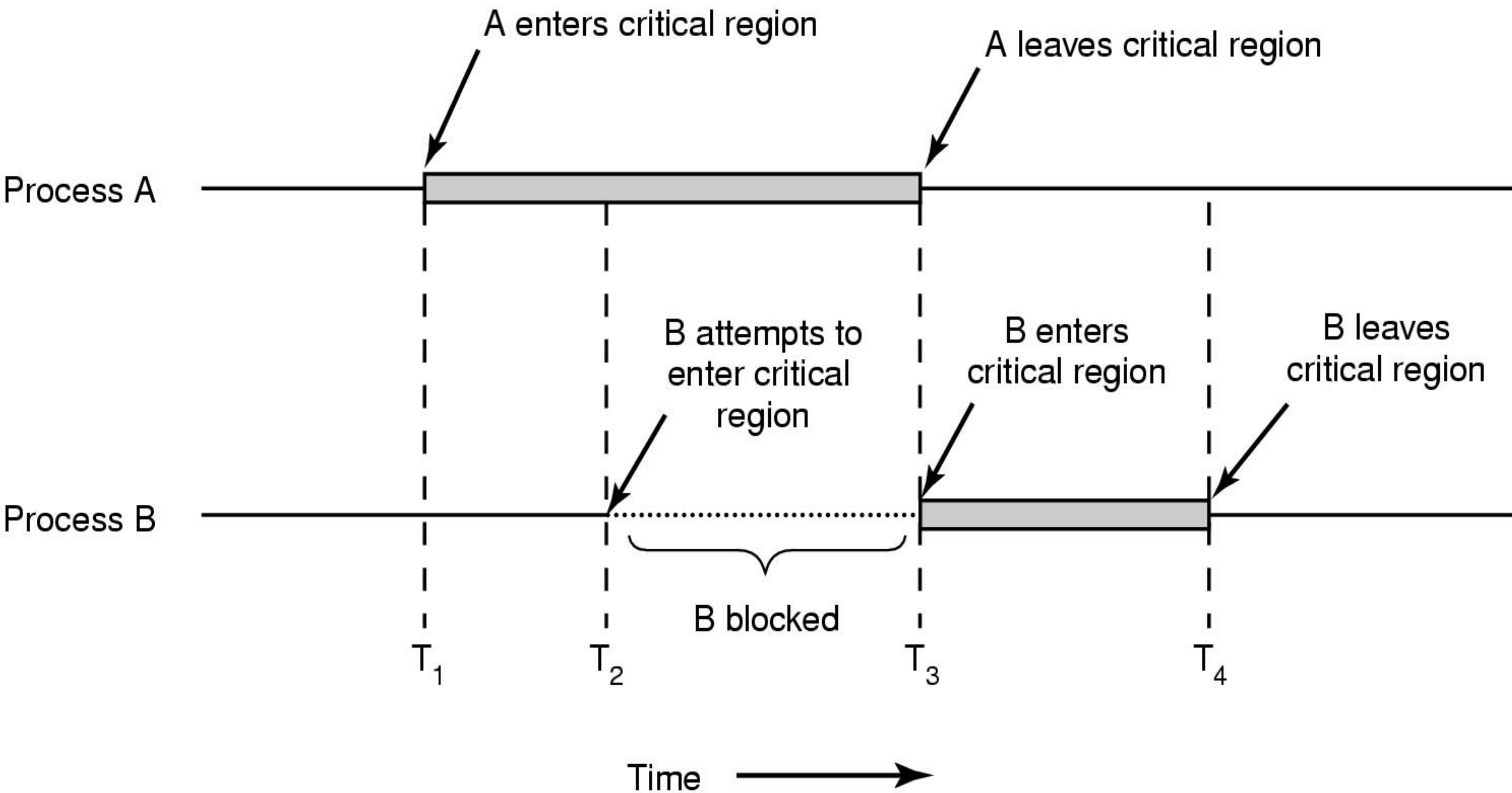
For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice video lectures on the subject of **OS with Linux** by Arif Butt available on the following link:
http://www.arifbutt.me/category/os-with-linux/

Today's Agenda



- Review of Previous Lecture.
- Possible Solutions of CS Problem
  - SW Based Solutions
    - Strict Alternation
    - Use of flags
    - Dekker's Algorithm
    - Peterson Algorithm
    - Busy Waiting and sleep() & wakeup()
    - Bakery Algorithm
  - HW Based Solutions
    - Disabling Interrupts
    - TSL
    - Swap
- Thread Synchronization using mutex

