CSE 513
Introduction to Operating Systems

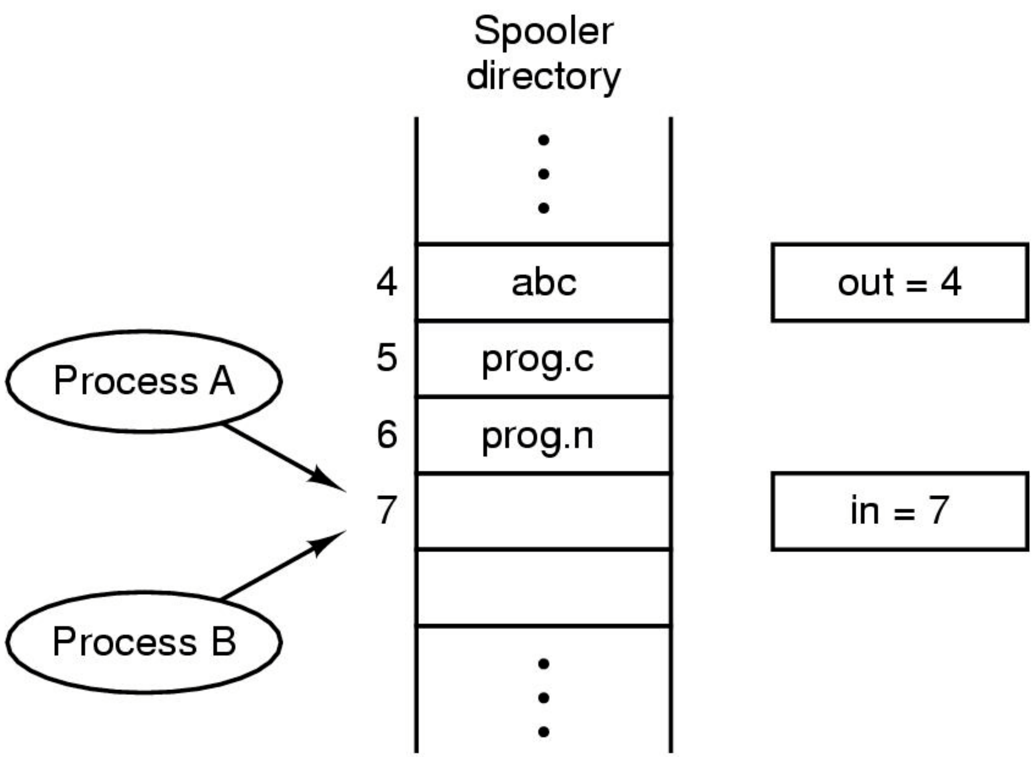
Class 3 - Interprocesses Communication & Synchronization

Jonathan Walpole
Dept. of Comp. Sci. and Eng.
Oregon Health and Science University

Race conditions

- **What is a race condition?**
  - two or more processes have an inconsistent view of a shared memory region (I.e., a variable)
- **Why do race conditions occur?**
  - values of memory locations replicated in registers during execution
  - context switches at arbitrary times during execution
  - processes can see "stale" memory values in registers
- **What solutions can we apply?**
  - prevent context switches by preventing interrupts?
  - make processes coordinate with each other to ensure mutual exclusion in accessing "critical sections" of code

Counter increment race condition



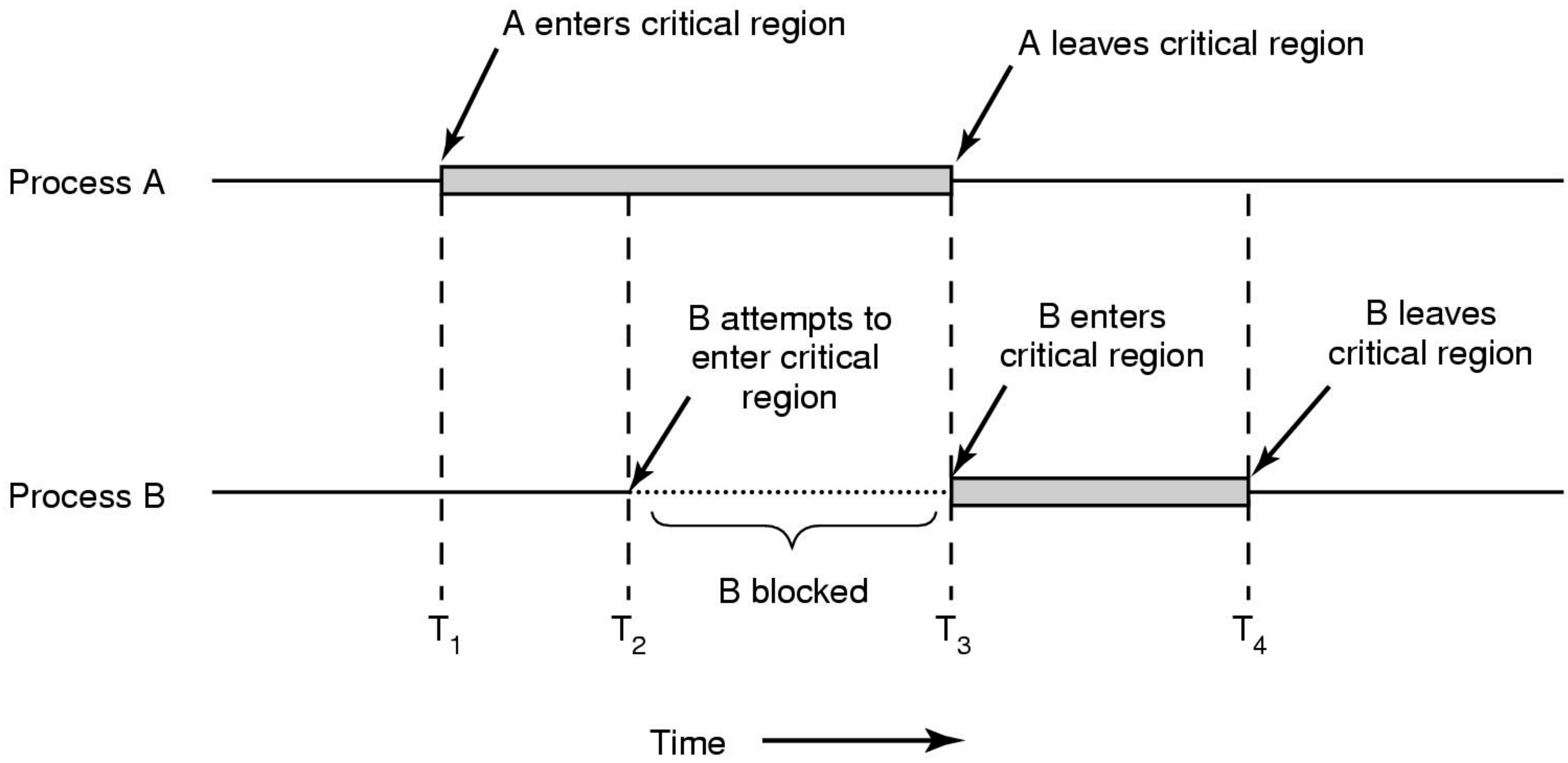
Incrementing a counter (load, increment, store)

