CS 492: Operating Systems

Inter Process Communication (I)

Race Condition

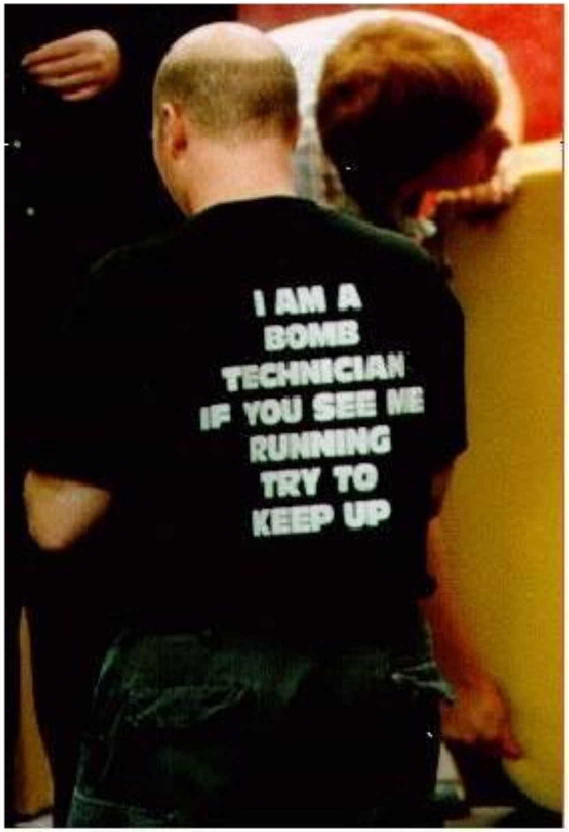
Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Logistics

- CAs office hours: Canvas, North Building 101
- Teams of 2. Submit by next Wed, Feb 7th.
- Assignment of teams to CA (coordination)
- First Individual Homework will be posted today.

(IPC) Race Condition



(One Way to Interpret Race Condition)

Goals for Today

- Inter-Process Communication (IPC)
  - Race conditions
  - Critical section
  - Mutual exclusion
    - Mechanisms for mutual exclusion

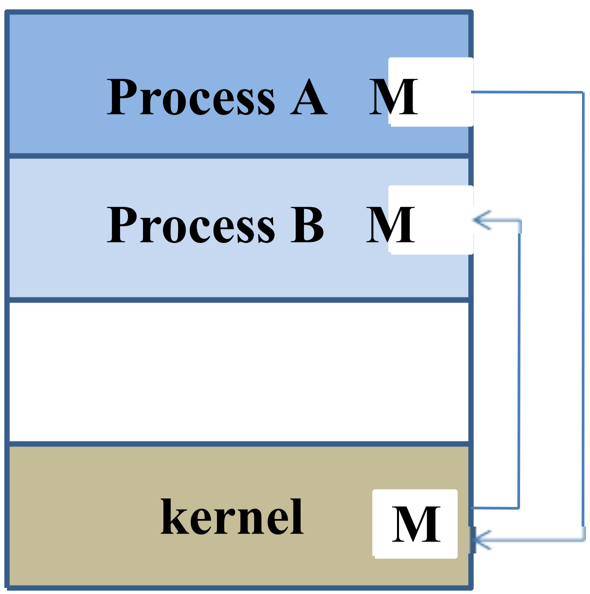
Why Inter-process Communication (IPC)?

- Processes may need to share data
  - Exchange data between multiple processes
  - Processes do not get in each other’s way
  - Maintain proper sequencing of actions in multiple processes

