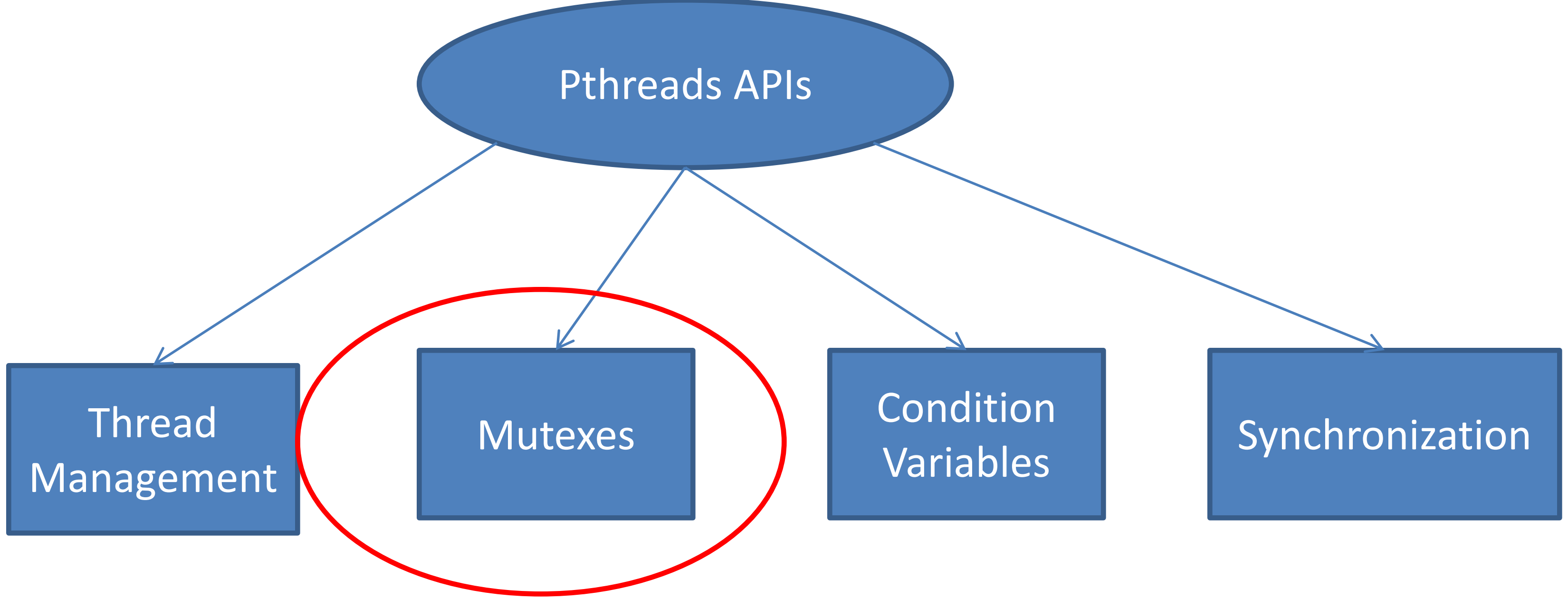
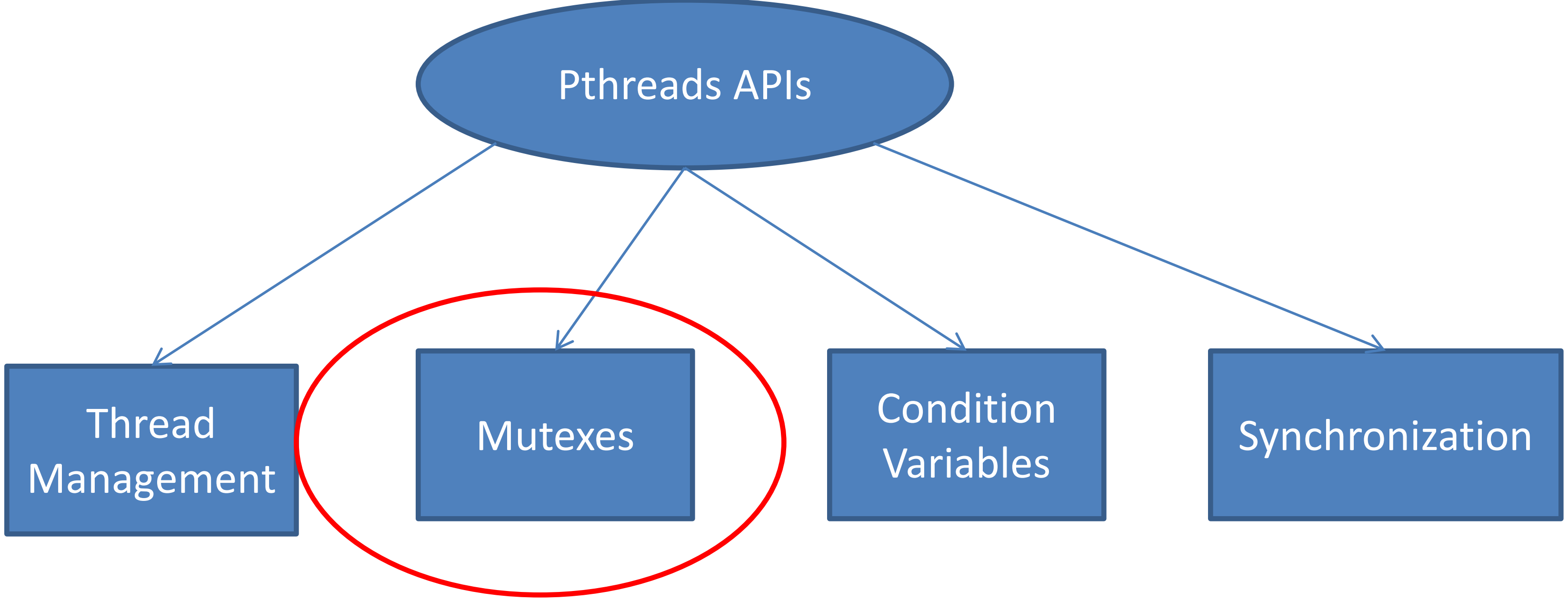
Operating Systems

