Interprocess Communication

CS 342 – Operating Systems

Ibrahim Korpeoglu
Bilkent University
Computer Engineering Department

Interprocess Communication

- Frequently, processes need to communicate with each other.
- There are three issues to consider in providing interprocess communication
  - Providing an environment so that processes can exchange messages or data
  - Making sure that no processes is getting in the way of some other process
    - Each process trying to get the last chunk of resource.
  - Providing proper sequencing. If B is dependent on the output of A, then B has to wait until A produces the output.

Interprocess Communication

- These issues apply equally well to the threads
  - But the first issue is trivial for threads because of the global variables that make it easy to share information.
  - The second and thirds issues need to be addresses for threads also.
  - Solutions that are developed for second and thirds issues can be applied to processes and threads.

Race Conditions

- These conditions occur, when two or more processes shared a common storage
- The storage could be in memory in disk, does not matter.
- When both processes read and write to the shared storage, then some inconsistencies may occur
  - The inconsistency will depend on who runs precisely when.
- The conditions are called race conditions.

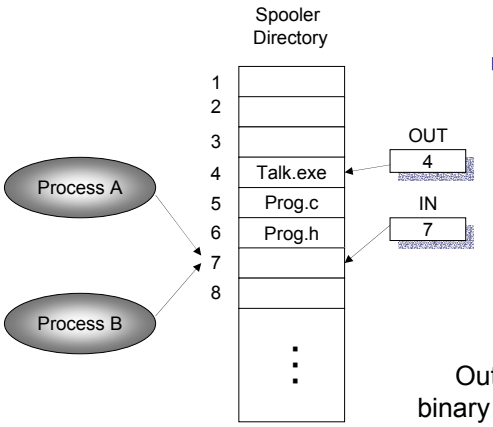
Race Conditions – An example

- Print spooler and structures may cause race conditions, if not implemented properly.
- Print spooler is a background process (daemon) that takes files from a well-defined directory (spooler directory) and prints them one-at-a-time to the printer.
- Any process that want to print a file does not print the file directly to the printer, but
  - The process writes the file (copies the file) to the spooler directory.
- At any point in time, the spooler directory may contain zero or more file that may belong to different processes and that wait for printer.

Race Conditions – An example

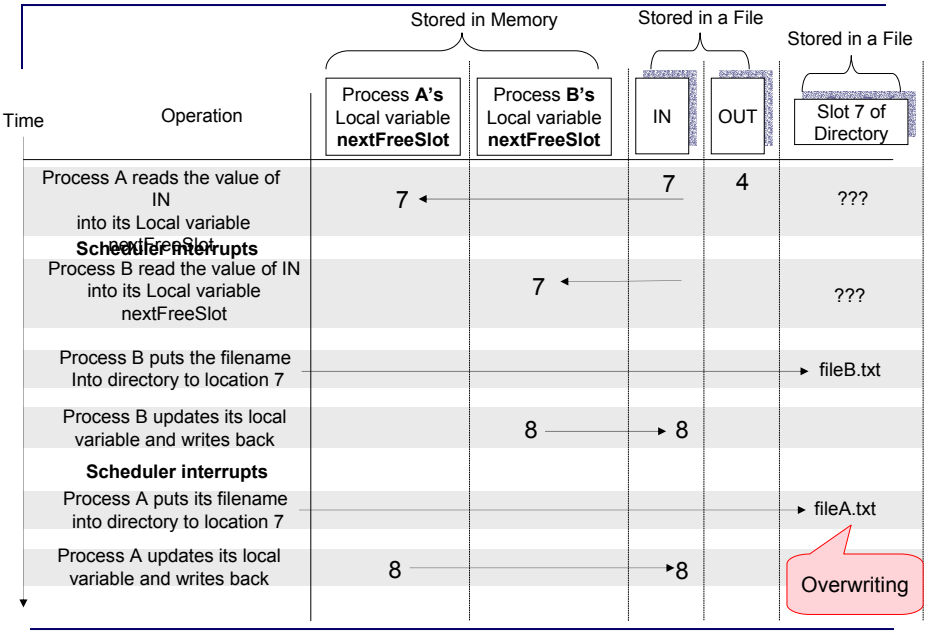
- Print spooler daemon implementation:
  - Assume the spooler daemon stores the name of the files that are to be printed in a directory that has a finite amount of slots (one slot per file to store its name). This directory could be a binary file in disk.
  - The spooler daemon also keeps two variables (in and out)
    - **In** points to the first empty slot where a new filename can be entered.
    - **Out** points to the first non-empty slot, from where the spooler will read a filename and print the corresponding file.