Context switch can occur after load and before increment!

Mutual exclusion conditions

- No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical region may block another process
- No process must wait forever to enter its critical region

Critical regions with mutual exclusion



Mutual exclusion using critical regions

How can we implement mutual exclusion?

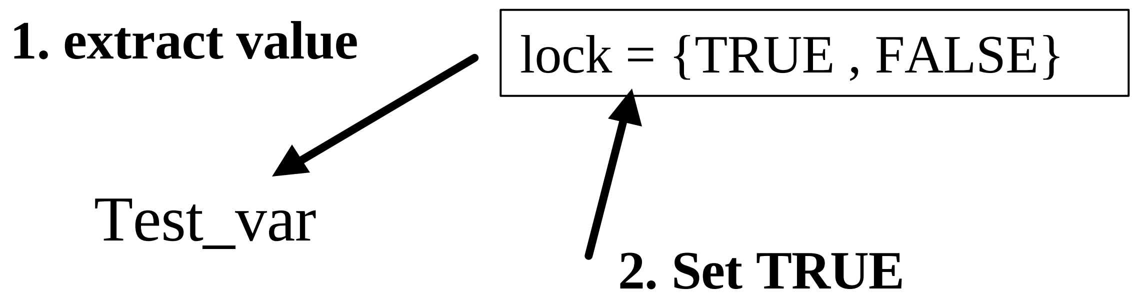
- **What about using a binary "lock" variable in memory and having processes check it and set it before entry to critical regions?**

- **Many computers have *some limited* hardware support for setting locks**
  - "Atomic" Test and Set Lock instruction
  - "Atomic" compare and swap operation

- **Solves the problem of**
  - Expressing intention to enter C.S.
  - Actually setting a lock to prevent concurrent access

Test and Set Lock

- **Test-and-set does two things atomically:**
  - Test a lock (whose value is returned)
  - Set the lock



- **Lock obtained when the return value is FALSE**
- **If TRUE, someone already had the lock (and still has it)**