Concurrency III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



- Mutex = Mutual Exclusion
- One of the primary means of protecting shared data when multiple writes occur.
- Acts like a lock protecting access to a shared data resource
- Only one thread can lock (or own) a mutex variable at any given time.



A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one thread succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally, the mutex is destroyed

It is up to the code writer to ensure that the necessary threads all make the mutex lock and unlock calls correctly.

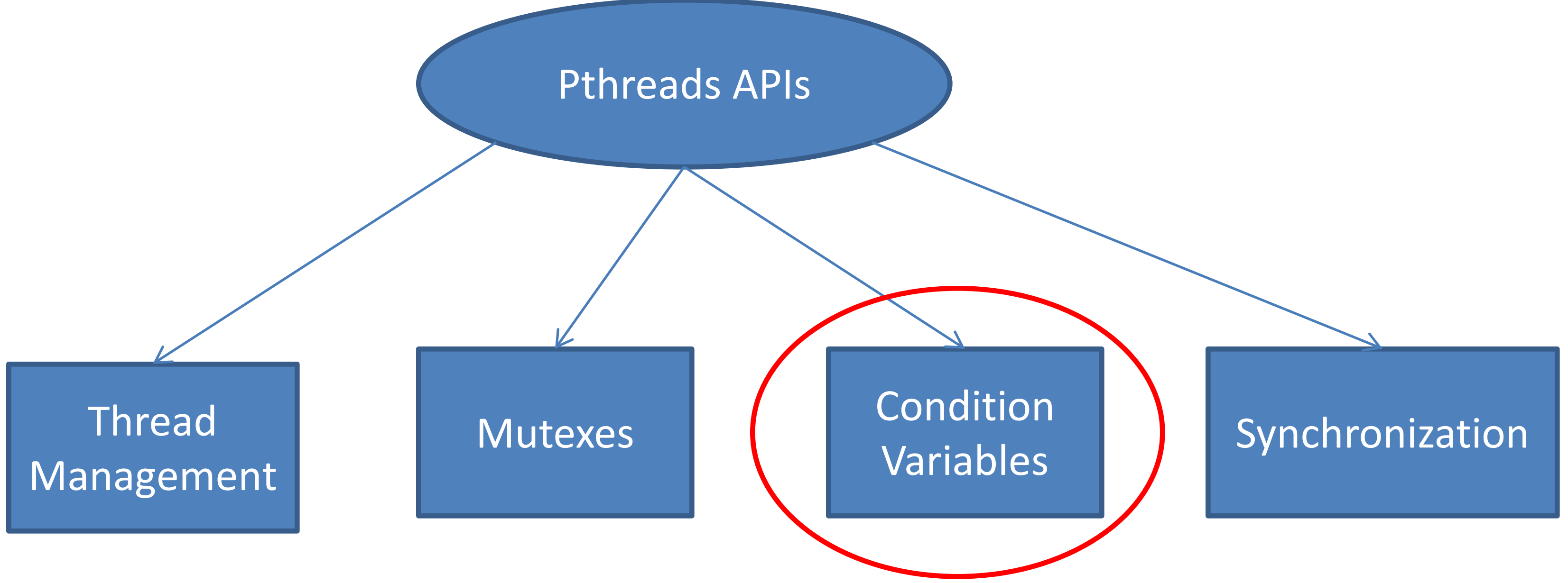
Mutex in POSIX Threads

- pthread_mutex_t p;
- pthread_mutex_init(&p, NULL);
- pthread_mutex_lock(&p);
- pthread_mutex_unlock(&p);
- pthread_mutex_destroy(&p);

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

```
pthread_mutex_lock(&p);
counter = counter + 1;
pthread_mutex_lock(&p);
```