Spooler Implementation



- Assume both process A and B want to print some file.
  - They want to enter the filename into the first empty slot in spooler directory

Out and In could be stored in a binary file that consist of two integers.



Race condition

- The previous slide gave an example of race condition.
- Process B's filename that has been stored slot 7 has been overwritten by process A's filename
- Process B's file will never be printed by the spooler.

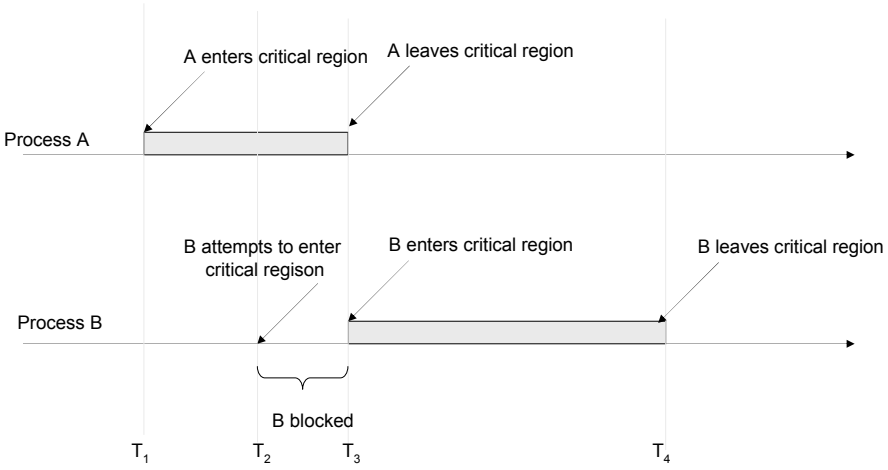
Avoiding Race conditions

- Definition: *Critical section of code*: the part that accesses the shared storage.
- Method to avoid race conditions:
  - Prevent to enter more than one process to their critical sections.
- Prevention will be done by use of some OS primitives.

Conditions to met

- 4 conditions are necessary to hold to have a good solution
  - No two processes may be smultaneously enter their critical regions (section)
  - No assumptions may be made about speeds or the number of CPUs.
  - No process running outside of its critical region may block some other process
  - No process should have to wait forever to enter its critical region

Mutual exclution using critical regions



Mutual Exclusion Methods – Initial Solutions

- Methods that are not good or not enough to solve the problem
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
- Peterson's solution
  - Method that works but inefficient in terms of CPU usage
- TSL instruction
  - Hardware solution
  - Still requires busy waiting

Mutual Exclusion Methods – More efficient solutions

- Sleep and Wakeup
- Semaphores
  - A solution that is widely used
- Mutexes
  - Like semaphores but simpler
- Monitors
- Message Passing
- Barriers

Disabling Interrupts

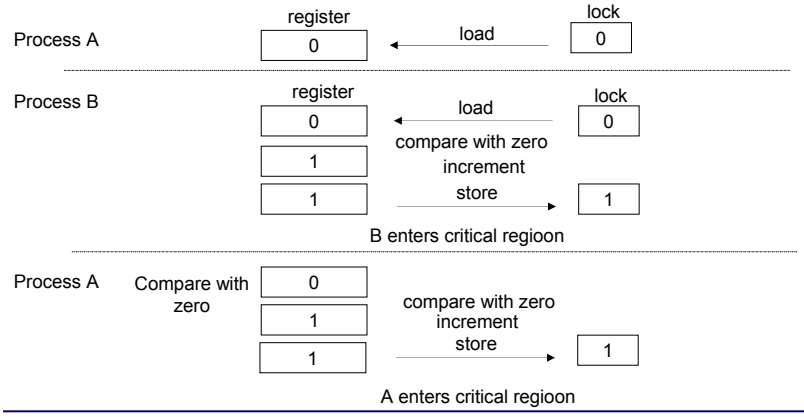
- Method
  - A process that wants to enter its critical region will disable all the interruts
  - Since all interrupts are disables, clock interrup will not be received.
  - Hence, process will not be interrupt and the CPU will not be given to scheduler.
  - Hence, some other process can not run while a process is in its critical region.
  - A process enables interrupts when it leaves its critical region.
- What happend if a process never enables interrupts: end of system
- It is not wise to give user processes the power to disable interrupts.
- Kernel can diable interrupts for a short amount of time while it is updating variables or lists. This is OK and many kernels do that.