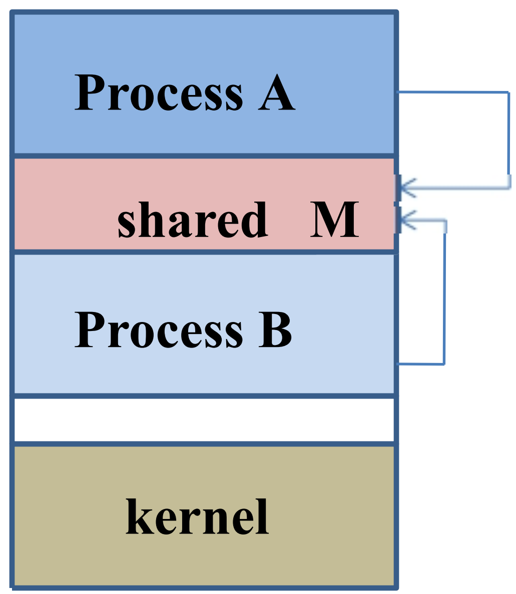
- Do these issues apply to threads as well?

Three IPC Approaches

1. By passing messages through the kernel
2. By sharing a region of physical memory
3. Through asynchronous signals or alerts

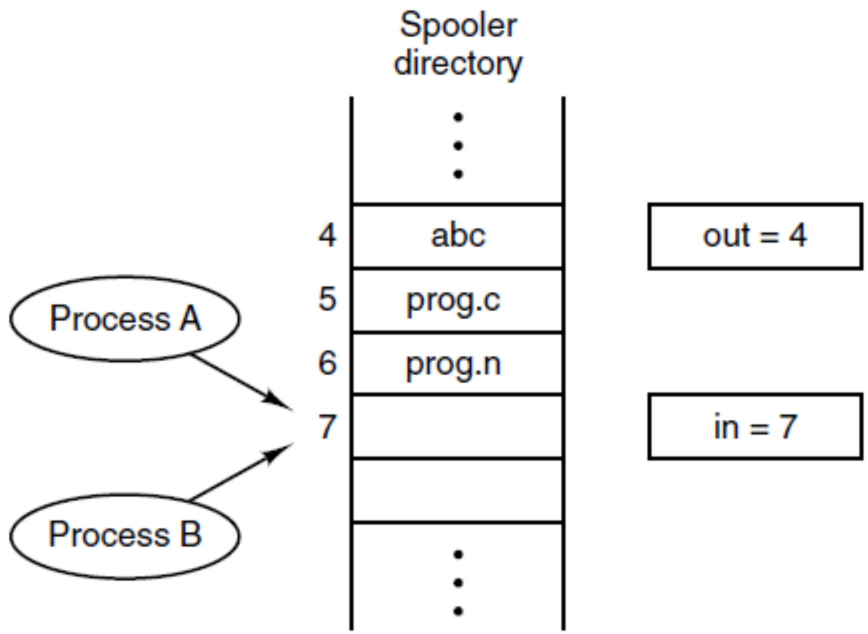


Approach 1



Approach 2

Spooler Directory



Two processes want to access
shared memory at the same time.

Another IPC Example

- Suppose we implement a function to withdraw money from a bank account

```
int withdraw(account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your mother share a bank account with a balance of $1500.00
- What happens if you and your mother both go to separate ATM machines, and simultaneously withdraw $100.00 from the account?

Example Continued

- We represent the situation by creating a separate thread for each ATM user doing a withdrawal
  - Both threads run on the same bank server system

Thread 1

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread 2

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

What's the problem with this?

- What are the possible balance values after each thread runs?

Interleaved Execution

- Assume each thread can context switch after each instruction

Execution sequence as seen by CPU