Test and Set Lock

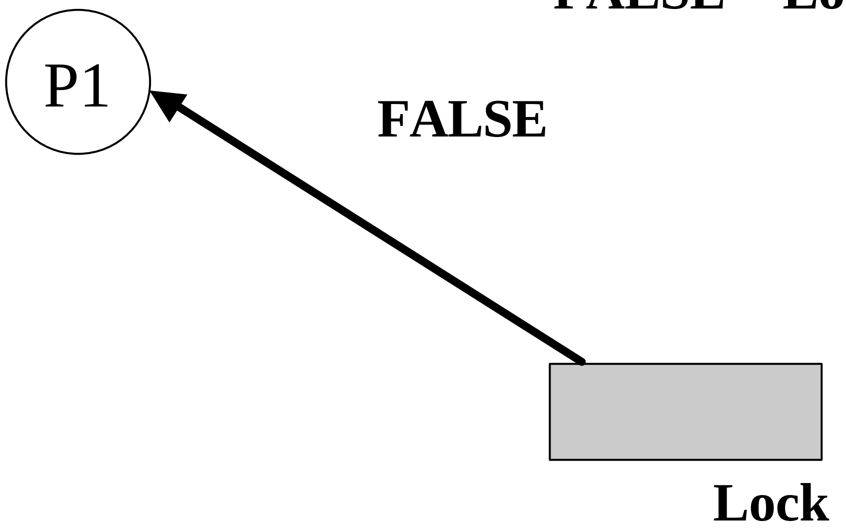
FALSE

Lock

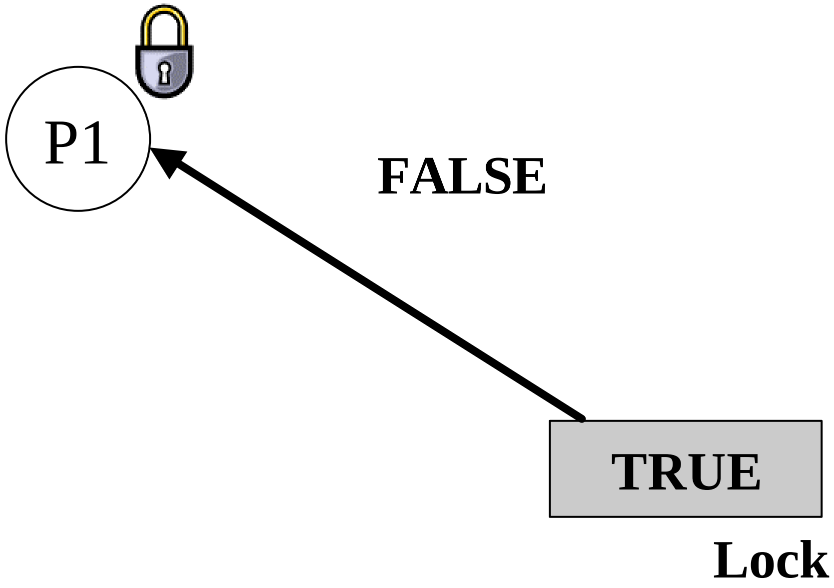
Test and Set Lock

Test and Set Lock

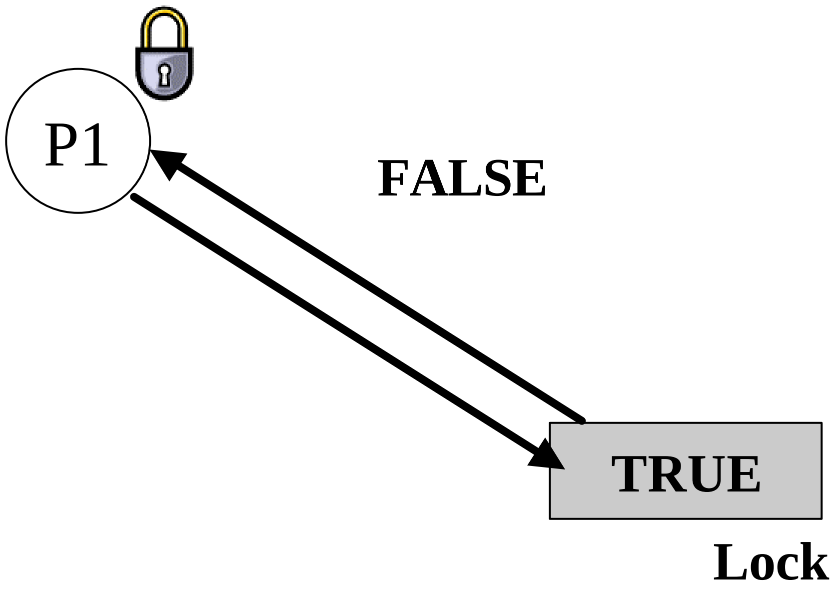
FALSE = Lock Available!!