Lock Variables

- Software solution
- Have a single shared variable *lock*.
  - İnitially set to zero.
- When a process wants to enter its critical seciton:
  - It loads (reads) the lock variable value from memory to register
  - It compares the value with 0
  - It sets the value to 1 in register
  - It stores (writes) the register value to memory
  - It enters its critical section
- When a process leaves its critical section
  - It sets the value of lock to 0 again.

Lock Variables

- Does not solve the problem.
- Accessing the shared variable lock causes now problem. It is not mutually excusively accessed.



Strict Alternation

Process 0

```
While (TRUE)
{
        while (turn != 0);
        critical_region();
        turn = 1;
        noncritical_region();
}
```

Process 1

```
While (TRUE)
{
        while (turn != 1);
        critical_region();
        turn = =;
        noncritical_region();
}
```

- Process 0 and 1 take strict turns in accessing the critical region

Peterson's Solution

process i (i in {0,1})

```
#define  N    2

int turn;
int interested[N];

void enter_region(int process)
{
        int other;

        other = 1 – process;
        interested[process] = TRUE;
        turn = process;
        while (turn == process && interested[other]=TRUE)
}

void leave_region(int process)
{
        interested[process] = FALSE;
}
```


Peterson's Solution

Process 0

```
#define  N    2
int turn;  /* should be shared */
int interested[N];

void main()
{

        while(1)
        {
                noncritical_region();

                enter_region(0):
                critical_region();
                leave_region(0);

                noncritical_region();
        }
}
```

Process 1

```
#define  N    2
int turn; /* should be shared */
int interested[N];

void main()
{

        while(1)
        {
                noncritical_region();

                enter_region(1):
                critical_region();
                leave_region(1);

                noncritical_region();
        }
}
```

TSL Instruction Solution

```
TSL REGISTER,  LOCK
```

```
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 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
```


Sleep and Wakeup

- Peterson's solution and TSL solution both are correct, but inefficient
  - They use busy waiting
  - Waste of CPU cycles
  - Causes also Priority Inversion Problem
    - Assume 2 processes, one with High priority (H) and one with Low priority (L).
    - Assume L is in critical region, and H is in busy waiting for L to exit its critical regison.
    - But scheduler will never schedule L, since H has higher priority. H will loop forever.

Sleep and Wakeup

- Lets look to some other methods that blocks the process when the process want to enter its critical region but could not do so since some other process is in its critical region.
- The processes will use two special system calls: sleep and wakeup.
- A process that wants to block will call sleep system call
- A process that want to wake-up a sleeping process will call wakeup system call.

Example problem that these primitives can be used

- Producer-consumer problem.
  - Also called bounded buffer problem.
- There is shared buffer between a producer process and consumer process
  - Producer puts items into the buffer when they get produced.
    - Producer has to sleep when there is no room in the buffer for a new item (buffer full)
    - Producer has to wake up consumer, when producer puts a new item into an *empty* buffer.
  - Consumer takes the items from the buffer when it wants to consume.
    - Consumer has to sleep if there is not item available in the buffer (buffer empty)
    - Consumer has to wake up producer, when the consumer takes out an item from a *full* buffer.

```
#define  N    100
int count = 0;

void producer() {
        int item;

        while (TRUE)
        {
                item = produce_item();  /*insert the item into the buffer*/
                if (count == N)
                        sleep();
                insert_item(item);
                count = count + 1;
                if (count == 1)
                        wakeup(consumer);
        }
}
```



```
void consumer() {
        int item;

        while (TRUE)
        {
                if (count == 0)
                        sleep();
                item = remove_item();  /*remove the item from buffer */
                count = count – 1;
                if (count == N-1)
                        wakeup(producer);
                consume_item(item);
        }
}
```

Problem with the solution

- A wakeup sent to a process can be lost if the process is not sleeping yet.
  - Then the process will sleep and it will never wakeup since no other wake-up signals will sent again.
- The solution is to store the wakeup signals.
- Semaphores is actullay achieving this goal.

Semaphores

- Dijkstra proposed to use integer variable to count the number of wakeups that are saved for future use.
  - A new varible called semaphore is introduced.
  - Operations on these kind of variables are defined.
    - Two operations: Up and Down.
  - The operations are implemented by OS and they are guaranteed by OS to be atomic.

