Operating Systems

Mutual Exclusion and Synchronization

dr. Tomasz Jordan Kruk

T.Kruk@ia.pw.edu.pl

Institute of Control & Computation Engineering
Warsaw University of Technology

Race Conditions in Operating System

IPC - InterProcess Communication

In operating systems running processes often share common memory areas, files or other resources. So called races are to be avoided.

Def. 1
Race condition – situation in which two or more processes perform some operation on shared resources and the final result of this operation depends on the moment of realization of the operation.

Race Condition Occurence Example

An example

```
void echo()
{
     chin = getchar();
     chout = chin;
     putchar( chout );
}
```

	PROCESS 1	PROCESS 2
1		
2	`chin = getchar();`	
3		`chin = getchar();`
4	`chout = chin;`	`chout = chin;`
5	`putchar( chout );`	
6		`putchar( chout );`
7		

Critical Region/ Critical Section

To avoid race conditions some mechanism must be created to protect against accessing resources by more than one process at the same time. Some **mutual exclusion** mechanizm must be introduced.

Def. 2
Critical region – (also: **critical section**), the piece of a program, in which there are some instructions accessing shared resources. Instructions constituting the critical region must be preceded and completed with instructions implementing mutual exclusion.

The choice of operations implementing mutual exclusion mechanism is an important feature of each operating system.

Logical Conditions to Implement Critical Region

For correct critical region implementation the following four conditions are required:

1. No two procesess 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 the critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Mechanisms Implementing Mutual Exclusion

Two approaches:

1. Mechanisms with busy waiting for accessing critical region:
   (a) disabling interrupts,
   (b) lock variables (*incorrect*),
   (c) strict alternation (*incorrect*),
   (d) Peterson's solution,
   (e) TSL instruction.
2. Mechanisms with suspension of the waiting process:
   (a) sleep and wakeup (*incorrect*),
   (b) semaphores,
   (c) monitors,
   (d) message passing.

Mechanisms with Busy Waiting

1. Disabling interrupts

- √ each process entering critical region disables interrupts,
- √ advantage: process inside critical region may update shared resources without any risk of races,
- √ disadvantage: if after leaving the region interrupts are not reenabled there will be a crash of the system. Moreover: useless in multiprocessor architectures,
- √ may be used inside operating system kernel when some system structures are to be updated, but is not recommended for implementation of mutual exclusion in user space.

2. Lock variables

A software solution. Considering having a single, shared lock variable, initially 0. When process A attempts to enter critical region:

- √ if lock = 0, set lock to 1 and enter critical region;
- √ if not, wait until lock becomes 0.

Thus:

- √ lock = 0 means no process in critical region,
- √ lock = 1 means there is a process in critical region.

This solution is **incorrect**, race conditions occur.

3. Strict Alternation

PROCESS 0

```
1  while( TRUE )
2  {
3    while( turn != 0 )
4      /* wait */;
5    critical_section();
6    turn = 1;
7    noncritical_section();
8  }
```

PROCESS 1

```
while( TRUE )
{
  while( turn != 1 )
    /* wait */;
  critical_section();
  turn = 0;
  noncritical_section();
}
```

- √ initially turn=0, the third condition violated. P0 may be blocked by P1 outside the critical region. Such situation is called **starvation**,
- √ this solution requires strict alternation (switching), e.g. two files cannot be printed one after another by the same process,
- √ this solution is **incorrect**, the problem of race conditions replaced by the problem of starvation.

4. Peterson's Solution (I)

- √ connecting two ideas: strict alternation and locking variables T. Dekker as the first person (1965) found a correct solution to the mutual exclusion problem. In 1981 Peterson found simpler solution to this problem.
- √ each process before entering critical region, calls enter_region with its number as a parameter, after leaving critical region leave_region is called.

Peterson's Solution (II)

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N]; /* initially 0 */

enter_region(int process)    /* process nr 0 or 1 */
{
  int other;
  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while( (turn == process) && (interested[other] == TRUE) );
}

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


5. TSL Instruction

Hardware support, some computer architectures offer an instruction **TEST AND SET LOCK** (**TSL**)

- √ instruction TSL executes indivisibly in the following way:
  - ⋆ reads the value of one word from the memory to some register,
  - ⋆ at the same moment stores the value from that register to the same location in memory.
- √ read and write operations are indivisable, i.e. any other process does not have any access to the memory location untill the TSL instruction finishes.

To demonstrate usage of TSL let us use shared variable *flag* to coordinate access to shared resources.

- √ when flag = 0, each process may set it to 1 with TSL instruction, and after this enter critical region,
- √ leaving the critical region flag should be set to 0 with *move*.

Critical region Organization with TSL

```
1  enter_region:                    leave_region:
2    tsl register, flag               mov flag, #0
3    cmp register, #0                 ret
4    jnz enter_region
5    ret
```

Processes competing for critical region must call *enter_region* and *leave_region* procedures in correct order.