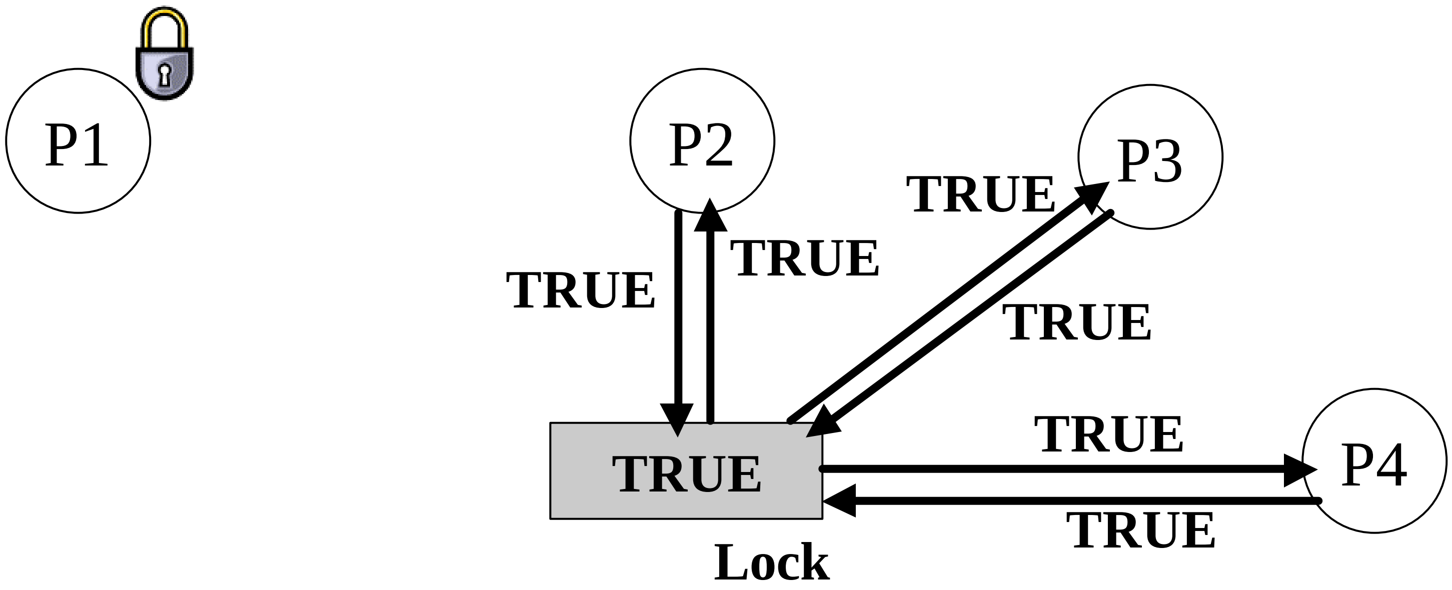
Test and Set Lock



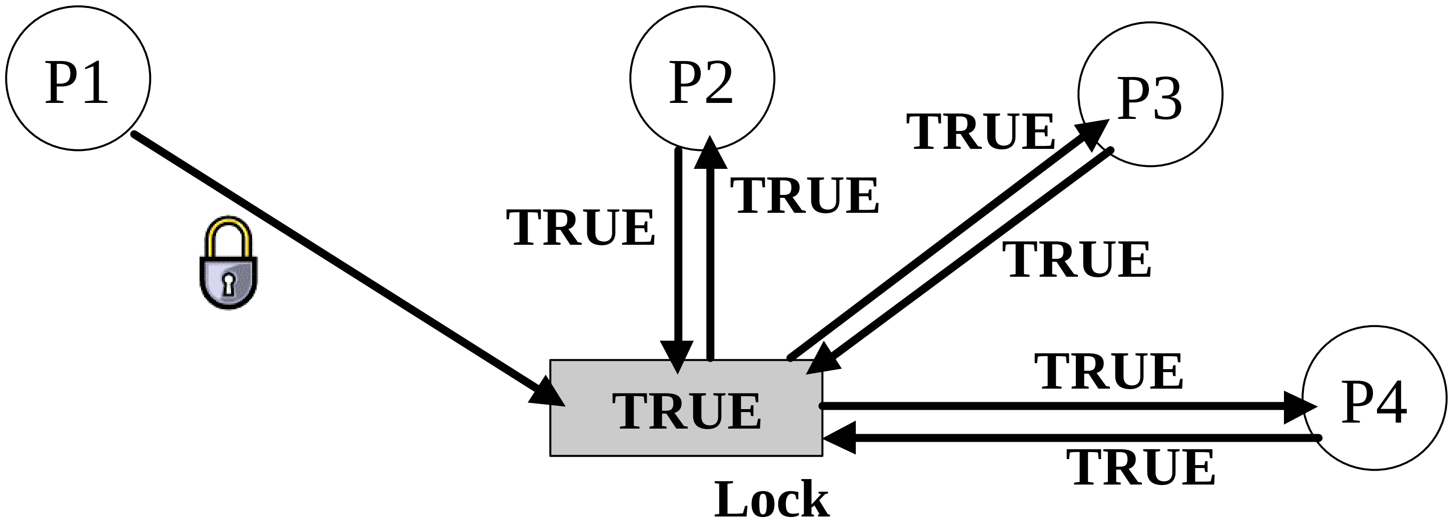
Test and Set Lock



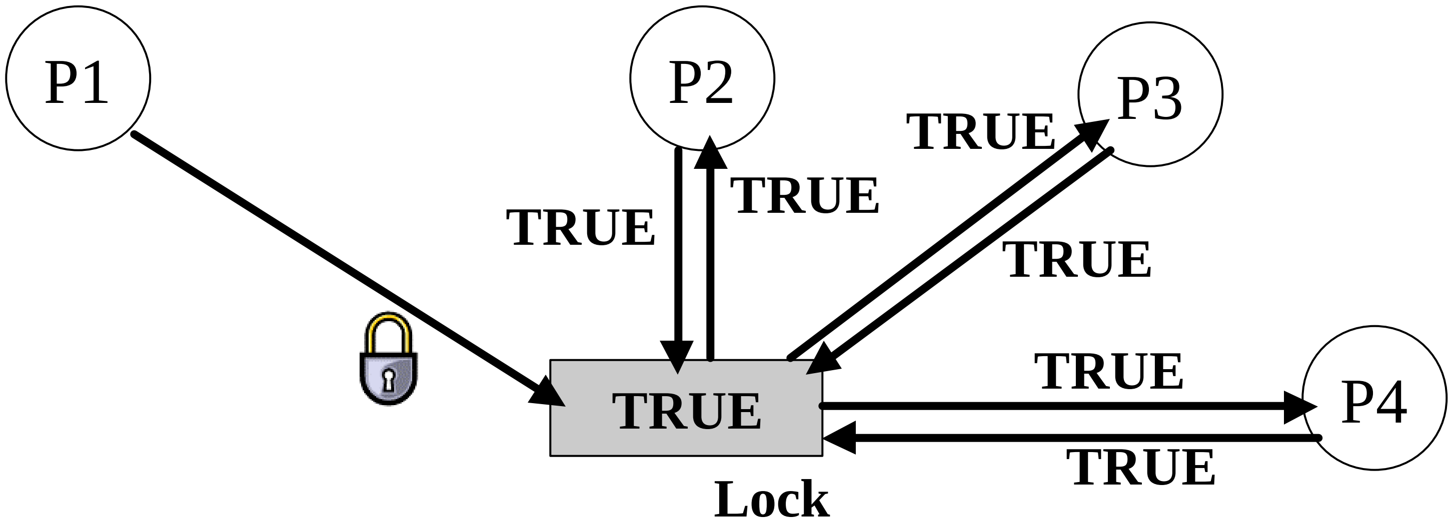
Test and Set Lock



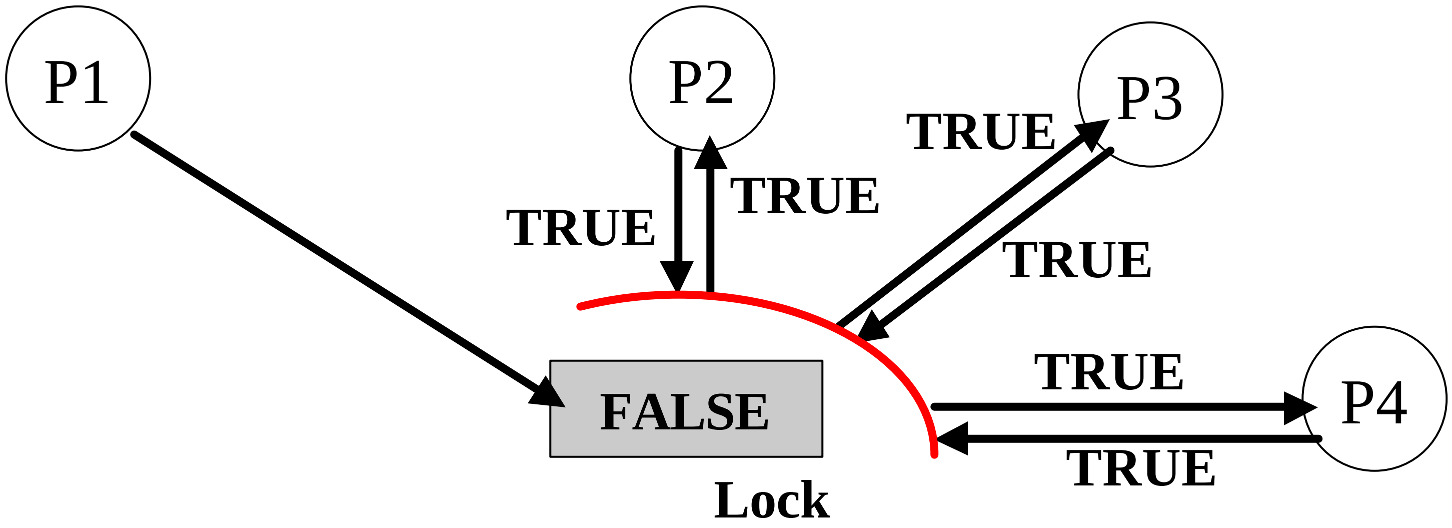
Test and Set Lock



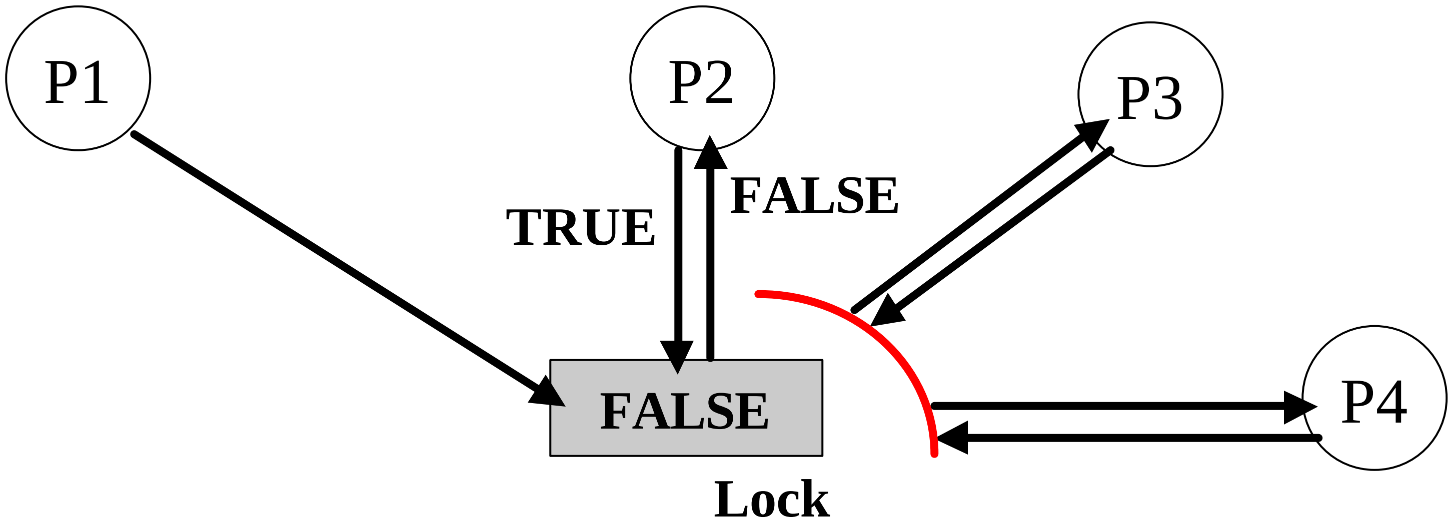
Test and Set Lock



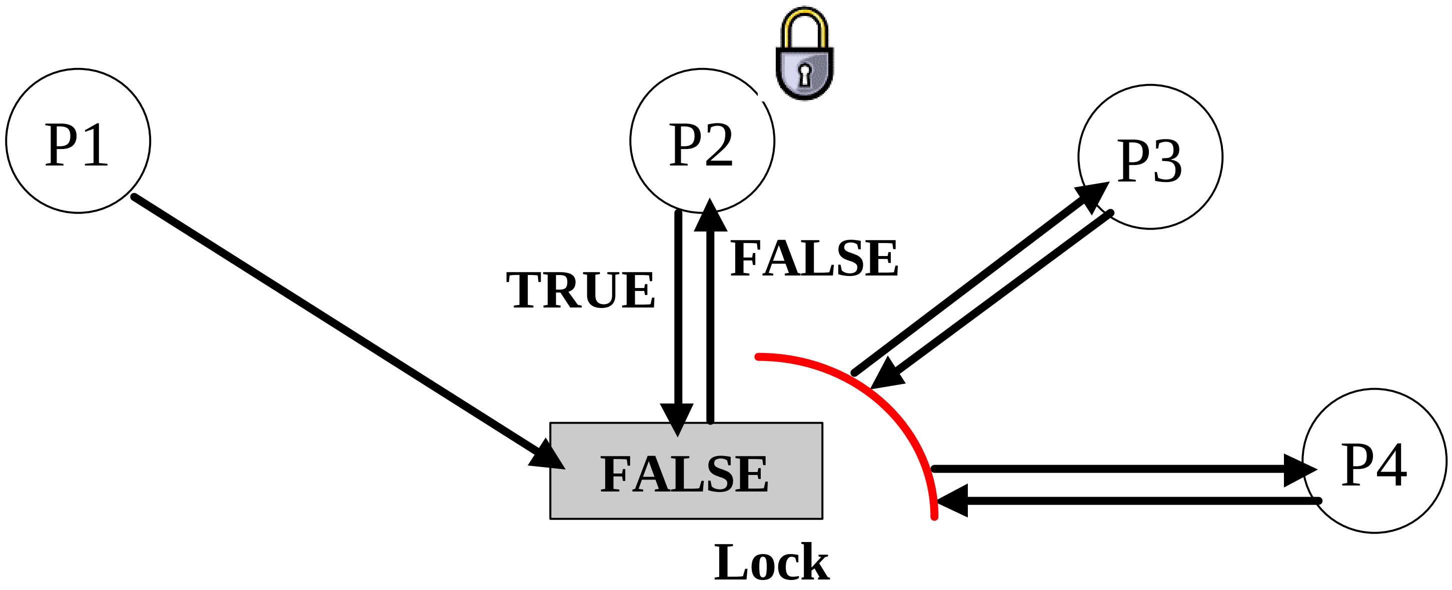
Test and Set Lock



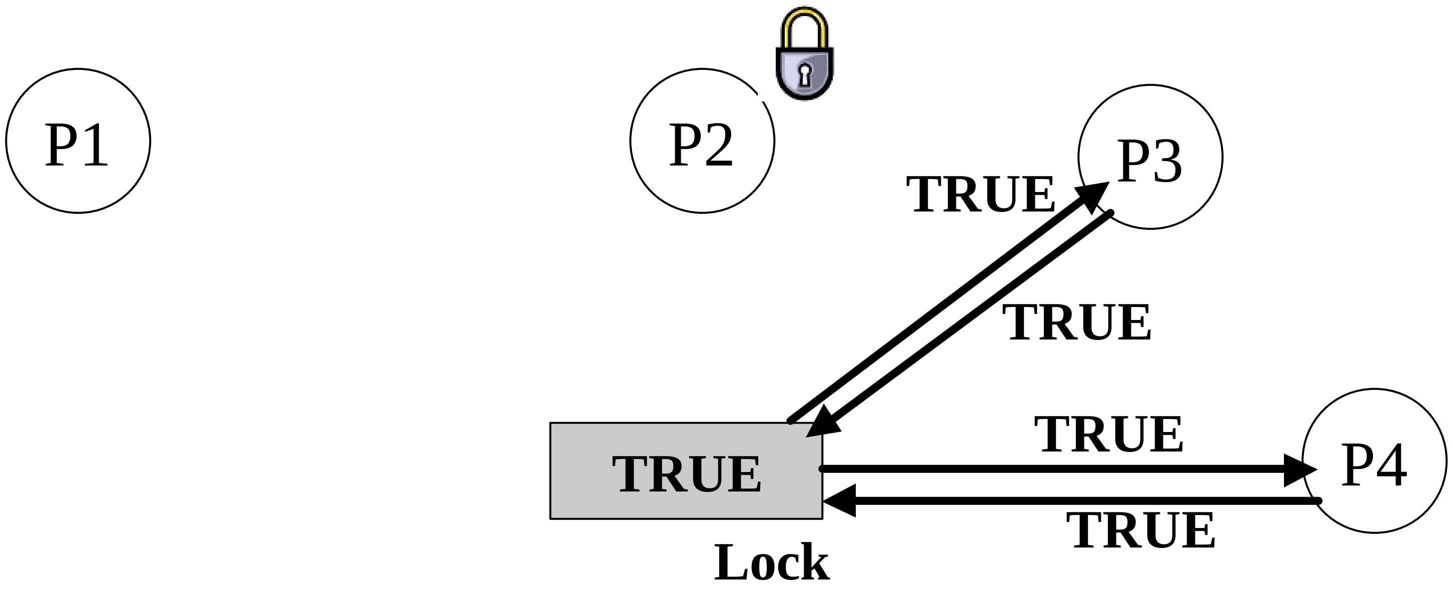
Test and Set Lock



Test and Set Lock



Test and Set Lock



Critical section entry code with TSL

I

```
repeat
  while(TSL(lock))
    no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

J

```
repeat
  while(TSL(lock))
    no-op;
  critical section
  Lock = FALSE;
  remainder section
until FALSE
```

Guaranteed that only one process returns with FALSE when a lock is returned to the system and others are waiting to act on the lock