Semaphores

- Down (semephore) operation
  - A process has to call this before it enters its critical region.
  - If value of semaphore is zero, process blocks (waits). Process can not return from down() call.
  - If value of semaphore is not zero, value of semaphore is decremented and process return from down() call. Can enter its critical region.
- Up (semaphore) operation
  - A process calls up when it is finished with critical section or when it want to signal some special evet such as “buffer is now an item that can be consumed”
  - Up operation increments the value of semaphore.
  - If one or more operations were sleeping on the semaphore, OS chooses one of to wakup. Each wakup causes the value of semaphore decremented by one.

Use of semaphores

```
#define  N    100    /* buffer size  */

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;  /* number of empty cells in buffer innitially */
semaphore full = 0;       /* number of full cells in buffer initially */
void producer()
{
        while (TRUE)
        {
                item = produce_item();
                down(&empty);   /* sleep if there is no empty cell in
                                   buffer*/
                down(&mutex);
                insert_item (item);
                up(&mutex);
                up(&full);  /*wakeup consumer if the buffer was all
                              empty*/
        }
}
```

Use of semaphores

```
void consumer()
{
        while (TRUE)
        {
                down(&full); /* sleep if all cells empty */
                down(&mutex);
                item = remove_item();
                up(&mutex);
                up(&empty); /* wakeup producer if buffer was full */
                consume_item(item);
        }
}
```


Use of semaphores

- In the example, we have seen two separate uses of semaphores
  - 1) For mutual exclusion
    - The *mutex* semaphore is used for mutual exclusion.
  - 2) For synchronization
    - The *full* and *empty* semaphores are used for synchronization.
    - With synchronization, certain event sequences are guaranteed to occur or not to occur.

Location for Semaphore Variables

- Semaphore variables or shared variables like *turn* could be kept
  - In Kernel, so that multiple processes can share them.
  - In the shared address space of a process.
    - Some OSs allow some part of address space of a process to be shared by some other processes.
  - In a file.
    - Files can be shared by multiple processes easily.

Mutexes

- Simplified version of semaphores.
  - Can not count
- Has binary value: 0 or 1.
  - 0: unlocked
  - 1: locked
- Two procedures are used with mutexes:
  - mutex_lock()
  - mutex_unlock()

Mutex operations

- mutex_lock(mutex)
  - A process should call it when it wants to enter to its critical region.
  - If mutex was unlocked, mutex_lock() returnb immediately and process can go into its critical region.
  - If mutex was locked, mutex_lock() does not return and process is blocked.

Mutex operations

- mutex_unlock(mutex)
  - When a process is finished with its critical region, it calls mutex_unlock().
  - The result of calling mutex_unlock is: when nore more processes are waiting to acquire the mutex lock, one of the is given the lock. It wakes up and continues to execute.

Use of Mutexes

- They are usually used to synchronize threads.
- They can implemented in user space is TSL instruction is available.

```
mutex_lock:
        TSL REGISTER, MUTEX
        CMP REGISTER, #0
        JZE ok
        CALL thread_yield
        JMP mutex_lock
ok:     RET
```

```
mutex_unlock:
        MOVE MUTEX, #0
        RET
```


Monitors

- Semaphores are difficult to use correctly:
  - Order of calling operations on semaphores is important.
- Monitors is a higher level solution.
  - A solution at the programming language level.
- A monitor is a collection of procedures, variables and data structures group together.
- Compiler knows about the monitors.

Monitors

```
monitor example
        integer I;
        condition c;

        procedure producer();
        …
        …
        end;

        procedure consumer()
        …
        …
        end;
end monitor;
```

- Processes may call procedures in a monitor whenever they want.
- Only one process can be active in a monitor procedure.
- Compiler treats monitor procedures differently and provides mutual exclusion.

Monitor procedures

```
monitor x

        procedure foo();
        …
        …   Enter_Region
        …
        …   Work
        …   (includes critical section)
        …
        …   Leave_Region
        …
        end;

        ….

end monitor;
```

- Each procedure of a monitor has the following structure:
  - Enter_region: implemented by use of semaphores or mutexes to achieve mutual exclusion in executing the work part of the procedure.
  - Leave_Region: used to release locks.

Monitors

- OS provides semaphores or mutexes.
- Language provides monitor constructs.
- Compiler uses OS primitives to translate the monitor procedures so that mutual exclusion is provided
- Programmer is not worried about coding with semaphores
- Programmer just have to include critical sections as part of monitor procedures.

