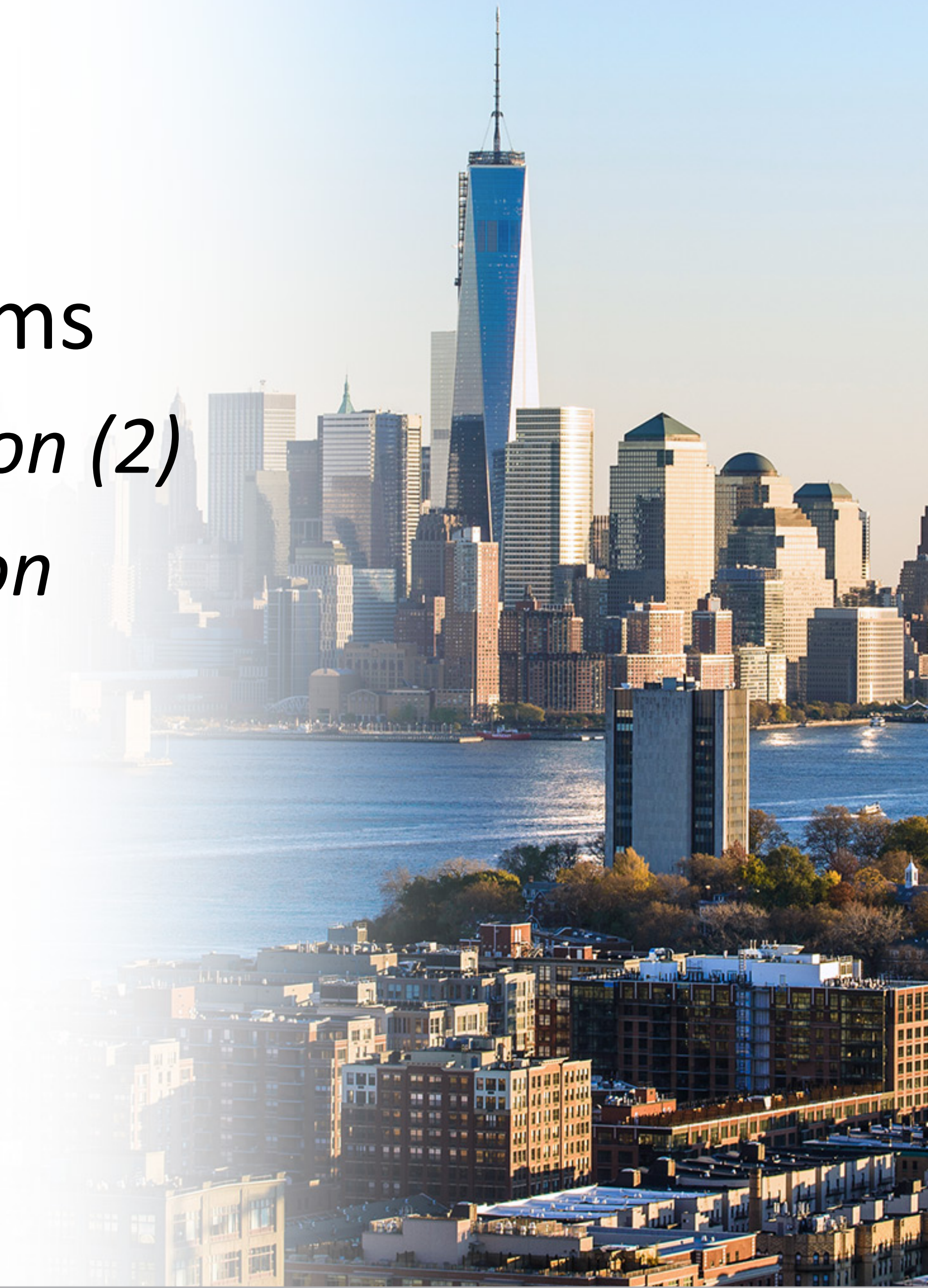
CS 492: Operating Systems

Inter Process Communication (2)