Critical Section Timeline - Example



Software Based Solutions

General Structure of CS Problem Solution

```
do {

        ENTERY SECTION

         <CS>                  //Access shared variables

        EXIT SECTION

          <RS>                 //Do other work

 }while(1);
```

Algorithm Using Strict Alternation

- Given by a Dutch mathematician Dekker for two processes P_0 and P_1.
- **turn** is a global/shared variable initialized to zero.
- If **turn** = i then P_i can enter in its CS otherwise it will wait.

Process P_0

```
do {
    while (turn!=0) ; //Entry Section
    <CS>
    turn = 1; //ExitSection
    <RS>
} while (1);
```

Process P_1

```
do {
    while (turn!=1) ; //Entry section
    <CS>
    turn = 0; //ExitSection
    <RS>
} while (1);
```

Algorithm Using Flags

- The limitation of strict alternation is solved in this algo.
- Instead of a single variable turn, take an array of two Boolean variables, **boolean flag[2]**; and initialize them to false; **flag [0] = flag [1] = false;**
- A process set its flag to true (showing its intention that it want to enter its CS) and check for the other process flag, if the other process flag is true keep spinning in loop.

Process P_0

```
do {
   flag[0] = true;
   while (flag[1]== true]);
   <CS>
   flag [0] = false;//ExitSection
   <RS>
} while (1);
```

Process P_1

```
do {
   flag[1] = true;
   while (flag[0]==true);
   <CS>
   flag [1] = false; //ExitSection
   <RS>
} while (1);
```

Use of Flags - Improved

- The limitation of previous algorithm is solved in this algo.
- Instead of waiting/spinning indefinitely in the while loop, the process set its flag to false, wait for a random period of time, set its flag back to true and then again try the while loop condition.

Process P_0

```
do {
    flag[0] = true;
    while (flag[1]) {
        flag[0] = false;
        wait(randno());
        flag[0] = true;
    }
    <CS>
    flag [0] = false;
    <RS>
} while (1);
```

Process P_1

```
do {
    flag[1] = true;
    while (flag[0])  {
        flag[1] = false;
        wait(randno());
        flag[1] = true;
    }
    <CS>
    flag [1] = false;
    <RS>
} while (1);
```

Peterson Algorithm

- The algorithm is given by Peterson, the person who wrote the first edition of our text book "OS concepts" in 1984.
- It combines the shared variables of previous algorithms.
- Keep two Boolean **flags** one for each process and a shared integer variable **turn**.

```
Boolean flag[2]; // initialized to false
int turn = 0;
```

- Before entering CS, each process set its flag to true and make turn equal to other process ID. (I am interested to go to my CS but if U wanna go, you go first). It then enters the while loop and checks whether the other process wants to enter its CS and is it his turn? if yes spin.

Peterson Algorithm (cont…)

Process P_0

```
do {
    flag[0] = true;
     turn = 1;
    while (flag[1] && turn == 1)
        ;  //spin
    <CS>
    flag [0] = false;
    <RS>
} while (1);
```

Process P_1

```
do {
    flag[1] = true;
     turn = 0;
    while (flag[0] && turn ==0)
        ;  //spin
    <CS>
    flag [1] = false;
    <RS>
} while (1);
```

Problem: Busy-Waiting

- Busy waiting means that a process is waiting for a condition to be satisfied, sitting in a tight loop, w/o relinquishing the CPU.
- Lets see a bigger picture:
  - Suppose that instead of only two, there are 100 cooperating processes.
  - One out of them is executing in its CS and the rest 50 out of 99 wants to get inside their CS.
  - So these 50 processes will be executing the while loop of their entry section.
  - Whenever the CPU is scheduled to them they keep spinning for the allocated time quantum. Thus wasting a very useful resource.

Problem: Busy-Waiting (…)

- Busy waiting not only waste precious CPU cycles, but it can also have unexpected effects like **priority inversion problem**.
  - Consider a computer with two cooperating processes, **H** (high priority) and **L** (low priority).
  - The scheduling rules are such that **H** is run whenever it is in ready state.
  - At a certain moment, **L** is in its CS, and **H** becomes ready to run. So **L** is preempted from it CS, and **H** executes.
  - **H** now begins busy waiting, now due to low priority **L** is never scheduled while **H** is running, **L** never gets the chance to leave its CS and execute the Exit section, so **H** loops forever.
  - This situation is referred to as the priority inversion problem.