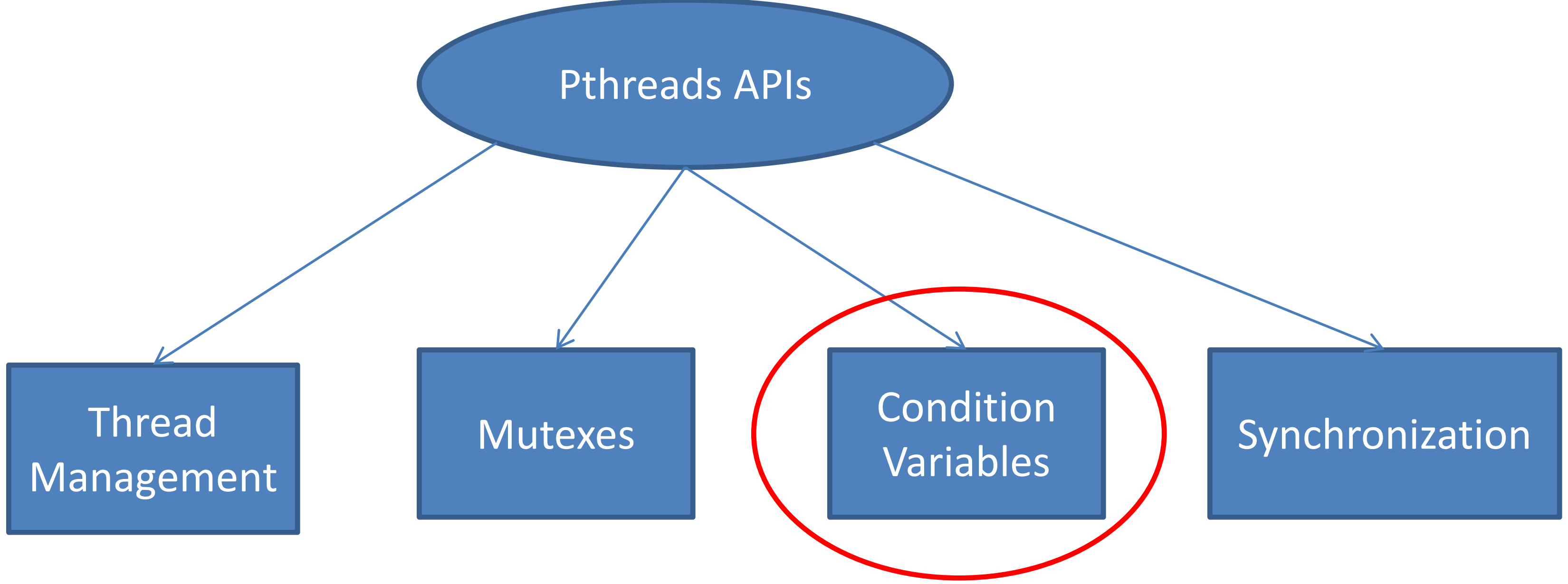
Note: You need to declare mutex p as global variable and initialize it with pthread_mutex_init()



- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- A condition variable is always used in conjunction with a mutex lock.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This is very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling

Steps for Using Condition Variables

- Main thread:
  - Declare and initialize global data/variables which requires synchronization
  - Declare and initialize a condition variable
  - Declare and initialize an associated mutex
  - Create threads A and B to do work
- Thread A
  - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
  - Lock associated mutex and check value of a global variable
  - If the value of the global variable is not what thread A is waiting for then call pthread_cond_wait() : this will do two things:
    - Block thread A and wait for signal from Thread-B.
    - automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
  - When signaled, wake up. Mutex is automatically and atomically locked.
  - Explicitly unlock mutex
  - Continue
- Thread B
  - Do work
  - Lock associated mutex
  - Change the value of the global variable that Thread-A is waiting upon.
  - If the condition that thread A is waiting for is fulfilled then signal Thread-A.
  - Unlock mutex.
  - Continue



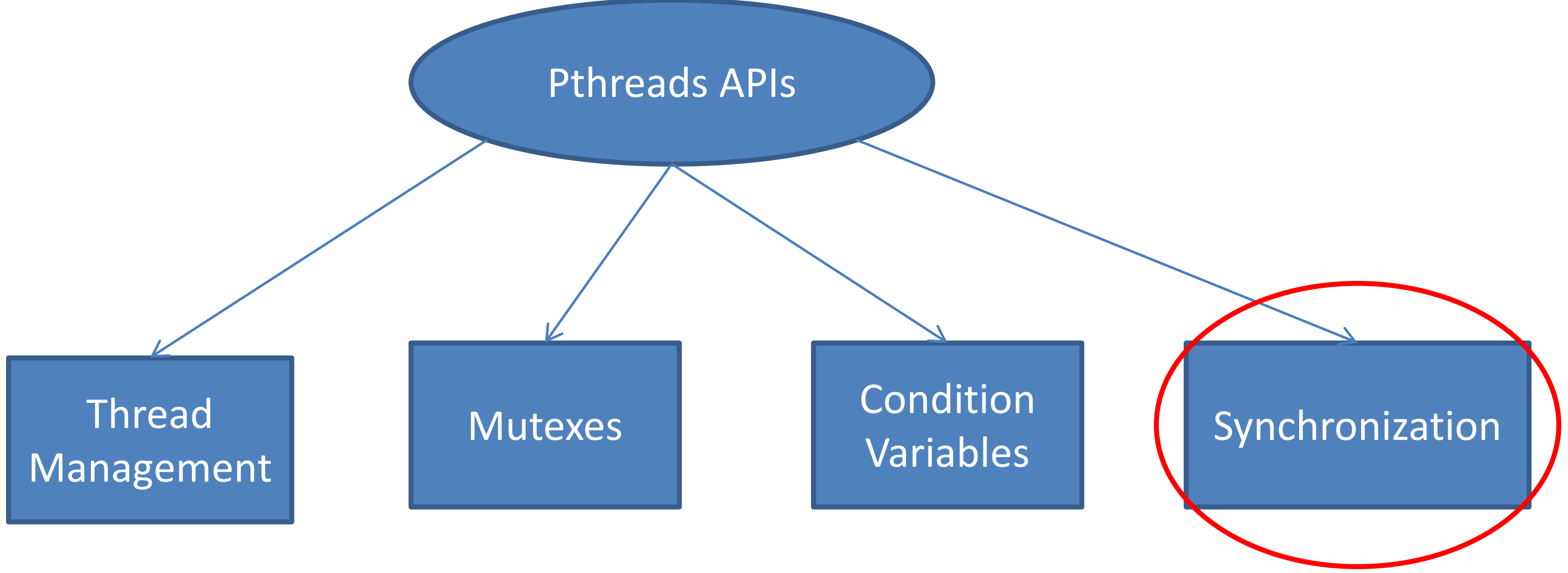
- Condition variables must be declared with type pthread_cond_t
- must be initialized before they can be used.
- There are two ways to initialize a condition variable:
    - Statically, when it is declared. For example:

      pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
    - Dynamically, with the pthread_cond_init() routine.
- pthread_cond_destroy() should be used to free a condition variable that is no longer needed.

```
int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *watch_count(void *t) {
            long my_id = (long)t;
            pthread_mutex_lock(&count_mutex);

          while (count<COUNT_LIMIT) {
                    pthread_cond_wait(&count_threshold_cv, &count_mutex);
                    count += 125;
           }
          pthread_mutex_unlock(&count_mutex);
          pthread_exit(NULL);
}

void *inc_count(void *t) {
      int i;
      long my_id = (long)t;

      for (i=0; i<TCOUNT; i++) {
              pthread_mutex_lock(&count_mutex);
              count++;
              if (count == COUNT_LIMIT)
                        pthread_cond_signal(&count_threshold_cv);
              pthread_mutex_unlock(&count_mutex);

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

block on the condition variable

unblocks threads blocked
for that condition variable



Definition: Synchronization is an enforcing mechanism used to impose constraints on the order of execution of threads, in order to coordinate thread execution and manage shared data.

By now you must have realized that we have 3 synchronization mechanisms:

- Mutexes
- Condition variables
- joins

Semaphores

- permit a limited number of threads to execute a section of the code
- similar to mutexes
- should include the semaphore.h header file
- semaphore functions have sem_ prefixes

Basic Semaphore functions

- creating a semaphore:
  - int sem_init(sem_t *sem, int pshared, unsigned int value)
    - initializes a semaphore object pointed to by sem
    - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
    - gives an initial value value to the semaphore
- terminating a semaphore:
  - int sem_destroy(sem_t *sem)
  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()

Basic Semaphore functions

- int sem_post(sem_t *sem)
  - atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2.
- int sem_wait(sem_t *sem)
  - atomically decreases the value of a semaphore by 1
  - If the value is 0 then the thread will block waiting it to become 1

Semaphores

- Only positive values (or 0)
- Its operations are atomic
- Main usage
  - mutual exclusion
  - synchornization
- Semaphores are discussed here for completeness. But, in practice, do not use them.
  - Internally, it acts like busy waiting