```
balance = get_balance(account);
balance -= amount;
```

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

context switch

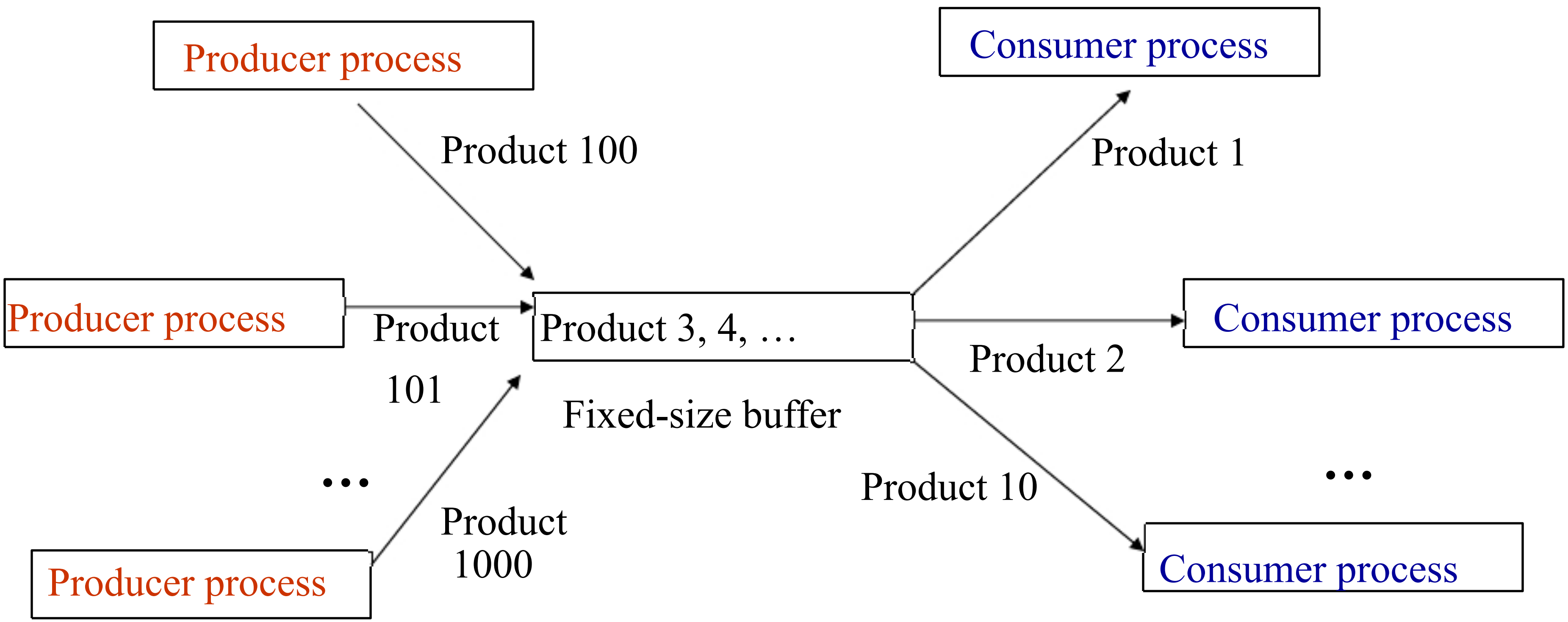
```
put_balance(account, balance);
```

What’s the account balance after this sequence?

More examples of IPC

- Producer-consumer problem
  - Fixed-size buffer *B*
  - *m* producer processes
  - *n* consumer processes
  - Producer and consumer processes share the buffer B

Example of Inter-process Communication: Producer-consumer Problem



Example of Inter-process Communication: Producer-consumer Problem (Cont.)

Shared variable:

```
  const int n;
  int  count=0;
 Item buffer [n];
```

Producer

```
    while (1){
          …
          produce an  item A;
          …
          while(count==n);
          insert(item);
          count++;
}
```

Consumer