Problem: Busy-Waiting (…)

- **Spin Locks are not appropriate for single CPUs. Why?**
  - Spin locks are not appropriate for single processor systems because the condition that would break a process out of the spin lock can be obtained only by executing a different process. If the spinning process is not relinquishing the CPU, other processes do not get the opportunity to execute and set the condition required by the spinning process to come out of loop.
  - In a multi processor system, other processes execute on other CPUs and thus set the condition required by the spinning process to come out of loop.

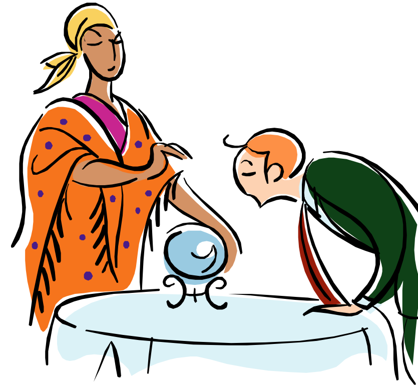
Sleep and Wakeup

- In busy waiting whenever a process wants to enter its CS, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.
- Busy waiting can be avoided by making a process sleep by relinquishing the CPU (in a queue) and block on a condition and wait to be awakened at some appropriate time in the future.
- Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep (in a queue) and having to wake it up when the appropriate program state is reached.
- **sleep()** is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The **wakeup()** call has one parameter, the process to be awakened.
- Next slide shows the producer consumer problem that uses these calls.

```
#define N 100                                       /* number of slots in the buffer */
int count = 0;                                      /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                                 /* repeat forever */
          item = produce_item( );                   /* generate next item */
          if (count == N) sleep( );                 /* if buffer is full, go to sleep */
          insert_item(item);                        /* put item in buffer */
          count = count + 1;                        /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);         /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                                 /* repeat forever */
          if (count == 0) sleep( );                 /* if buffer is empty, got to sleep */
          item = remove_item( );                    /* take item out of buffer */
          count = count − 1;                        /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer);  /* was buffer full? */
          consume_item(item);                       /* print item */
     }
}
```

Sleep and Wakeup

$100 QUESTION



- Is there a race condition in the Producer Consumer code shown on previous slide? Justify.

Solution to N-Process CS Problem

- Suppose there are n-cooperating processes $P_0, P_1, P_2, P_3, \ldots, P_{n-1}$. Each process has a segment of code called a CS in which the process may change shared data.
- **Bakery Algorithm.** (by Leslie Lamport)
  - Consider a bakery and whenever a person enters the bakery, he is given a token number (Tnumber). The person with the smallest token number is served first.
  - Let the bakery has two doors, each with a separate Tnumber dispenser.
  - Two persons enter the bakery exactly at the same time from the two doors and both are allocated the same Tnumber; e.g. 54.
  - When the man at the counter announces the number, 54, two persons get up and goes to the counter. Who is to be served first?
    - Ladies first.
    - Senior Citizen first.

Solution to N-Process CS Problem

- Bakery Algorithm (cont…).
  - If the bakery is your system and the persons inside are processes and the same thing happens; What to do?
    - Lets serve the process with the smaller ID.
  - Before entering its CS, every process gets a Tnumber. Holder of smaller Tnumber enters to its CS. If process P_i and P_j gets the same number then

if i < j then

P_i is served first;

else

P_j is served first;

Bakery Algorithm (cont…)

Example. Consider following six cooperating processes, with a Tnumber assigned to each. Give the sequence in which they will enter their CSs.

PID	**Tnumber**
P_0	8
P_1	5
P_2	0
P_3	5
P_4	6
P_5	2

The required sequence is $< P_5 \;\; P_1 \;\; P_3 \;\; P_4 \;\; P_0 >$

Bakery Algorithm (cont…)

Algorithm Semantics.

