Interprocess Communication (Part 1)

Operating Systems

Assoc. Prof. Milos Jovanovik, PhD

Agenda

- Problems in IPC
  - Race Condition
  - Critical Section
  - Deadlocks
- Solution Techniques
  - Lock variables
  - Hardware solutions
  - Semaphores
  - Monitors
- Classical IPC Problems

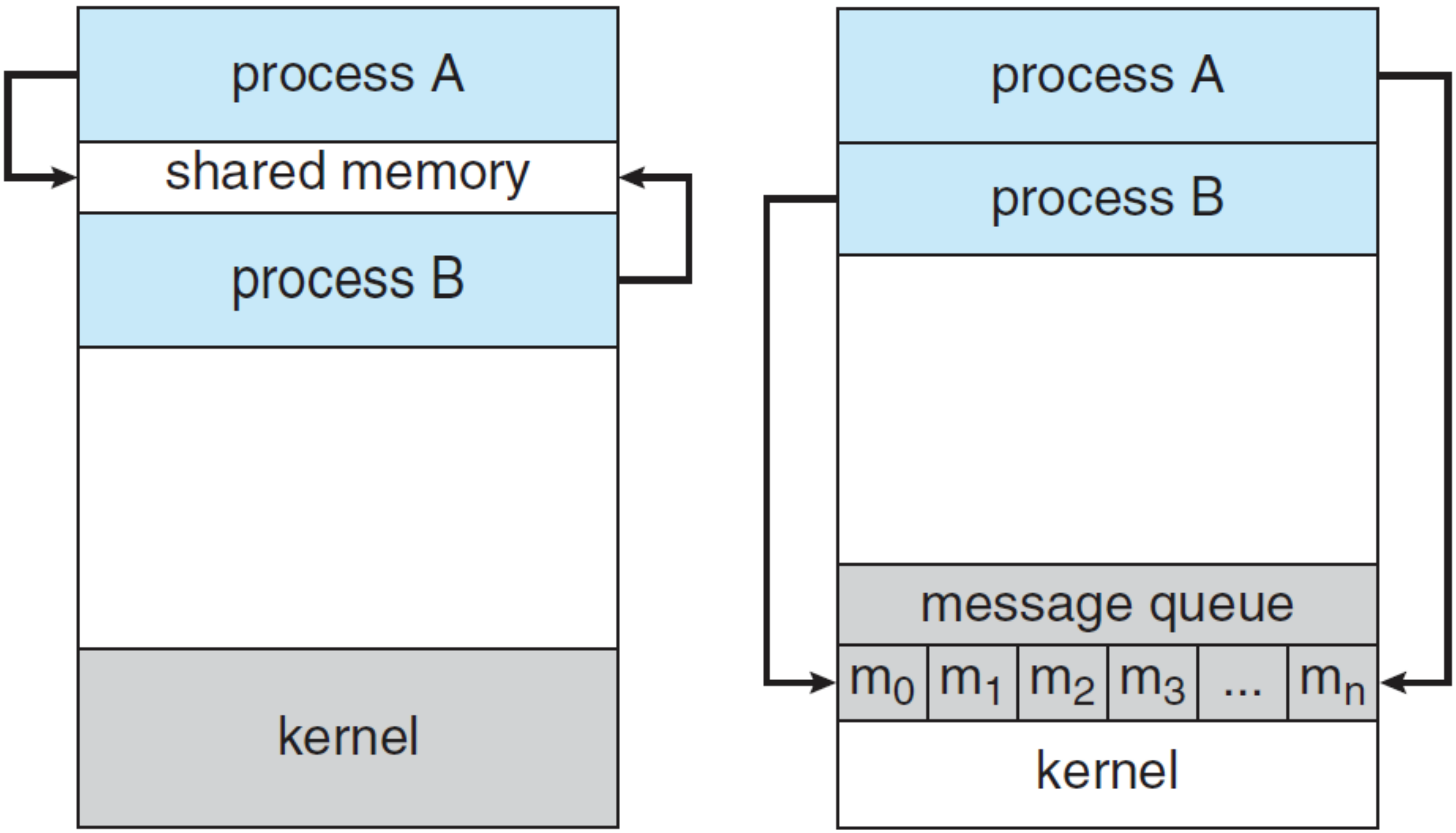
Interprocess Communication

- Sometimes, processes need to communicate
  - How can they do this?
- How can they not get in each others way when sharing resources?
  - Two clients trying to reserve the last airplane seat;
- How can they sequence their actions when there are dependencies?
  - Process A needs to provide something first, so that Process B can read and process it afterwards;