Generalized primitives for critical sections

- **Thus far, the solutions have used *busy waiting***
  - a process consumes CPU resources to evaluate when a lock becomes free
  - on a single CPU system busy waiting prevents the lock holder from running, completing the critical section and releasing the lock!
  - it would be better to block instead of busy wait (on a single CPU system)

- **Blocking synchronization primitives**
  - *sleep* – allows a process to sleep on a *condition*
  - *wakeup* – allows a process to signal another process that a *condition* it was waiting on is true
  - but how can these be implemented?

Blocking synchronization primitives

- **Sleep and wakeup are system calls**
  - OS can implement them by managing a data structure that records who is blocked and on what condition
  - but how can the OS access these data structures atomically?
- **Concurrency in the OS: context switches and interrupts**
  - the OS can arrange not to perform a context switch while manipulating its data structures for sleep and wakeup
  - but what about interrupts?
  - what if interrupt handlers manipulate the sleep and wakeup data structures? What if they need synchronization?
  - how can the OS synchronize access to its own critical sections?

Disabling interrupts

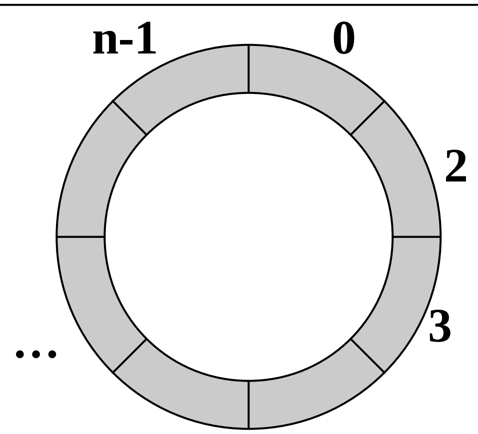
- **Disabling interrupts in the OS vs disabling interrupts in user processes**
  - why not allow user processes to disable interrupts?
  - is it ok to disable interrupts in the OS?
  - what precautions should you take?

Generic synchronization problems

Producer/Consumer with busy waiting