Condition Variables

- Monitor procedures can be called in a mutually excluded manner.
- This provides coordinated access to the shared buffer and variables.
- But how can a process can go to sleep if it can not proceed?
  - It can not proceed because it has to wait some special event to occur
  - This event can be different than an process exiting its critical region.

Condition Variable

- The solution is by:
  - Use of condition variables, and
  - Use of wait() and signal() operations (primitives) on them.
- When a process want to go to to sleep:
  - (For example, when producer wants to go to sleep since the buffer is full)
  - Process calls wait() on some condition variable x.

Condition Variable

- When an other process want to wakeup the sleeping process:
  - (for example, when consumers wants to wakeup producer, when consumer removes an item from full buffer).
  - Process calls signal() on the same condition variable x.
- Condition variables are not counters.
  - They don't save the signals.
  - Therefore, signal() should be called after wait().

Signal

- When a process calls signal (inside monitor), the waiting process will start execting (inside) monitor.
- To prevent being both processes in monitor:
  - Signaling process should exit monitor immediately.
  - Namely, signaling process should signal() as the last statement in a monitor procedure.

Producer-Consumer Problem with Monitors

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
...             if (count=N) then
                        wait (full);
                insert_item(item);
                count := count + 1;
                if (count = 1) then
                        signal (empty);
        end;
```

Producer-Consumer Problem with Monitors

```
        function remove: integer;
        begin
...             if (count=N) then
                        wait (empty);
                remove = remove_item;
                count := count - 1;
                if (count = N-1) then
                        signal (full);
        end;
end monitor;
```


Producer-Consumer Problem with Monitors

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item);
        end;
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove();
                consume_item(item);
        end;
end;
```

Limitations of semaphores and Monitors

- Semaphores are difficult to program. Careless use of them can cause deadlocks.
- Semaphore variable should be stored in a shared physical memory
  - With CPU computer this is not problem.
  - With multiple CPUs computer, each CPU can have its own local memory
  - Also, different computers will have different memories.
  - The same problem is valid for monitors also.
- Monitors require support from programming languages.
  - Most programming languages do not support: C, Pascal.
  - Java supports

Message Passing

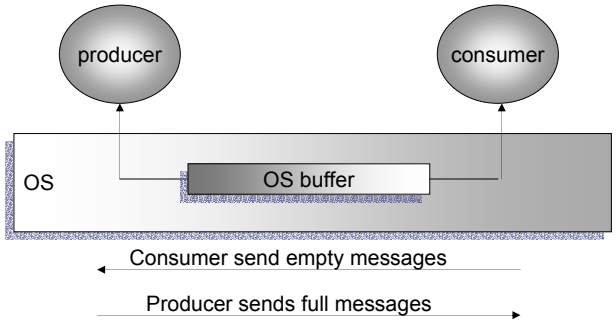
- Solves IPC between process that can reside on different computers.
- Two primitives need to be provided by OS as system calls: send and receive.
  - send (destination, &message);
  - receive (source, &message);
- At the receiver side, if no message is available, than the receiver can block.
- At the sender side, if there is no buffer space in the kernel to put the message, the sender can block.

Design Issues

- Reliability is a problem.
  - Message can get lost
  - Sender should retransmit the lost message,
  - Lost messages can be indicated to the sender by use *acknowledgment* messages.
- Receiver should be able to distinguish multiple copied of the same message.
  - Sequence numbers need to be used in message headers.
- A naming method to name processes needs to be developed.
- Authentication of sender and receiver should be done.

Producer-Consumer Problem with Message Passing

Assume producer and consumer are on the same computer



At most N messages can be outstanding at the same time.

Producer-Consumer Problem with Message Passing

```
#define N 100

void producer (void)
{
        int item;
        message m;

        while (TRUE) {
                item = produce_item();
                receive (consumer,  &m); /* first receive and
                                        empty message from consumer */
                build_message(&m, item);
                send (consumer, &m); /* send a full message */
        }
}
```


Producer-Consumer Problem with Message Passing

```
void consumer (void)
{
        int item;
        message m;

        for (i=0; i<N; i++)    /* first send N empty messages*/
                send (producer, &m);
        while (TRUE) {
                receive (producer,  &m); /* receive a full message*/
                item = extract_item(&m); /* get the item */
                send (producer, &m);   /* send back an empty
                                                message */
                consume_item(item);
        }
}
```