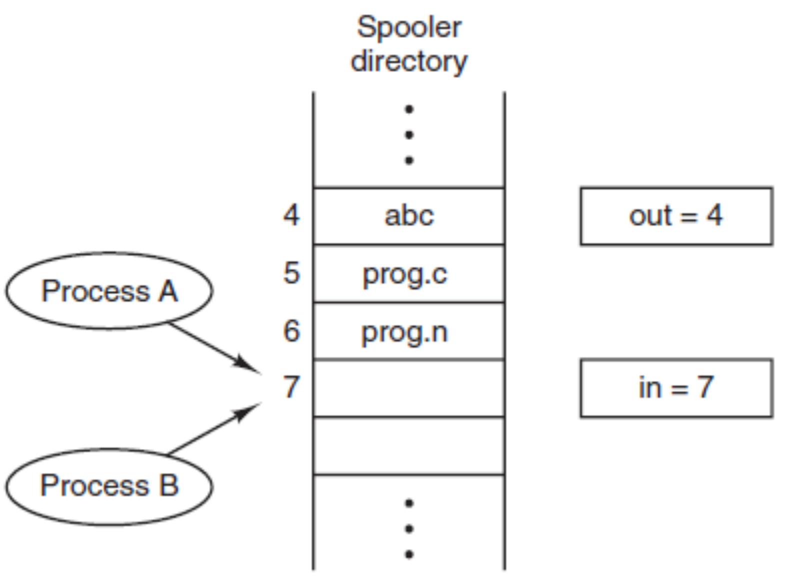
Interprocess Communication

- Processes executing concurrently in the operating system may be:
  - Independent: they cannot affect or be affected by other processes;
  - Cooperating: they can affect or be affected by other processes;
    - Any process sharing data with other processes is a cooperating process
- Models of interprocess communication:
  - Shared memory
  - Message passing

Interprocess Communication



IPC using a Shared Variable



- Two shared variables:
  - out – next file to be printed
  - in – next free slot
- Process A and B want to print
- Process A reads the variable in and memorizes it in local variable next-free
- A clock interrupt occurs and the CPU decides to switch to process B
- Process B reads the value 7 from in, sends the file name and it sets in to 8
- Process A activates again and sends the file name to the place pointed by the next-free, in location 7, “overriding” the file of process B
- User B will wait forever for the printed document

Problem: Process B starts to use the shared variable before A finishes with the variable!