```
process producer{
    while(1){
      //produce char c
      while (count==n)
        no_op;
      buf[InP] = c;
      InP = InP + 1 mod n
      count++;
      }
}
```

```
process consumer{
    while(1){
      while (count==0)
        no_op;
      c = buf[OutP];
      OutP = OutP + 1 mod n
      count--;
      //consume char
      }
}
```




```
Global variables:
    char buf[n]
    int InP, OutP; // [0-n-1]
    int count
```

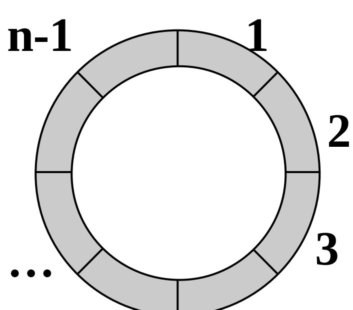
Problems with busy waiting solution

- Producer and consumer can't run at the same time
- Count variable can be corrupted if context switch occurs at the wrong time
- Bugs difficult to track down

Producer/Consumer with blocking

```
process producer{
•   while(1){
•     //produce char c
•     if (count==n)
•       sleep(full);
•     buf[InP] = c;
•     InP = InP + 1 mod n
•     count++;
•     if (count == 1)
•       wakeup(empty);
•     }
}
```

```
process consumer{
•   while(1){
•     while (count==0)
•       sleep(empty);
•     c = buf[OutP];
•     OutP = OutP + 1 mod n
•     count--;
•     if (count == n-1)
•       wakeup(full);
•     //consume char
•     }
}
```




```
Global variables:
    char buf[n]
    int InP, OutP; // [0-n-1]
    int count
```

Problems with the blocking solution

- Count variable can be corrupted
- Increments or decrements may be lost
- Both processes may sleep forever
- Buffer contents may be over-written

- Code that manipulates count must be made a critical section and protected using mutual exclusion
- Sleep and wakeup must be implemented as system calls
- OS must use synchronization mechanisms (TSL or interrupt disabling) in its implementation of sleep and wake-up … I.e., the critical sections of OS code that manipulate sleep/wakeup data structures must be protected using mutual exclusion

Semaphores

- **An abstract data type that can be used for condition synchronization and mutual exclusion**

- **Integer variable with two operations:**
  - *down* (sema_var)

    decrement sema_var by 1, if possible

    if not possible, “wait” until possible
  - *up*(sema_var)

    increment sema_var by 1

- **Both *up()* and *down()* are assumed to be atomic**
  - made to be atomic by OS implementation

Semaphores

- **There are multiple names for semaphores**
  - *Down(S), wait(S), P(S)*
  - *Up(S), signal(S), V(S)*

- **Semaphore implementations**
  - Binary semaphores (mutex)
    - **support mutual exclusion (lock either set or free)**
  - Counting semaphores
    - **support multiple values for more sophisticated coordination and controlled concurrent access among processes**

Using Semaphores for Mutex

```
semaphore mutex = 1
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

Using Semaphores for Mutex

semaphore mutex = 0

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

Using Semaphores for Mutex

```
semaphore mutex = 0
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

Using Semaphores for Mutex

semaphore mutex = 1

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

Using Semaphores for Mutex

***semaphore* mutex = 1**

Check again to see if it can be decremented

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

```
repeat
  down(mutex);
  critical section
  up(mutex);
  remainder section
until FALSE
```

In class exercise…

- **Implement producer consumer solution:**

Counting semaphores in producer/consumer

```
Global variables
  semaphore full_buffs = 0;
  semaphore empty_buffs = n;
  char buff[n];
  int InP,  OutP;
```

```
process producer{
•   while(1){
•     //produce char c

•     down(empty_buffs);
•     buf[InP] = c;
•     InP = InP + 1 mod n
•     up(full_buffs);

•   }
}
```

```
process consumer{
•   while(1){

•     down(full_buffs);
•     c = buf[OutP];
•     OutP = OutP + 1 mod n
•     up(empty_buffs);

•     //consume char
•     }
}
```

Implementing semaphores

- **Generally, the hardware has some simple mechanism to support semaphores**
  - Control over interrupts (almost all)
  - Special atomic instructions in ISA
    - **test and set lock**
    - **compare and swap**
- **Spin-Locks vs. Blocking**
  - Spin-locks (busy waiting)
    - **may waste a lot of cycles on uni-processors**
  - Blocking
    - **may waste a lot of cycles on multi-processors**

Implementing semphores

- **Blocking**

```
struct semaphore{
    int val;
    list L;
    }
```

```
Down(semaphore sem)
  DISABLE_INTS
    sem.val--;
    if (sem.val < 0){
      add proc to sem.L
      block(proc);
      }
  ENABLE_INTS
```

```
Up(semaphore sem)
  DISABLE_INTS
    sem.val++;
    if (sem.val <= 0) {
      proc = remove next
         proc from sem.L
      wakeup(proc);
      }
  ENABLE_INTS
```

Semaphores in UNIX

- **User-accessible semaphores in UNIX are somewhat complex**
  - each up and down operation is done atomically on an "array" of semaphores.

- **********WORDS OF WARNING **********
  - Semaphores are allocated by (and in) the operating system (number based on configuration parameters).
  - Semaphores in UNIX ARE A SHARED RESOURCE AMONG EVERYONE (most implementations are).
  - REMOVE your semaphores after you are done with them.

Typical usage