  - Not easy to reason about.

To summarize this part

- Critical region is part of the code where a thread is modifying shared data.
- The case of wo, or more, threads modifying shared data at the same time leads to wrong answers.
- Mutexes, condition variables, etc. are ways to ensuring mutual exclusion (i.e., only one thread is in the critical region at a time).

Question: All that was about threads within a process. How about across processes?

Answer:

- Even across processes there are critical regions, race conditions, and a need for mutual exclusion.

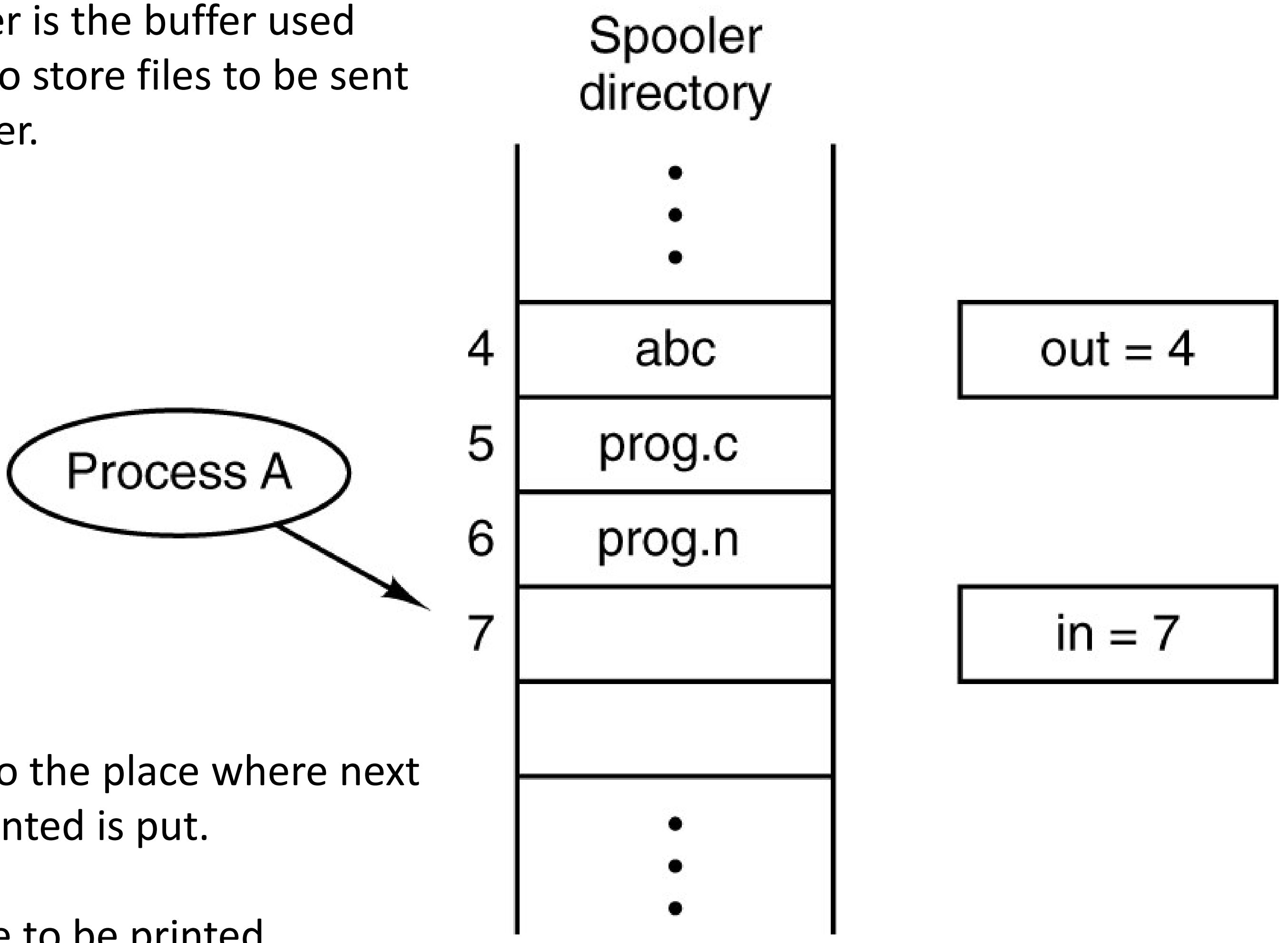
- Technique like mutex can also be used among processes.

Interprocess Communication (IPC)

- Processes may need to communicate with other processes
- Three main issues:
  - How can one process pass information to another?
  - Need to make sure two or more processes do not get in each other's way.
  - Ensure proper sequencing when dependencies exist

Example of IPC

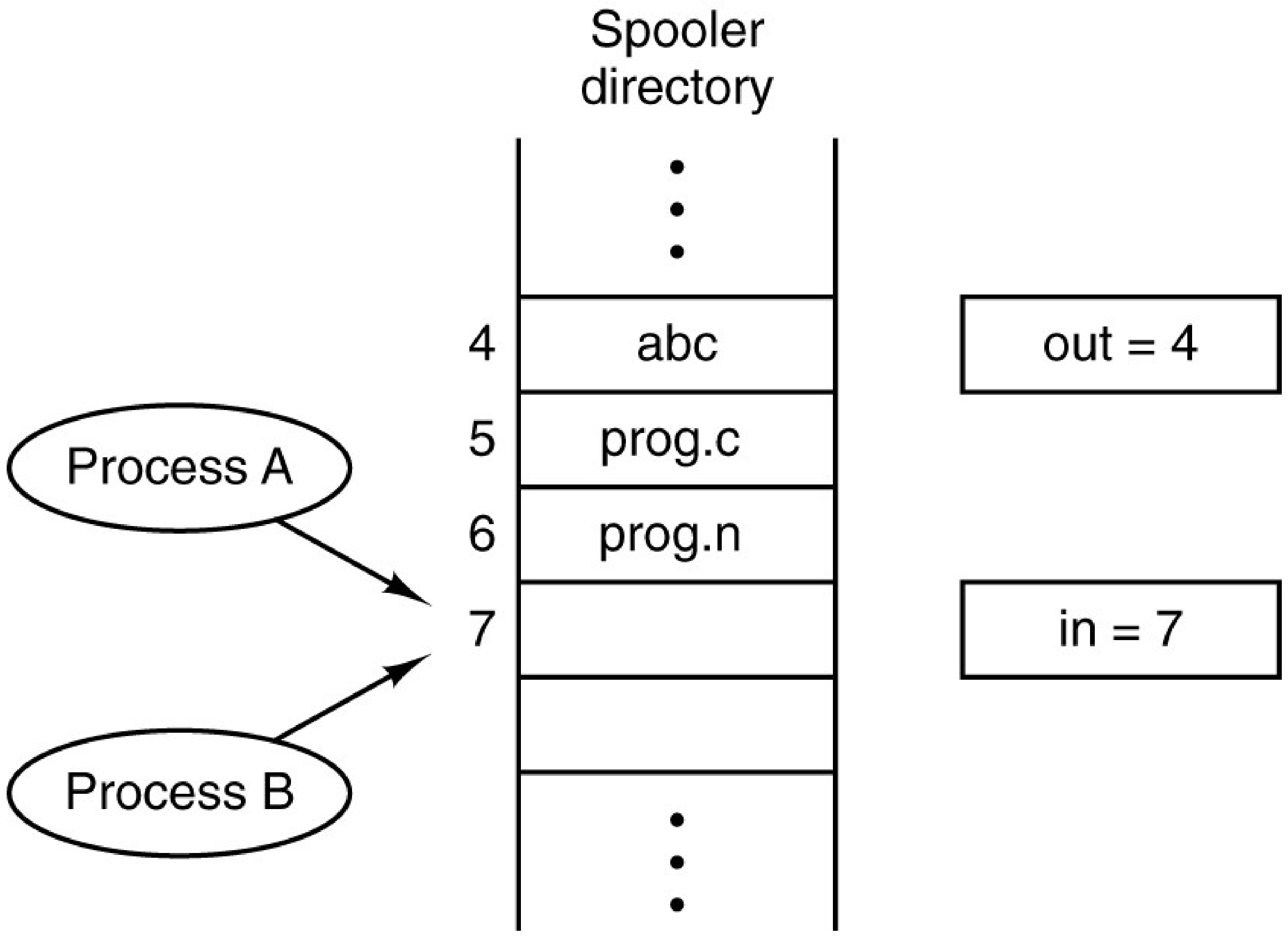
Print spooler is the buffer used by the OS to store files to be sent to the printer.



in: pointer to the place where next file to be printed is put.

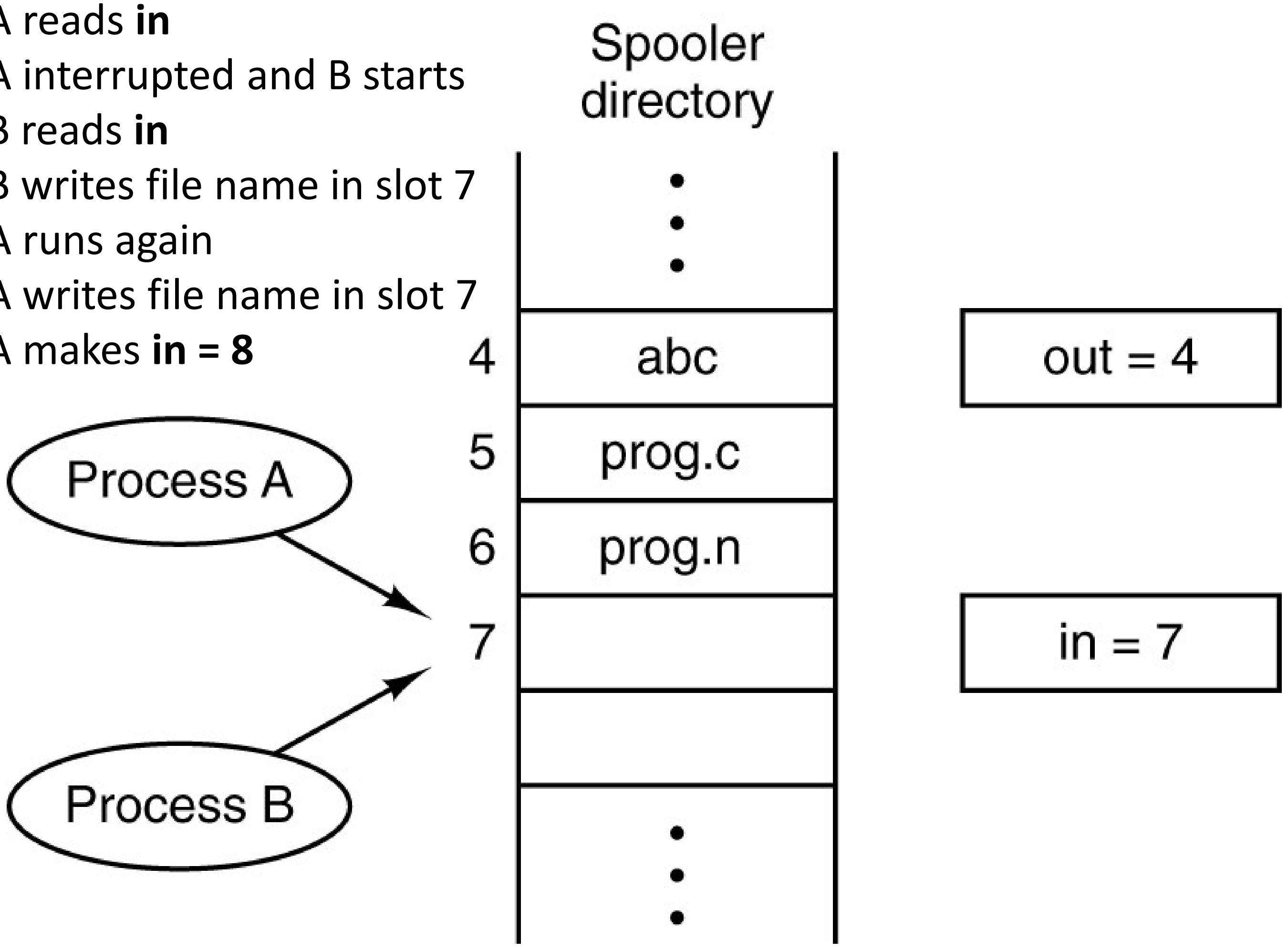
out: next file to be printed.

Example of IPC