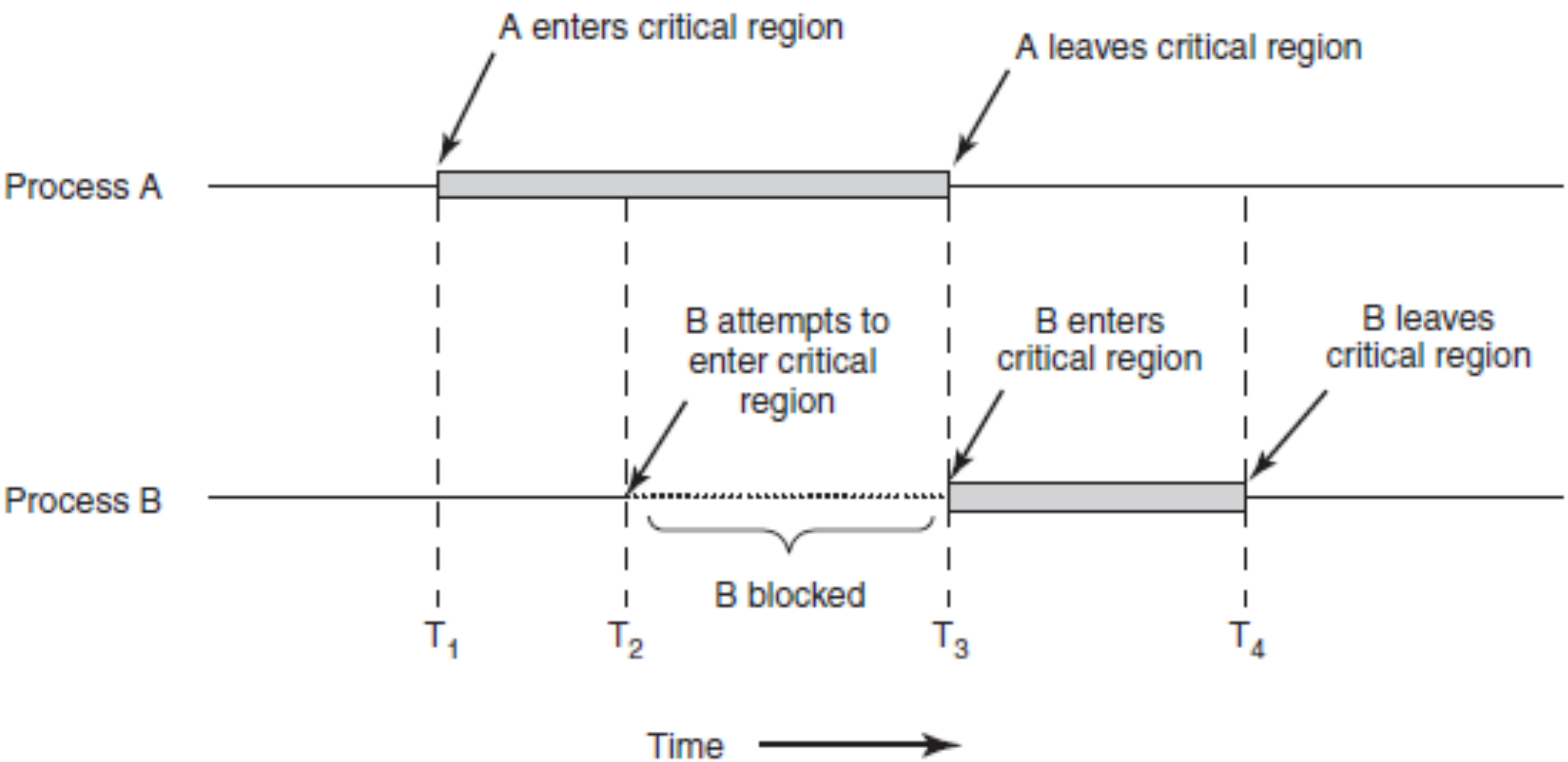
Race Condition

- Situations where two or more processes are reading or writing shared data and the final result depends on who runs precisely when, is called a **race condition** between the processes
- Race conditions are difficult to detect
  - Most test runs may be fine, with a race condition happening only once in a while;
- The user needs an OS mechanism to avoid race conditions – main design goal of OS
- Solution: mutual exclusion
  - Making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

Critical Region / Segment

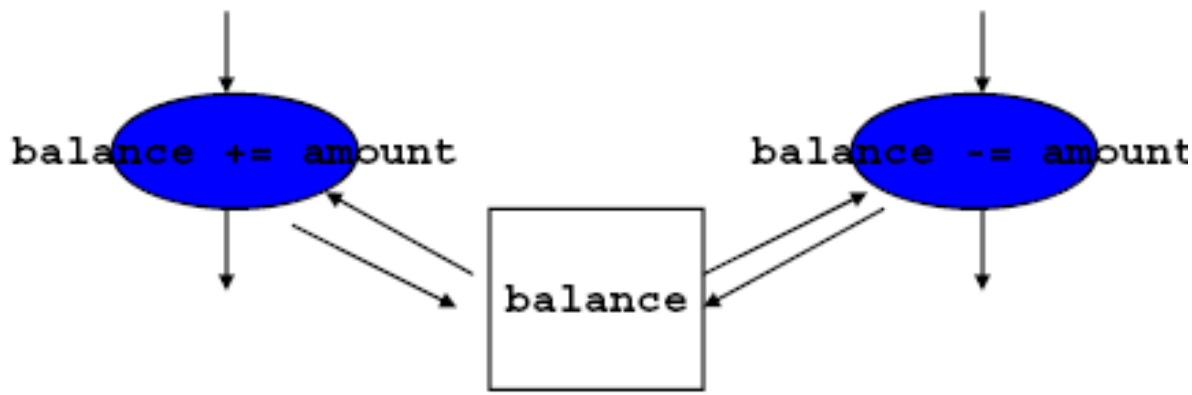
- A critical region is a part of the program where the shared memory (variables), files, etc., are being accessed
- It's an abstraction which consists of a number of sequential program statements, which have to be executed without interrupt and errors
    - No two processes should be in their critical section at the same time, so that they avoid race conditions