```
main(){
       int sem_id;
       sem_id = NewSemaphore(1);
       ...
       Down(sem_id);

       [CRITICAL SECTION]

       Up (sem_id);

       ...
       FreeSemaphore(sem_id);
       }
```

Managing your UNIX semaphores

- **Listing currently allocated ipc resources**

```
ipcs
```

- **Removing semaphores**

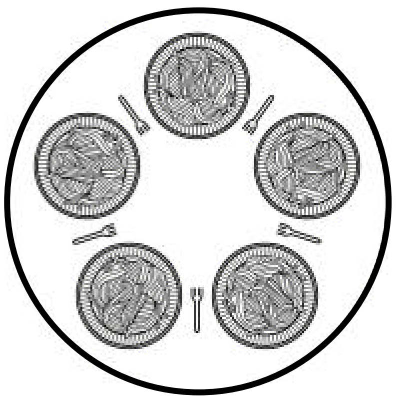
```
ipcrm -s <sem number>
```

Classical IPC problems

- **There are a number of "classic" IPC problems including:**
  - Producer / Consumer synchronization
  - The dining philosophers problem
  - The sleeping barber problem
  - The readers and writers problem
  - Counting semaphores out of binary semaphores

Dining Philosophers Problem

- **Five philosophers sit at a table**
- **Between each philosopher there is one chopstick**
- **Philosophers:**



```
while(!dead){
  Think(hard);
  Grab first chopstick;
  Grab second chopstick;
  Eat;
  Put first chopstick back;
  Put second chopstick back;
  }
```

- ***Why do they need to synchronize?***
- ***How should they do it?***

Dining philospher's solution???

- Why doesn't this work?

```
#define N 5
Philosopher()
{
  while(!dead){
    Think(hard);
    take_fork(i);
    take_fork((i+1)% N);
    Eat(alot);
    put_fork(i);
    put_fork((i+1)% N);
    }
}
```

Dining philosopher's solution (part 1)

```
#define N           5          /* number of philosophers */
#define LEFT        (i+N−1)%N  /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0          /* philosopher is thinking */
#define HUNGRY      1          /* philosopher is trying to get forks */
#define EATING      2          /* philosopher is eating */
typedef int semaphore;         /* semaphores are a special kind of int */
int state[N];                  /* array to keep track of everyone's state */
semaphore mutex = 1;           /* mutual exclusion for critical regions */
semaphore s[N];                /* one semaphore per philosopher */

void philosopher(int i)        /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {             /* repeat forever */
        think( );              /* philosopher is thinking */
        take_forks(i);         /* acquire two forks or block */
        eat( );                /* yum-yum, spaghetti */
        put_forks(i);          /* put both forks back on table */
    }
}
```

Dining philosopher's solution (part 2)

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
      test(i);                      /* try to acquire 2 forks */
      up(&mutex);                   /* exit critical region */
      down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = THINKING;          /* philosopher has finished eating */
      test(LEFT);                   /* see if left neighbor can now eat */
      test(RIGHT);                  /* see if right neighbor can now eat */
      up(&mutex);                   /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Dining Philosophers

- Is this correct?
- What does it mean for it to be correct?
- Is there an easier way?

Sleeping Barber Problem

Sleeping barber

- **Barber**
  - if there are people waiting for a hair cut bring them to the barber chair, and give them a haircut
  - else go to sleep

- **Customer:**
  - if the waiting chairs are all full, then leave store.
  - if someone is getting a haircut, then wait for the barber to free up by sitting in a chair
  - if the barber is sleeping, then wake him up and get a haircut

Solution to the sleeping barber problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

The readers and writers problem

- Readers and writers want to access a database
- Multiple readers can proceed concurrently
- Writers must synchronize with readers and other writers
- Maximize concurrency
- Prevent starvation

One solution to readers and writers

```
typedef int semaphore;            /* use your imagination */
semaphore mutex = 1;              /* controls access to 'rc' */
semaphore db = 1;                 /* controls access to the database */
int rc = 0;                       /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {               /* repeat forever */
          down(&mutex);           /* get exclusive access to 'rc' */
          rc = rc + 1;            /* one reader more now */
          if (rc == 1) down(&db); /* if this is the first reader ... */
          up(&mutex);             /* release exclusive access to 'rc' */
          read_data_base();       /* access the data */
          down(&mutex);           /* get exclusive access to 'rc' */
          rc = rc – 1;            /* one reader fewer now */
          if (rc == 0) up(&db);   /* if this is the last reader ... */
          up(&mutex);             /* release exclusive access to 'rc' */
          use_data_read();        /* noncritical region */
     }
}

void writer(void)
{
     while (TRUE) {               /* repeat forever */
          think_up_data();        /* noncritical region */
          down(&db);              /* get exclusive access */
          write_data_base();      /* update the data */
          up(&db);                /* release exclusive access */
     }
}
```

Counting semaphores

- A binary semaphore can only take on the values of [0, 1].
- Class exercise: create a counting semaphore (generalized semaphore that we discussed previously) using just a binary semaphore!!

Possible solution

```
Semaphore S1, S2, S3; // BINARY!!
int C = N;  // N is # locks
```

```
down_c(sem){
  downB(S3);
  downB(S1);
  C = C – 1;
  if (C<0) {
    upB(S1);
    downB(S2);
    }
  else {
    upB(S1);
    }
  upB(S3);
  }
```

```
up_c(sem){
  downB(S1);
  C = C + 1;
  if (C<=0) {
    upB(S2);
    }
  upB(S1);
  }
```