Software/Hardware Solution

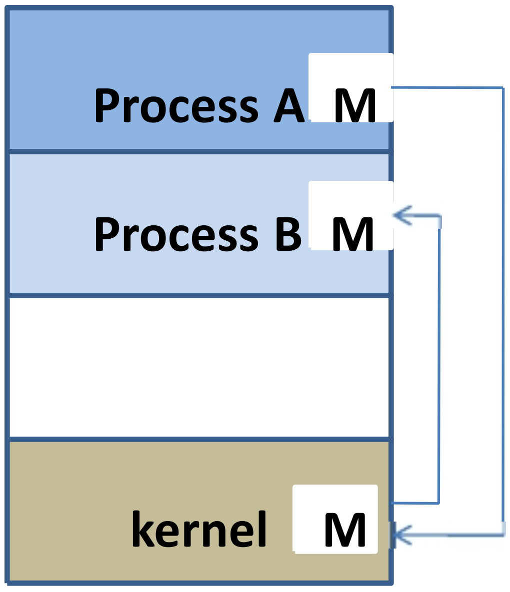


Instructor: Iraklis Tsekourakis

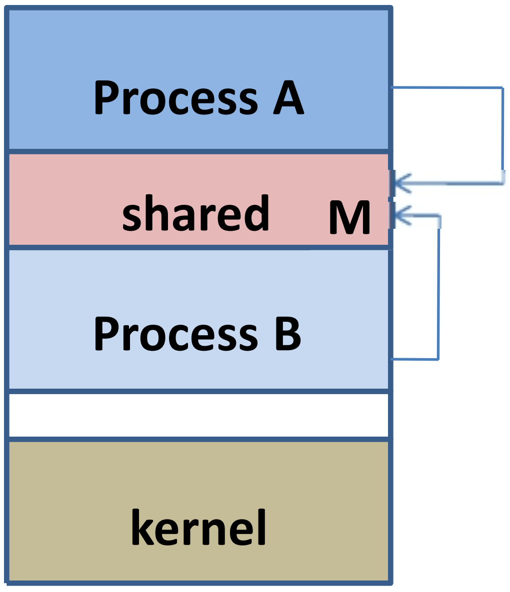
Email: itsekour@stevens.edu

Recap: Inter-process Communication

- How can processes interact in real time?
  - (a) By passing messages through the kernel
  - (b) By sharing a region of physical memory
  - (c) Through asynchronous signals or alerts



(a)



(b)

Recap

- Sharing a region of physical memory
  - Race conditions
    - Cooperating processes read and write the shared memory
  - Critical section
    - Section of code where the shared memory is accessed
  - Mutual exclusion
    - Only one thread can be in critical section at a time

Mechanisms for Mutual Exclusion

1) Software solution
   1.1) Strict alternation
   1.2) Peterson’s solution

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

3) Higher level solutions

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!
  - Only allows two processes (extensions to more than 2 processes are expensive)
  - Fast process will get blocked by the slow process

Software Mechanism 2: Peterson's Solution

- Assume two threads, P0 and P1.
- Variables:
  - int turn – whose turn to enter critical section
  - bool interested[2]
    - interested[i]=true means Pi is ready to enter critical section

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 */
}
```

Peterson’s solution: More Discussion

- Mutual exclusion
  - Q1: Why Peterson’s solution guarantees mutual exclusion?
  - Q2: Why Peterson’s solution is more efficient than the strict alternation?

```
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 */ ;
}
```

- Still only works for 2 processes
  - Can generalize to n processes, but for some fixed n

Mechanisms for Mutual Exclusion

1) Software solution
   1.1) Strict alternation
   1.2) Peterson's solution

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

3) Higher level solutions

Hardware Mechanism 1: Disabling Interrupts

- Idea:
  - Right before entering the critical section: the process disables system interrupts;
  - Before leaving the critical section: the process re-enables the interrupts;
  - Effect: CPU cannot be switched to other processes when a process is in the critical section.

- Disadvantages??

Hardware Mechanism 1: Disabling Interrupts (2)

- On the other hand, it could be convenient for the kernel to disable interrupts for a few instructions
  - Update variables or lists (e.g. ready processes list)
- Actually, not anymore

Hardware Mechanism 2: Locking Test-and-Set Lock (TSL)

- Test and modify the content of a word atomically
  - The word here is the lock.
  - **TSL REGISTER,LOCK**

TestAndSet function:

- Return the current value of memory word *lock, then set lock* to be true.
- Atomic operation (the code cannot be interrupted during execution)

Hardware Mechanism 2: The TSL Instruction (1)

```
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
```

Entering and leaving a critical region using the TSL instruction.

Hardware Mechanism 2: Locking Test-and-Set Lock (TSL)

Implementation

Enter_region:

//If the lock is set, wait till it is reset by other processes

//Else enter the critical section and set the lock

Critical section;

Leave_region:

//reset the lock

Hardware Mechanism 2: The TSL Instruction (2)

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        CMP REGISTER,#0             | was lock zero?
        JNE enter_region            | if it was non zero, 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
```

Entering and leaving a critical region
using the XCHG instruction

Busy Waiting..

- Disabling interrupts, using TSL, strict alternation, and Peterson’s algorithm
  - All -> busy waiting.
  - Not only wasteful, but can also have unexpected effects!! **The priority Inversion Problem**
- Can we do better?

Mechanisms for Mutual Exclusion

1) Software solution
   1.1) Strict alternation
   1.2) Peterson's solution
   1.3) Sleep and wakeup

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

3) Higher level solutions

Sleep and Wakeup

The producer-consumer problem (1)

```
#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)
{
```

Sleep and Wakeup
The producer-consumer problem (2)

```
            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….

```
#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 */
        }
}
```

Any race conditions here? Any cases where it fails?