Critical Sections and Mutual Exclusion



Critical Sections

```
shared double balance;
Code for P1                          Code for P2
      . . .                                . . .
balance += amount;                   balance -= amount;
      . . .                                . . .
```



Critical Sections

```
shared double balance;
```

Code for P1

```
      . . .
balance += amount;
      . . .
```

Code for P2

```
      . . .
balance -= amount;
      . . .
```

Process P1 (assembler)

```
load R1, balance
load R2, amount
add R1, R2
store R1, balance
```

Process P2 (assembler)

```
load R1, balance
load R2, amount
sub R1, R2
store R1, balance
```

Race Conditions

- There is a race between the processes – who will execute its critical section
- The section can be defined in different code structures, in different processes
  - A static analytics will not detected possible race conditions
- Without mutual exclusion, multiple executions of the same programs may result with **inconsistent results**
- There is a need for an OS mechanism for the user, to be able to handle race conditions

Critical Section Problem

- A system with **n** processes $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has a critical section (it changes variables' values, tables, writes to files shared with other processes, ...)
- RULE:
  - When one process works in its critical section, no other process has permission to execute code in its own critical section;
- We need a protocol for process cooperation

General Structure of P_i

- Each process has to ask for a permission in order to enter in its critical section
- The critical section can be followed with an exit section

```
do {
   entry_section
   critical_section
   exit_section
   noncritical_section
}
```

Possible OS mechanisms

- Hardware solutions
  - Disabling interrupts
  - Special atomic instructions
- Software solutions
  - Lock variables
  - Semaphores
  - Monitors
- …

Implementation Criteria

- Each implementation needs to be:
  - **Correct**: only one process may execute the critical section at any given moment.
  - **Efficient**: entering and exiting a critical section has to be fast.
  - **Flexible**: a good implementation allows a maximized concurrency and has minimal limitations.

Correct Solution

- General conditions for a correct solution of a racing condition:
  1. Mutual exclusion: No two processes may be simultaneously inside their critical regions.
  2. Progress: If no process is in the critical section, and there are processes waiting to enter it, one of them must get access.
  3. Limited waiting: No process should have to wait forever to enter its critical region.

1. Shared Lock Variable – Busy Waiting

A
l
g
o
r
i
t
h
m
1

- Processes (P_0 and P_1) are sharing the variable turn
- A process can access a variable when its turn comes
- When a process wants to access a variable, but it is not its turn, it is in a state called busy waiting (wasting CPU time)

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

(a)

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

(b)

Advantages and Problems

- This solution allows only one process to be in the critical section in a given moment.
- It does not allow progress: we need strict process alternation of the processes which are executing the critical section.
- **A problem arises if one process is much slower than the other.**

Process 0	Process 1
	turn = 0
wait for turn = 0	noncritical_region (enter)
critical_region (enter)	
critical_region (finish)	
turn = 1	
noncritical_region (enter)	
noncritical_region (finish)	
wait for turn = 0 (rule 2 is broken!)	

Algorithm 2

- The first algorithm does not keep all the necessary information for the process state, it keeps only information about which process can enter in its critical section
- We introduce a Boolean array flag[P_1..P_2]
- If flag[P_i] is true, the i^{th} process P_i is ready to enter its critical section
- The i^{th} process checks if the j^{th} process is NOT ready to enter its critical section
- If also flag[P_j] is true, then P_i waits until flag[P_j] becomes false, then P_i will enter its critical section

Solution: Locking

```
Flag[P1..P2]
Process 1

   Flag[P1] = true;
   while(Flag[P2])
     ; // wait
   /* enter C.S. */
   /* leave C.S. */
   Flag[P1] = false;
   ..................
```

```
Process 2

   Flag[P2] = true;
   while(Flag[P1])
     ; // wait
   /* enter C.S. */
   /* leave C.S. */
   Flag[P2] = false;
   ..................
```

Mutual exclusion with busy waiting

Problem:

deadlock: (Flag[P_1]=true , Flag[P_2]=true), infinite loop → rule 3 is broken!