Disadvantages of solutions based on busy waiting

- √ waste of processor time,
- √ possibility of deadlock/starvation in systems with multipriority scheduling, so called **priority inversion**.

Solutions with Waiting Process Suspension

1. Sleep and Wakeup

The simplest solution is to create two system calls **sleep()** and **wakeup()**.

- √ by calling **sleep()** the calling process is suspended till being woken by other process calling wakeup(),
- √ **wakeup** function called with process number as a single argument.

Example of sleep()/wakeup() Usage

Producer-consumer problem - problem of a buffer with limited capacity.

Let two processes share a buffer with limited capacity. Process called producer will put pieces of information in the buffer. Process called consumer will take pieces of information from that buffer.

Let us assume:

- √ if Pr is trying tu put a message into full buffer, Pr hat to be suspended,
- √ if Co is trying to take a message from the empty buffer, Co has to be suspended.

Let in *count* variables number of taken positions in the buffer is hold. Let the size of the buffer will be *N*.

Producer: if(count == N) { go to sleep } else { add *msg* and count++ }

Consumer: if(count == 0) { go to sleep } else { take *msg* and count– }

Producer-Consumer with Race Conditions

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

void producer(void)
{
while (TRUE){
     produce_item();
     if (count == N)
        sleep();
     enter_item();
     count = count + 1;
     if (count == 1)
        wakeup(consumer);
     }
}
```

```
void consumer(void)
{
while (TRUE){
     if (count == 0)
        sleep();
     remove_item();
     count = count - 1;
     if (count == N-1)
        wakeup(producer);
     consume_item();
     }
}
```

Disadvantage: *wakeup* signal may be lost, which leads to deadlock.

2. Semaphores: definition

- √ 1965 r. - E. W. Dijkstra proposed integer variable to count *wakeup* signals,
- √ proposed variable called **semaphore**, initialized with nonnegative integer value and defined by definition of two atomic operations, **P(s)** i **V(s)**:

  P(S): while $S \leq 0$ do
  ;
  $S := S - 1$;

  V(S): $S := S + 1$;

- √ Dutch P and V from *proberen (to test)* and *verhogen (to increment)*, now usually *down()/up()*, *wait()/signal()*, and for binary semaphores often *lock()/unlock()*.

Semaphores: Implementation

```
struct semaphore
{
   int    count;
   queue_t  queue;
}

void down( semaphore s )
{
   s.count--;
   if( s.count < 0 )
   {
      enter process to
         queue s.queue;
      block proces;
   }
}
```

```
void up( semapahore s )
{
   s.count++;
   if( s.count <= 0 )
   {
         remove one process
              from s.queue and
              put to ready queue;
   }
}
```


Semaphores: Producer-Consumer Algorithm

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full  = 0;
int count = 0;
```

```
void producer(void)
{
   while (TRUE)
   {
      produce_item();
      down( empty );
      down( mutex );
      enter_item();
      up( mutex );
      up( full );
   }
}
```

```
void consumer(void)
{
   while (TRUE)
   {
      down( full);
      down( mutex );
      remove_item();
      up( mutex );
      up( empty );
      consume_item();
   }
}
```


Mutex - the Binary Semaphore

- √ used, when there is no requirement to count signal occurences but only to organize **mut**ual **ex**clusion,
- √ efficient and simple implementation, e.g. for user-level threads.

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

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

3. Monitors

In order to make writing programs with mutual exclusion easier, Hoare (1974) and Hansen (1975) proposed higher-level synchronization mechanism, **monitor**.

- √ **monitor** is a set of procedures, variables and structures, which are grouped together in one structure. Only one active process may be inside monitor,
- √ monitors are constructions of higher-level languages. Responsibility for correct implementation of mutual exclusion is in charge of compiler,
- √ in introduced concept there is a lack of mechanism for process suspension,

Monitors: Notation

Monitor representation in some language with Pascal-like syntax.

```
monitor Buffer
 var
  byte   b[100];
  integer head, tail;
 procedure insert( int item )
 begin
  ...
 end;
 procedure remove( int item )
 begin
  ...
 end;
end monitor;
```

```
monitor Buffer
{
   char   b[100];
   integer head, tail;
   public void insert( Item i )
   {
    ...
   }
   public Item remove( void )
   {
    ...
   }
}
```


Monitors: Process Suspension

It was proposed to introduce **conditional variables** with two operations **wait(variable)** and **signal(variable)**.

- √ when monitor procedure discovers that it is not possible to continue some operation, *wait* is performed on some conditional variable. Process which executes procedure is suspended.
- √ other process may now enter the critical region. When it lives, it performs *signal* in order to wake up the process suspended on some conditional variable.