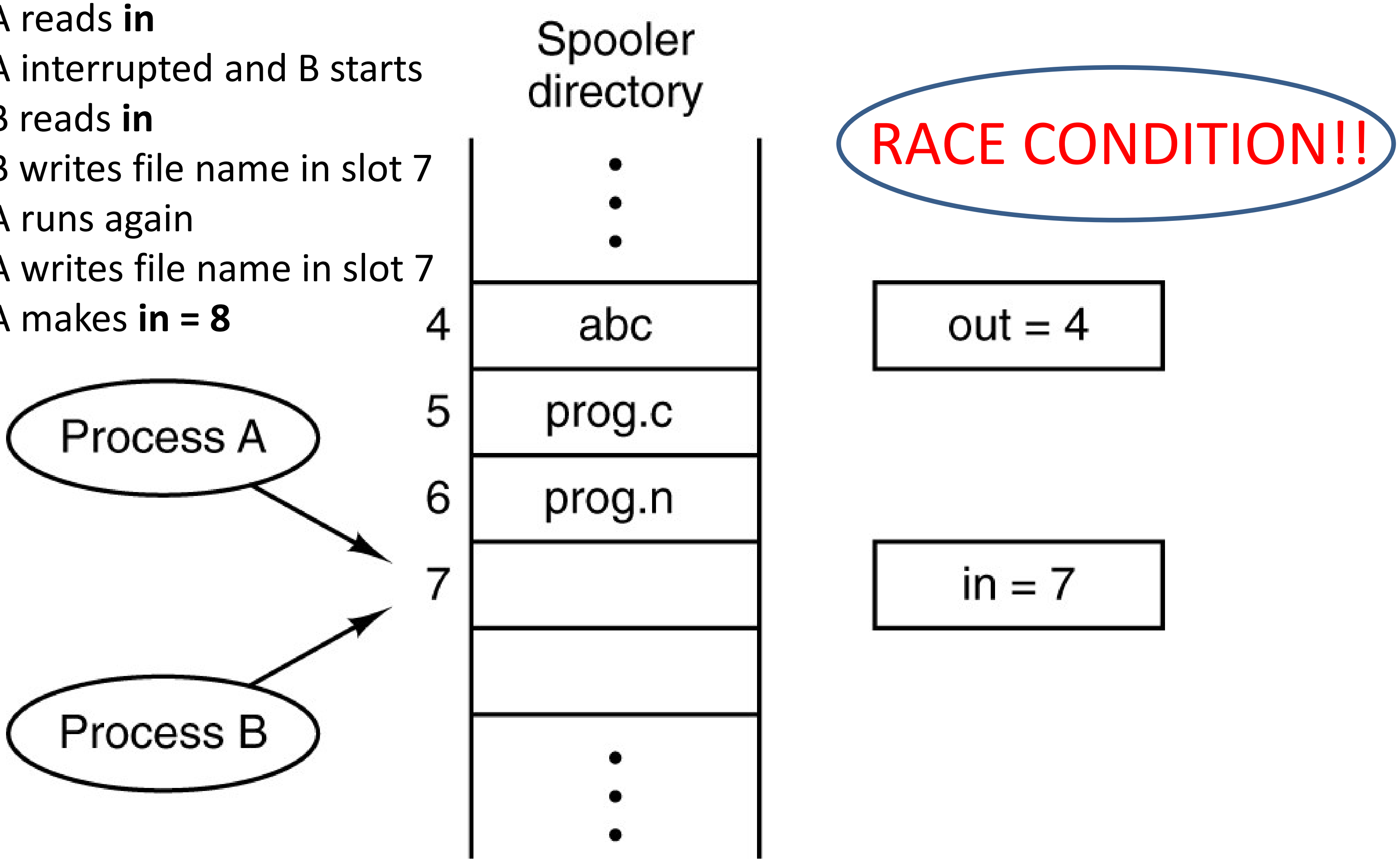
Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**



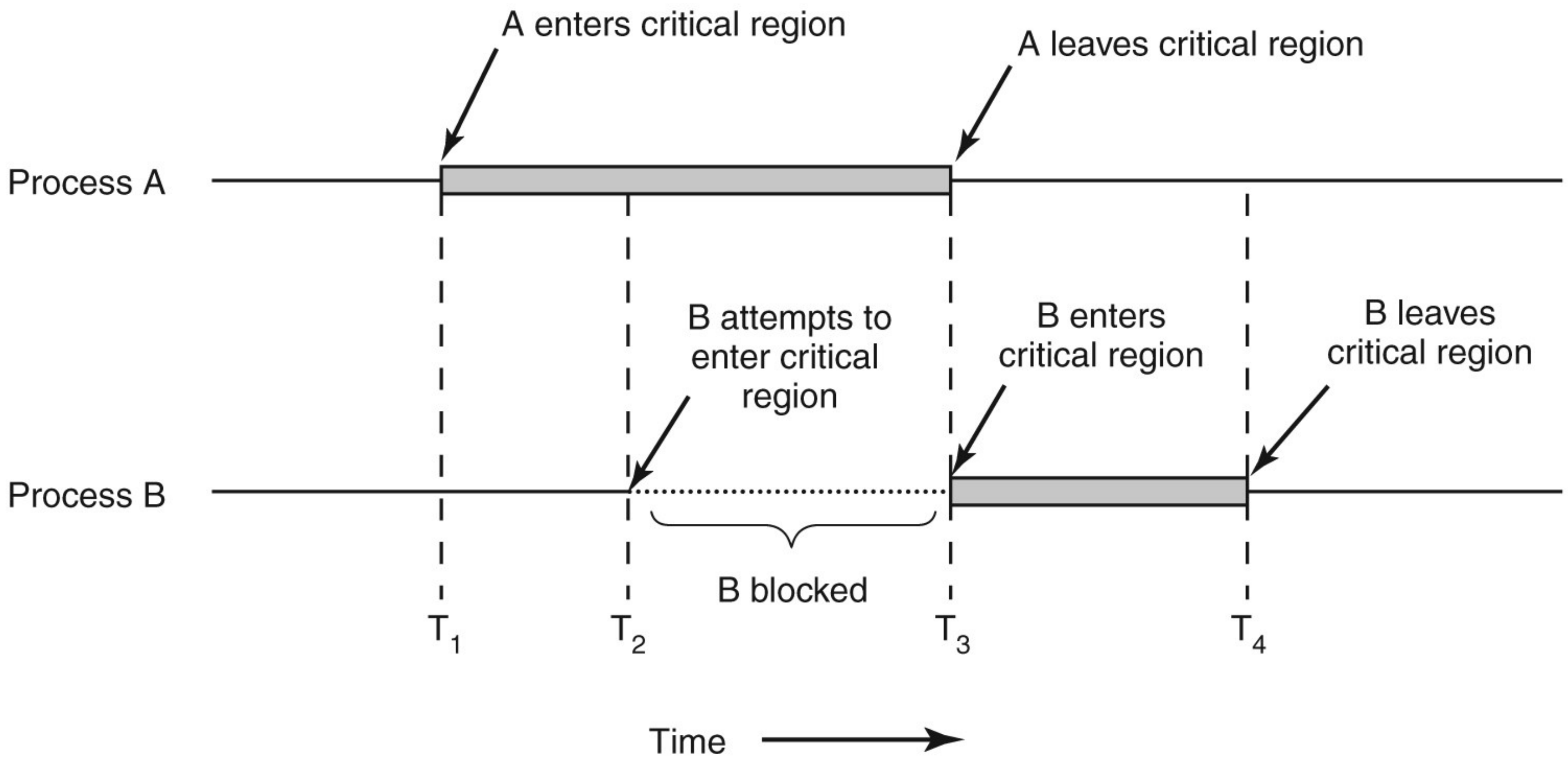
Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**



Conditions of Good Solutions

1. No two processes may be simultaneously inside their critical region.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process has to wait forever to enter its critical region.


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

Solution 1: Disabling Interrupts

Have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.

Solution 1:
Why is it Bad?

- Unwise to give user processes the power to turn off interrupts.
- Affects only one CPU and not other CPUs in the system in case of multicore or multiprocessor systems

Solution 2: Lock Variables

Have a shared (lock) variable, initially set to 0. When a process wants to enter its critical region, it first tests the lock:

- If 0, the process sets it to 1 and enters the critical region
- If 1, process waits until it becomes 0

Solution 2: Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Solution 2:
Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Two processes will be in the critical region at the same time!!

Solution 3: Strict Alternation