Algorithm 3

- By combining the two algorithms, we obtain an algorithm which fulfils all three requirements for correctness
- In order to enter the critical section, P_i sets the flag[P_i] to true, and sets turn=P_j, which means that the other processes can enter their critical section (if they need to)
- The process that sets the flag and has its own value in the variable turn, enters the critical section

Solution: Combination

```
Flag[P1..P2], turn
Process 1

   Flag[P1] = true;
   turn = P2;
   while(Flag[P2] and turn == P2)
      ;
   /* enter C.S. */
   /* leave C.S. */
   Flag[P1] = false;
   ………………
```

```
Process 2

   Flag[P2] = true;
   turn = P1;
   while(Flag[P1] and turn == P1)
      ;
   /* enter C.S. */
   /* leave C.S. */
   Flag[P2] = false;
   ………………
```

Peterson's Solution

(modern version of Dekker's algorithm)

Peterson’s Solution

```
#include <prototypes.h>
#define FALSE 0
#define TRUE 1
#define N 2                              // number of processes
turn int turn, interested[N]    // all values initially 0
void enter_region(int process)
{
   int other;                            // number of the other process
   other = 1 - process;
   interested[process] = TRUE;                // show that you are
     interested
   turn = other;                         // set the flag
   while ( turn == other && interested[other] == true) ; // wait
}
void leave_region(int process) { // process who is leaving
   interested[process] = FALSE;  // indicate departure from C.S.
}
```

Peterson's Solution (Example)

Process = 0

```
interested[0]=interested[1]=FALSE
int turn;
int interested[2];

void  enter_region(...){
    int  other = 1;
    interested[0] = TRUE;
    turn = 1;
    while (turn == 1 &&
            interested[1] == TRUE);
}

void  leave_region(...){
    interested[0] = FALSE;}
```

Process = 1

```
interested[0]=interested[1]=FALSE
int turn;
int interested[2];

void  enter_region(...){
    int  other = 0;
    interested[1] = TRUE;
    turn = 0;
    while (turn == 0 &&
            interested[0] == TRUE);
}

void  leave_region(...){
    interested[1] = FALSE;}
```

2. Locking: Hardware support

- Disabling interrupts
- Special machine instructions

2. Disabling interrupts

- A simple solution for a single-processor systems:
  - Disable interrupts before the critical section
  - Enable interrupts after the critical section
    - When interrupts are disabled, no clock interrupt can occur -> there is no context switch -> there is no time limit for the process
- Disabling HW interrupts is a dangerous manoeuvre for the OS:
  - It can postpone the system response to important events
  - A process may never exit, leaving the interrupts blocked!
- Unwise solution for multi-processor systems
  - Disabling interrupts affects only one core, while other cores may access the shared variable
- It is not suitable as a technique for user processes
  - Convenient approach for the kernel to disable interrupts for a few instructions, while it is updating variables or lists
  - It does not work for systems with multiple CPUs (CPU cores)

Example: Disabling Interrupts

```
shared double balance;
```

Process P1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Process P2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

- The interrupts can be disabled an arbitrarily long time
- We just want P_1 и P_2 not to interfere; using this technique we block all P_i
- We can use a mutual “lock” variable

2. Special Atomic Instructions

- Special atomic machine instructions
  - CPU executes them in **a single** instruction cycle
  - They are not subject to instruction mixing
  - They are used for control of access to a memory location:
    - Reading and writing
    - Reading and testing

Test and Set Lock Instruction

- It executes **atomically**, in **one cycle**, **without interrupts**
  - TSL RX,LOCK reads the contents of the shared memory word lock into register RX and then stores a nonzero value at the memory address lock.
    - No other processor can access the memory word until the instruction is finished.
    - The CPU executing the instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

```
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
```

Test and Set Lock Instruction–Realization

- This instruction cannot be interrupted – no one can access the memory while it is executing

```
boolean TestSet (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

Usage

- Shared boolean variable lock, initially set to false.
- Solution:

```
int lock=false;
while(true){
        while(!TestSet(&lock))
                  ;   /* do nothing
         //    critical section
         lock = FALSE;
        //      uncritical section

 }
```

```
boolean TestSet(int i) {
  if (i == 0) {
      i = 1;
      return true;
      }
  else {
      return false;
      }
  }
```

Exchange of Values

- Exchange instruction (XCHG)
- An atomic exchange between a register and a memory location
- Atomic operation, in a single CPU instruction

```
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
```

XCHG Realization

```
void exchange(int register, int memory) {
     int temp;
     temp = memory;
     memory = register;
     register = temp;
  }
```

Machine Instructions for Mutual Exclusion

- Pros
  - Can be used for an arbitrary number of processors which can share the same working memory
  - They are simple for work and verification
  - Can be used for several critical sections

Machine Instructions for Mutual Exclusion

- Cons
  - Busy waiting wastes CPU time
  - Starvation is possible when a process exits from the critical section, but more than one process are waiting to enter

Basic Problem

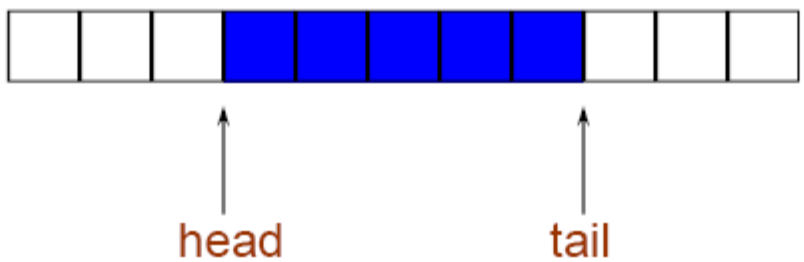
- These approaches (with busy waiting) are wasting CPU time while waiting for a permission to enter the critical section
- Example (Priority Inversion Problem)
  - 2 processes with different scheduling priorities:
    - H – with high priority (the CPU is assigned to this process when it is in ready state)
    - L – with low priority
  - Scenario
    - L is executing the critical section
    - H becomes ready and the CPU is assigned to it (because it has higher priority).
    - H begins with busy waiting because L is in the critical section (locked the access)
    - L cannot leave the critical section because H “is working” (even though it waits), and therefore, is never scheduled
    - H is in an infinite loop

sleep(), wakeup()

- Solution: Use of system calls which block the process instead of busy waiting
- Example:
  - sleep() suspends the process until some other process awakes him with wakeup()
  - wakeup() has one parameter, the process that needs to be awaken
- System calls of this type are used in solutions with blocking

Bounded Buffer Problem

- Producer-consumer problem
- Important for: network interfaces, I/O controllers, message exchange, etc.



- Bounded buffer with limited capacity (N) shared by many processes
  - The producer puts information into the buffer (on the tail)
  - The consumer takes information out of the buffer (from the head)

Bounded Buffer Problem

- Possible problems:
  - The producer wants to put a new item in the buffer, but it is already full.
  - The consumer wants to remove an item from the buffer, but the buffer is empty.
- Simple solution:
  - Full buffer for the producer: it goes to sleep until the consumer removes one or more items and wakes it up.
  - Empty buffer for the consumer: it goes to sleep until the producer puts something in the buffer and wakes it up.

Solution: Classical Blocking

```
#include <prototypes.h>
#define N 100 // number of slots
int count 0;
void producer (void) {
 int item;
 while (TRUE) {
  produce_item(&item); //next item
  if (count == N) sleep(); //full?
  insert_item(item);    //put item
  count = count + 1;
  if (count == 1)
   wakeup(consumer);    // empty?
} }
```

```
void consumer (void) {
 int item;
 while (TRUE) {
  if (count == 0) sleep(); //empty?
  remove_item(&item); // take item
  count = count - 1;
  if (count == N-1)
   wakeup(producer);  // full?
  consume_item(item); // print item
  }
 }
```

Realization

- Problem: a race condition because access to *count* is unconstrained

Producer	Consumer
	Test of count==0, read count (just before sleep())
insert_item()	
count++	
wakeup(consumer)	consumer is not sleeping (signal is lost)
produce next item	
	count==0 => sleep
insert_item()	
…	
count==N => sleep	

Questions?