```
    while (1){
          while(count==0);
          item=remove_item();
          count--;
          …
          consume an item;
}
```

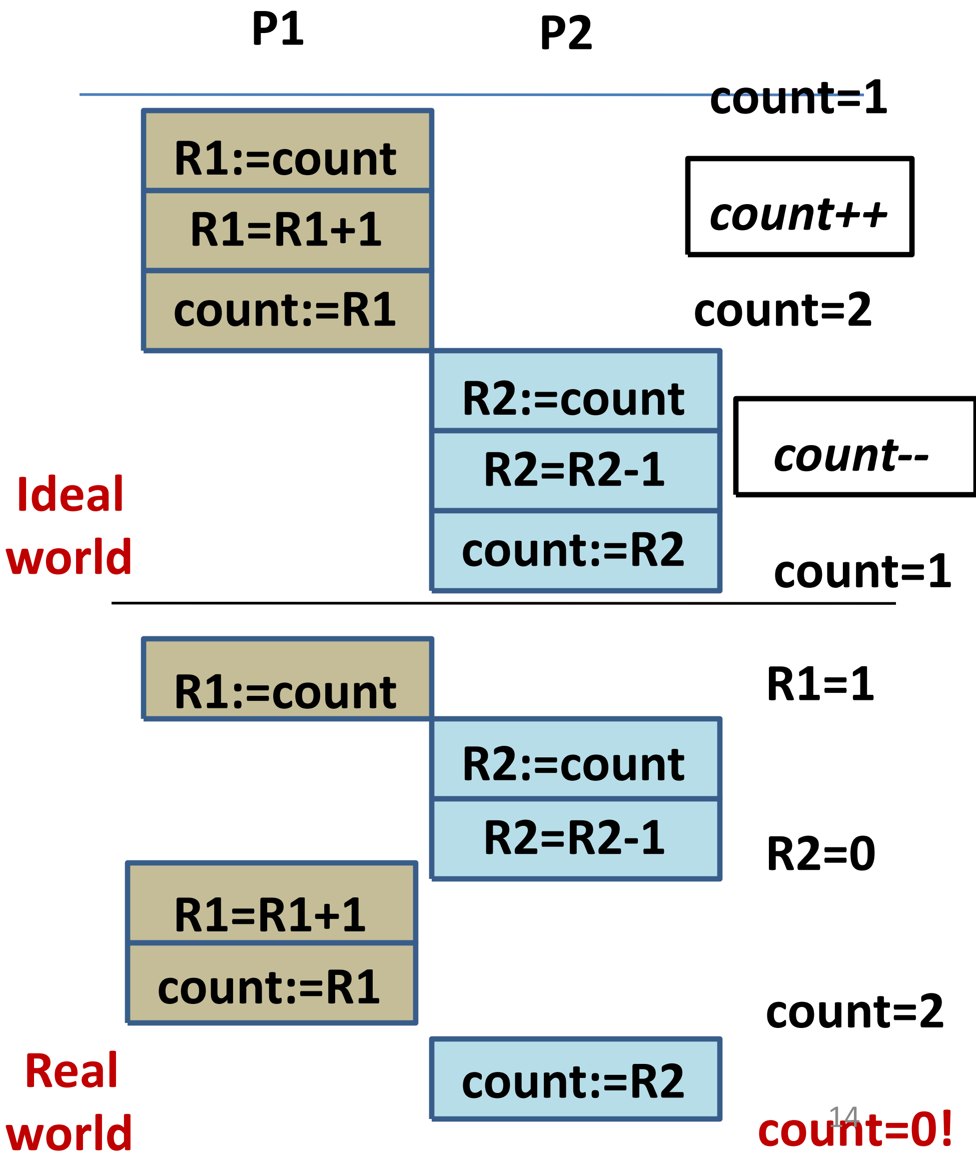
Possible Chaos

- count++/count-- instruction is not an atomic operation.
- count++:

  R1: = count;

  R1 = R1+1;

  count:=R1;



Possible Outcome

- The result of the concurrent access is non-deterministic
- Result depends on:
  - Timing
  - When context switches occurred
  - Which process/thread ran at context switch
  - What the processes/threads were doing

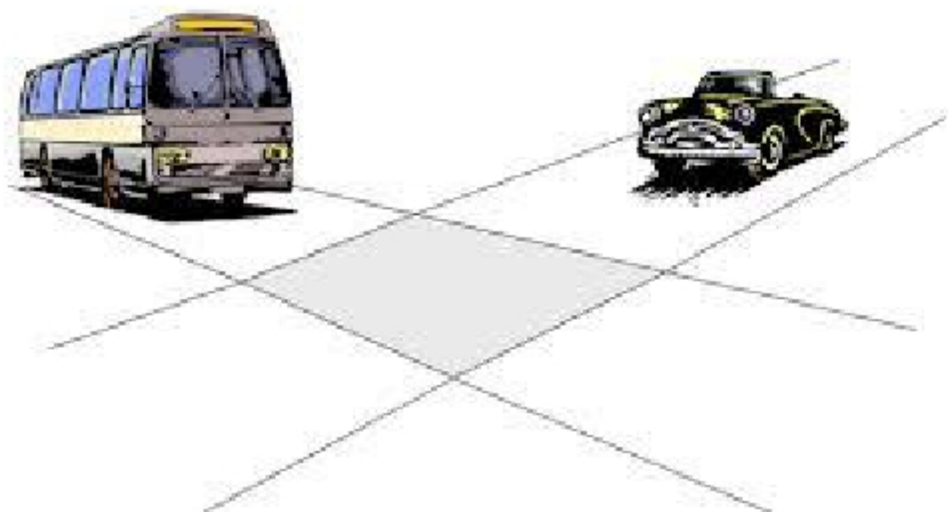
Race Conditions

- **Race condition**
  - Two or more processes are reading or writing the shared data
  - The final result depends on the order of the processes are executed

- Need to eliminate race conditions!

Critical Regions

- *Critical region or section* – Part of program where the shared memory is accessed
  - Read/write of the data in critical section may lead to errors and race conditions



Critical Regions (2)

Requirements to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
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 should have to wait forever to enter its critical region.

Critical Region in Producer-consumer Problem