- Ticket numbering scheme always generate numbers in the increasing order of enumeration, i.e. 1, 2, 3, 3, 4, 5, … So every upcoming process is given a bigger number.
- **Notations**
  - (Tnumber, PID) is an ordered pair.
  - (a, b) < (c, d)   if (a < c) OR (a == c & b < d)
  - Max(…) is a function that is passed the existing Ticket numbers and it will return the largest out of them.
- **Data Structures**
  - Boolean choosing[n]; for each process there is a slot in this array initialized to false
  - int Tnumber[n]; for each process there is a number in this array initialized to zero

Bakery Algorithm

```
do {
    choosing[i] = true;
    Tnumber[i] = max(Tnumber[0], Tnumber[1], ..., Tnumber [n – 1])   + 1 ;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((Tnumber[j] != 0) && (Tnumber[j],j) < (Tnumber[i],i) ) ;
    }
    <CS>
    Tnumber[i] = 0;
    <RS>
} while (1);
```

Before choosing a Tnumber, every process will first set its choosing to true and later will set it to false.

This loop is going to compare the (no, id) pair of P_i with (no, id) pair of all other processes and finally selects which process goes to the CS

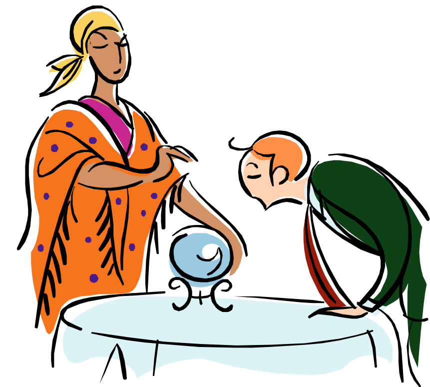
If Process P_j is in the process of getting a ticket number lets wait.

If P_j is having a Tnumber equal to 0, i.e. P_j is not interested to enter its CS. So break this while loop, go back to for loop, increment j and check next process

If P_j is interested to go to its CS, then check its ordered pair, with ordered pair of P_i. If P_j's (no, id) pair is less than P_i's pair then P_i wait in this loop, else move up to for loop, increment j and check next process.

After leaving the CS, P_i set its Tnumber to 0, showing that it is now not interested to enter its CS

Bakery Algorithm



$100 QUESTION

Why does Lamport used a variable called choosing? What will happen if we don't use it?

```
1   boolean choosing[n];
2   int Tnumber[n];
3   do{
4   choosing[i] = true;
5   Tnumber[i]  = max(Tnumber[0],Tnumber[1],...,Tnumber[n-1]) + 1;
6   choosing[i] = false;
7   for (j = 0 ; j < n ; j++){
8     while (choosing[j])
9            ; // do nothing
10    while ((Tnumber[j] != 0)&&(Tnumber[j], j) < (Tnumber[i],i)))
11           ;  // do nothing
12   }
13   < CRITICAL SECTION >
14   Tnumber[i] = 0;
15   < REMAINDER SECTION >
16   }
```

Scenario: Assume line # 1, 4, 6, 8 are commented

1. Lets assume there are two processes (Po and P1) are executing concurrently, and both reaches the line containing max function (line#5) at the same time.
2. Max function returns zero for both processes (addition is yet not done by any of the processes).
3. Let at this instant of time P0 is interrupted and P1 continues its execution.
4. P1 assigns 1 to its Tnumber[1] slot.
5. P1 continues its execution and reaches line #10, For j = 0, the first condition of the while loop will evaluate to false. For j = 1 the second condition of while loop condition will be evaluated and that will also be false. For j =2,3,4 the first condition of while loop will be evaluated to false. So P1 enters its CS. All is fine up till now.
6. Lets assume that P1 is preempted in its CS and now P0 which is there at line #5 starts its execution.
7. P0 also assigns 1 to its Tnumber[0] slot.
8. P0 continues its execution and reaches line #10, for j=0 and 1 the second condition of the while loop is evaluated to false. For j =2,3,4 the first condition of the while loop is evaluated to false. So P0 also enters its CS. ME violated.

Assume line # 1, 4, 6, 8 are there

1. P1 will wait at line #8 until P0 is also given Tnumber (of course 1). Remember both P0 and P1 has same Tnumber i.e. 1.
2. Now if P1 executes line #10 first, it will check the second condition of while loop and wait.
3. If P0 executes line #10, and will successfully enters the CS being senior most i.e. having the smallest ID.
4. Once it leaves the CS, it will set its Tnumber to 0, and P1 who is waiting at line#10 will come out of the loop and enter CS.