```
while (TRUE) {
      while (turn != 0)        /* loop */ ;
      critical_region( );
      turn = 1;
      noncritical_region( );
}
```

```
while (TRUE) {
      while (turn != 1)        /* loop */ ;
      critical_region( );
      turn = 0;
      noncritical_region( );
}
```

Process 0

Variable turn is initially 0

Process 1

Solution 3: Strict Alternation

Busy waiting

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Variable turn is initially 0

Process 0

Process 1

Solution 3: Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
    while (turn != 0)         /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)         /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 0

Process 1

Problematic Scenario:

- Process 1 spends a lot of time here!
- Process 0 finishes its part and sets turn to 1
- Process 1 is stuck in noncritical region and prohibits process 0 from entering the critical region.

Solution 3: Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

Process 0

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 1

Violating condition 3!!

Taking turn is not a good idea when one of the processes is much slower than the other.

Solution 4:
Peterson's Solution

```
     process 0                         process 1

enter_region(0)                   enter_region(1)

Critical Section                  Critical Section

leave_region(0);                  leave_region(1);
```

Solution 4: Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                            /* number of processes */

int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Hardware Solution

- The instruction: TSL RX, LOCK
  - TSL = Test and Set Lock
  - Reads the content of memory word *lock* into register RX, and then stores a nonzero value into *lock*
  - The whole operation is atomic

Hardware Solution

```
enter_region:
        TSL REGISTER,LOCK              | copy lock to register and set lock to 1
        CMP REGISTER,#0                | was lock zero?
        JNE enter_region               | if it was nonzero, lock was set, so loop
        RET                            | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                   | store a 0 in lock
        RET                            | return to caller
```

About Previous Solutions

- Processes must call enter_region and leave_region in the correct timing. If a process cheats, the mutual exclusion will fail.
- The main drawbacks of all these solutions is busy waiting. Keeping the CPU busy doing nothing is not the best thing to do.
  - Wastes CPU time
  - Priority inversion problem (process of higher priority has to wait for a process of lower priority).

Sleep and Wakeup

- Inter-process Communication (IPC) primitives
- Block instead of wasting CPU time
- Two system calls:
  - sleep: causes the caller to block until another process wakes it up
  - wakeup: has one parameter, the process to be awakened

Note: We do not show exact syntax here. But a pseudocode to explain the concept.

First Let's see the: Producer Consumer Problem

- Two processes share a common fixed size buffer.
- One process (producer): puts info into the buffer.
- The other process (consumer): removes info from the buffer.

```
#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 */
        }
}
```

```
#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? */
     }
}
```

What happens if consumer() stopped (i.e. its time slice is done) after reading count (=0) and before calling sleep() ?

Answer: **LOST WAKEUP PROBLEM**

```
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 */
     }
}
```

How to Solve The Lost Wakeup Problem?

- Add a wakeup waiting bit to the picture
  - When a wakeup is sent to a process that is still awake, this bit is set.
  - Later, when the process tries to go to sleep and the bit is set, the bit will be reset but the process will remain awake.
- BUT: What happens when we have more than two processes? How many bits shall we use?

Better Solution for Lost Wakeup Problem: **Semaphores**

- Integer to count the number of wakeups saved for future use
- Two primitives: down and up
    - atomic actions

down: if value = 0 then sleeps

otherwise, decrements it and continue

up: increments the value, and wakes up a sleeping process (if any)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Mutexes with Processes

- A variable that can be in one of two states: locked and unlocked
- Can be used to manage critical sections
- Managed using TSL

```
mutex_lock:
          TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
          CMP REGISTER,#0             | was mutex zero?
          JZE ok                      | if it was zero, mutex was unlocked, so return
          CALL thread_yield           | mutex is busy; schedule another thread
          JMP mutex_lock              | try again
ok:       RET                         | return to caller; critical region entered


mutex_unlock:
          MOVE MUTEX,#0               | store a 0 in mutex
          RET                         | return to caller
```

Didn't We Say Processes Do Not Share Address Space?

- Some of the shared data structures can be stored in the kernel and accessed through system calls.
- Most modern OSes offer ways for processes to share some portions of their address spaces with other processes

Conclusions

- We saw some primitives to deal with critical sections.
- These primitives are provided by OS to ensure mutual exclusive access to critical regions.
- We saw techniques for threads and processes.