```
Shared variable:
   const int n;
   int  count=0;
  Item buffer [n];
```

What is the critical section of the the producer/consumer problem?

```
Producer
      while (1){
              …
              produce an  item A;
              …
              while(count==n);
              insert(item);
               count++;
}
```

```
Consumer
      while (1){
              while(count==0);
              item = remove(item);
               count--;
              …
              consume an item;
}
```

The Critical-Region Problem

- N processes are competing to use shared data.
  - Structure of process P_i ----Each process has a code segment, called the **critical section**, in which the shared data is accessed.

repeat

entry section /* enter critical section */

critical section /* access shared variables */

exit section /* leave critical section */

remainder section /* do other work */

until false

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

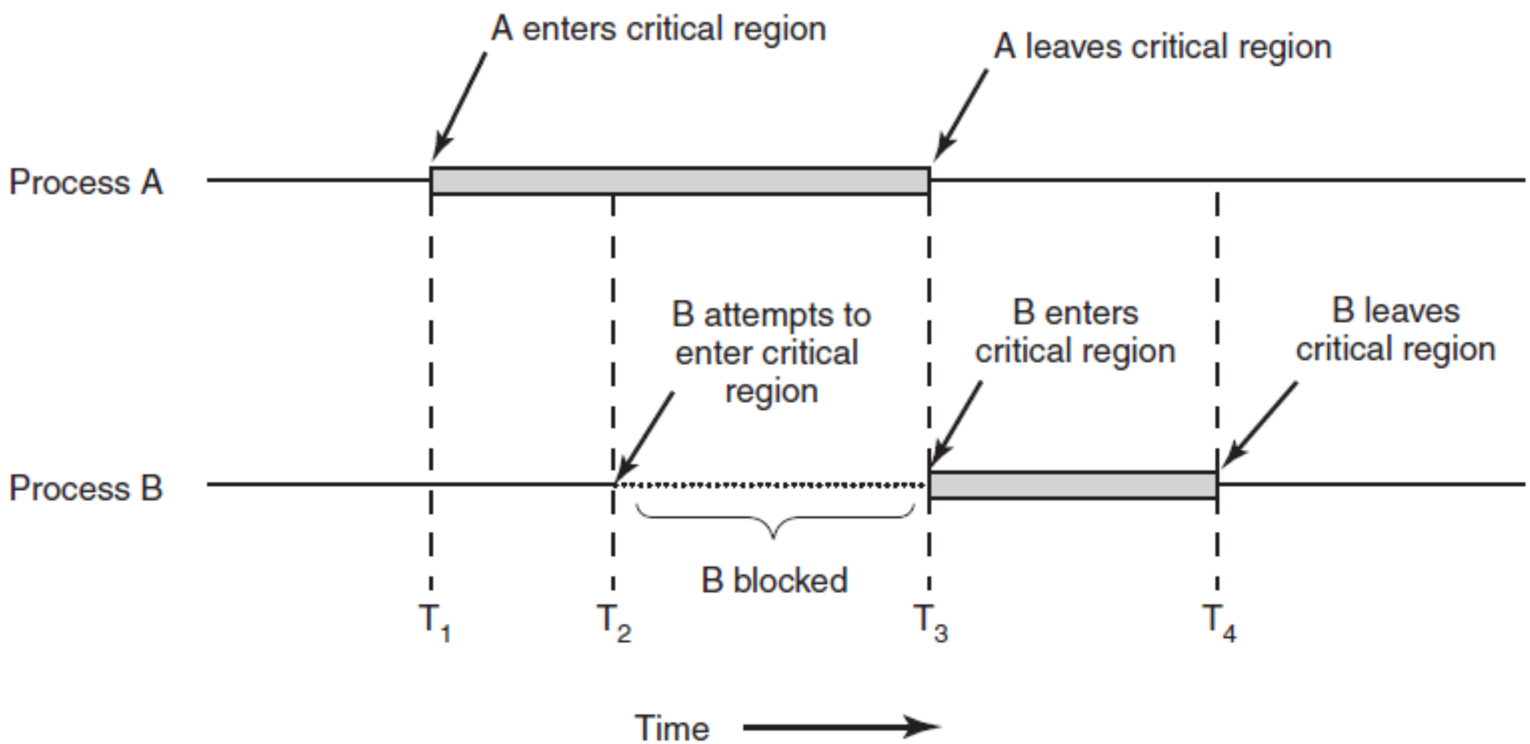
Desired Solution

- **Mutual Exclusion**: Only one thread can be in critical section at a time

- 4 properties of mutual exclusion within critical section:
  1) No 2 processes may be simultaneously inside their critical sections (*correctness*)
  2) No process should wait indefinitely long to enter its critical section (*liveness* – freedom from *starvation*)
  3) No process stopped outside its critical section should block other processes
  4) No assumptions about relative speeds of processes or numbers of processes

No Mutual Exclusion?

Critical Regions/Mutual Exclusion



Mutual exclusion using critical regions.