The reason of choosing is to prevent the second while loop (line #10) to be executed while a process Pj is in the process of getting its ticket number. So the process loops in the first while loop, i.e. while(choosing[j]);

Hardware Based Solutions

Disabling Interrupts

- H/W solution does not mean that we are using some IC to handle the CS problem. It basically mean that we are using instructions specified in the Instruction Set Architecture of a processor to handle CSP.
- In a uni-processor environment, we can handle CSP, if we forbid/mask interrupts (by setting MI bit to 1), while some shared variable is being modified.
  - However, if a user level program is given the ability to disable interrupts, then it can disable the timer interrupt. Thus context switching will not take place, thus allowing it to use the CPU w/o letting other processes to execute.
- In a multi-processor environment, it is not feasible to disable interrupts, because
  - It will only prevent processes from executing on the CPU in which interrupts are disabled. Processes can execute on other CPUs and therefore does not guarantee mutually exclusive access to program state.
  - Disabling interrupts on all CPUs gives a great performance loss.
- Disabling interrupts works, but safe to use only inside OS/kernel.

Structure of CS Problem by Disabling Interrupts

```
do {

    Disable Interrupts

     <CS>                //Access shared variables

    Enable Interrupts

      <RS>               //Do other work

}while(1);
```

Preemptive and Non Preemptive Kernels

- A preemptive kernel allows a process to be preempted while it is running in kernel mode, while a non preemptive kernel does not allow.
- A non preemptive kernel is free form race conditions on kernel data structures, because only one process is active in the kernel at a time.
- Preemptive kernels are difficult to design specially for SMP architectures (since two instructions from two different kernel mode processes can run simultaneously thus making race conditions difficult to handle).
- **Why preemptive kernel?**
  - More suitable for real time programming, as it will allow a real time process to preempt a process currently running in the kernel.
  - More responsive.
  - **Examples**. Windows 2000 and XP have non preemptive kernels. Linux 2.6 onwards, kernel is preemptive.

TSL (Test-Set-Lock) Instruction

- TSL instruction is always executed atomically. Thus if two TSL instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- TSL instruction returns whatever (T/F) is passed to it and sets the lock to true.

```
boolean TestAndSet(boolean &lock)
{
  boolean rv = lock; //READ
   lock = true;        //MODIFY
   return rv;          //RETURN
}
```

Solution to CSP usingTSL

Mutual Exclusion and Progress holds but

Bounded wait doesn't. HOW? & WHY?

```
boolean TestAndSet(boolean & lock)
{
     boolean rv = lock; //READ
    lock = true; //MODIFY
    return rv;  //RETURN
}
```

```
boolean lock = false;
```

Process P_i

```
do {
    while (TestAndSet(lock))
         ;
    <CS>
    lock = false;
    <RS>
} while (1);
```

Process P_j

```
do {
    while (TestAndSet(lock))
         ;
    <CS>
    lock = false;
    <RS>
} while (1);
```

SWAP Instruction

- swap instruction is always executed atomically. Thus if two swap instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```
void swap(boolean &a, boolean &b)
{
  boolean temp = a;
   a = b;
   b = temp;
}
```

Solution to CSP using SWAP

Mutual Exclusion and Progress holds but Bounded wait doesn't. HOW? & WHY?

```
void swap(boolean &a, boolean &b)
{
    boolean temp = a;
    a = b;
    b = temp;
}
```

```
boolean lock = false;
```

Process P_i

```
do {
    boolean key = true;
    while (key == true)
        swap(lock, key);
   <CS>
   lock = false;
   <RS>
} while (1);
```

Process P_j

```
do {
    boolean key = true;
    while (key == true)
        swap(lock, key);
   <CS>
   lock = false;
   <RS>
} while (1);
```

Achieving Bounded wait using TestAndSet

```
boolean lock = false;
boolean waiting[n]; // all set to false

do{
  waiting[i] = true;
  boolean key = true;
  while (waiting[i] && key)
      key = TestAndSet(lock);
  waiting[i] = false;

  < CRITICAL SECTION >

  j = (i+1) % n;
  while ((j != i) && !waiting[j])
      j = (j + 1) % n;
  if (j == i)
       lock = false;
  else
       waiting[j] = false;

  < REMAINDER SECTION >

  }while(1);
```

Achieving Bounded wait using SWAP

```
boolean lock = false;
boolean waiting[n]; // all set to false

do{
  waiting[i] = true;
  boolean key = true;
  while (waiting[i] && key)
      swap(lock, key);
  waiting[i] = false;

  < CRITICAL SECTION >

  j = (i+1) % n;
  while ((j != i) && !waiting[j])
      j = (j + 1) % n;
  if (j == i)
       lock = false;
  else
       waiting[j] = false;

  < REMAINDER SECTION >

  }while(1);
```

LIVE FREE OR DIE

UNIX*

TRADEMARK OF BELL LABS*

Example (Race Condition)

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

```
void * inc(void * arg){
   for(long i=0;i<100000000;i++)
       balance++;
   pthread_exit(NULL);
}
```

```
void * dec(void * arg){
   for(long i=0;i<100000000;i++)
       balance--;
   pthread_exit(NULL);
}
```

Thread Synchronization using Mutexes

- The pthread library provides three synchronization mechanisms; *mutexes*, *joins* and *condition variables*
- **Mutex**
  - A mutex is used to achieve both mutual exclusion as well as serialization.
  - A mutex is a special type of lock that only one thread may lock at a time.
  - If a thread locks a mutex and later a second thread also tries to lock the same mutex, the second thread is blocked. When the first thread unlocks the mutex, the second thread is allowed to resume execution.
  - Linux guarantees that race condition do not occur among threads attempting to lock a mutex.

Typical way to use a mutex

1. Create an initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one thread succeed and that thread owns the mutex
4. The owner thread carry out operations on shared data.
5. The owner thread unlock the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

Important Library Calls

Mutex Initialization:

```
static pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_lock(pthread_mutex_t * mptr);
```

- Calling thread will lock the mutex object referenced by mptr. If mutex is already locked, the calling thread shall block until the mutex become available.
- Used when a thread is going to enter its CS.

```
int pthread_mutex_unlock(pthread_mutex_t * mptr);
```

- This call should be made only by the owner thread. This will release the mutex object referenced by mptr. If there are threads blocked on the mutex object referenced by mptr when the unlock() is called, the scheduling policy shall determine which thread shall acquire the mutex
- Used when a thread comes out of the CS

Example (Solution Race Condition)

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
pthread_mutex_t mut;
int main(){
   pthread_t t1, t2;
   pthread_mutex_init(&mut, NULL);
   pthread_create(&t1, NULL, inc,NULL);
   pthread_create(&t2, NULL, dec,NULL);
   pthread_join(t1,NULL);   pthread_join(t2,NULL);
   pthread_mutex_destroy(&mut);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

```
void * dec(void * arg){
   for(long i=0;i<100000000;i++){
       pthread_mutex_lock(&mut);
       balance--;
     pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
}
```

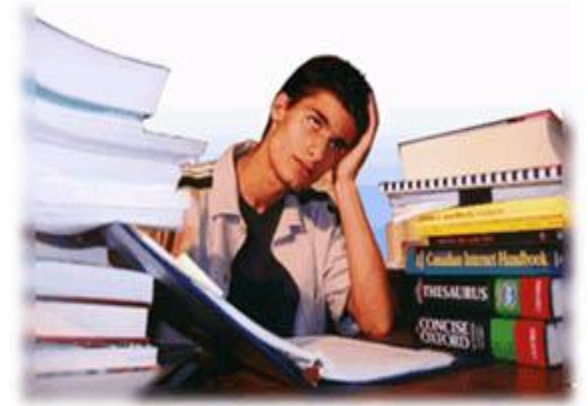
```
void * inc(void * arg){
   for(long i=0;i<100000000;i++){
       pthread_mutex_lock(&mut);
       balance++;
     pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
}
```

Mutex Mistakes

- Only the owner of the mutex should unlock the mutex
- Do not lock a mutex that is already locked
- Do not unlock a mutex that is already unlocked
- Do not destroy a locked mutex

SUMMARY

We're done for now, but Todo's for you after this lecture…



- Go through the slides and Book Sections: 6.1 to 6.4
- Try to make an understanding about the non determinism of concurrent multi threaded programs.
- Try hand/mind execution of the sample programs discussed in class to see whether they fulfill the characteristics of CS problem solution.
- Write a multithreaded C program that is passed two filenames as command line arguments. It counts the number of characters in those two files in a global variable. Create two threads for the task. Handle race condition.

If you have problems visit me in counseling hours. . . .