Mechanisms for Mutual Exclusion

1) Software solution
   – Strict alternation
   – Peterson’s solution

2) Hardware solution
   – Interrupt disabling
   – Test-and-Set lock (TSL)

3) Higher level solutions (mutex, semaphores…)

Today’s Task

1) Software solution
   – Strict alternation
   – Peterson’s solution

2) Hardware solution
   – Interrupt disabling
   – Test-and-Set lock (TSL)

3) Higher level solutions (mutex, semaphores…)
   – Next lecture

Software Mechanism 1: Strict Alternation

- Use a shared variable ***Turn*** to strictly alternate between processes
- Waiting process continually reads the variable to see if it can proceed
  - Called **spin-lock** – lock wherein a process **busy waits**

Spin lock

Turn=0: process 0's turn to execute

```
void process0 (void *ignored) {
    while(true){
        while(turn!=0); //loop
        critical_region();
        turn = 1;
        noncritical_region(); }}
```

Turn=1: process 1's turn to execute

```
void process1 (void *ignored) {
    while(true){
        while(turn!=1); //loop
        critical_region();
        turn = 0;
        noncritical_region();}}
```

Strict Alternation: More Discussion

- Disadvantages
  - Horribly wasteful!
    - Processes waiting to acquire locks spin on the CPU
  - If Process 0 is faster than Process 1, then Process 0 will constantly get blocked by waiting for Process 1 to finish critical section
  - Only allow two processes
    - How to change to n-process model?
  - Not a serious candidate since, it violates condition 3 of Critical Regions

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 */
}
```