After *signal* calling:

- √ Hoare version: the awoken process continues execution and the calling one is suspended,
- √ Hansen version: the calling process has to leave at once the monitor.

Monitors: Producer-Consumer Algorithm (I)

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

   procedure enter;
   begin
      if count = N
        then wait(full);
      enter_item;
      count := count + 1;
      if count = 1
        then signal(empty);
   end;

   count := 0;
end monitor;
```

```
procedure remove;
begin
   if count = 0
     then wait(empty);
   remove_item;
   count := count - 1;
   if count = N-1
     then signal(full);
end;
```

Monitors: Producer-Consumer Algorithm (II)

```
procedure producer;
begin
  while true do
  begin
       produce_item;
       Buffer.enter;
  end
end;
```

```
procedure consumer;
begin
  while true do
  begin
       Buffer.remove;
       consume_item;
  end
end;
```

Monitor mechanism, main features:

- √ wait()/signal() function pair protects against loosing signals (what may happen with sleep()/wakeup()),
- √ not all higher-level languages offer monitors (Euclid, Concurrent Pascal),
- √ some languages offer incomplete mechanisms (Java and *synchronized*),
- √ solutions not dedicated for distributed environment because of required accessibility of shared memory.

Monitors: Producer-Consumer Algorithm in Java (I)

```
public class ProducerConsumer {
 static final int N = 100;              static producer p = new producer();
 static consumer c = new consumer();  static our_monitor  mon = new our_monitor();
  public static void main(String args[]) {
    p.start();
    c.start(); }

  static class producer extends Thread {
    public void run() {  // contains thread code
      int item;
      while (true) {
        item = produce_item();
        mon.insert(item); } }
    private int produce_item(){ } }

  static class consumer extends Thread {
    public void run(){
      int item;
      while(true){
        item = mon.remove();
        consume_item(); } }
    private void consume_item(){ } }
```


Monitors: Producer-Consumer Algorithm in Java (II)

```
  static class our_monitor{
    private int buffer[] = new int [N];    private int count = 0, lo = 0, hi = 0;

    public synchronized void insert(int val){
      if (count == N) go_to_sleep();
      buffer(hi) = val;
      hi = (hi +1) % N;   // ring buffer
      count = count + 1; // one more item in buffer
      if (count == 1) notify(); // notify() in in Java }
    public synchonized int remove(){
      int val;
      if (count == 0) go_to_sleep(); // buffer empty
      val = buffer[lo];
      lo = (lo+1) % N;
      count = count - 1;
      if ( count == N-1) notify();
      return val; }
    private go_to_sleep(){
      try {wait();}
      catch(InterruptedException exp){}; }
  }
}
```


4. Message Passing

Based on two system calls:

- √ **send**(*destination*, &*message*);
- √ **receive**(*source*, &*message*);

Different methods of message addressing:

1. **direct addressing**, each process contains unique address. The following rendezvous mechanism may be used:
   - √ if *send* called before *receive*, the sending process suspended till the moment of the very message sending after *receive* call,
   - √ if *receive* called before *send*, the receiving process suspended till the moment of the very message sending after *send* call.
2. **indirect addressing**, via some mailbox playing the role of the intermediate buffer. *send* and *receive* has as an argument mailbox address, not the address of any particular process.

Messages: Producer-Consumer Algorithm (I)

Preliminary assumptions:

- √ messages have the same size,
- √ messages sent but still not received automatically buffered by the operating system,
- √ N messages used, analogically to N positions in a shared buffer,
- √ messages treated as a transport medium for information, i.e. they may be either full or empty,
- √ algorithm starts with sending by consumer N empty messages to producer,
- √ it is obligatory for producer to have an empty message received from consumer in order to send a message to consumer. The number of messages between producer and consumer is constant and irrespective of production or consumption speed.

Messages: Producer-Consumer Algorithm (II)

```
#define N 100

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


  while( TRUE )
  {
    produce_item( &item );
    receive( consumer, &m );
    build_message( &m, &item );
    send( consumer, &m );
  }
}
```

```
void consumer( void )
{
  int item, i;
  message m;
  for( i = 0; i < N; i++ )
    send( producer, &m );

  while( TRUE )
  {
    receive( consumer, &m );
    extract_item( &m, &item );
    send( consumer, &m );
  }
}
```


The Dining Philosophers Problem (I)

- √ five philosophers sitting around a circular table,
- √ five plates and five forks alternately on the table,
- √ each philosopher only eats and thinks, for eating one plate and two forks are required,
- √ fork is a resource shared by adjacent philosophers,
- √ how to organize synchronization?

The Dining Philosophers Problem (II)

```
#define N 5

void philosopher( int i )
{
    while( TRUE )
    {
        think()
        take_fork( i );
        take_fork( ( i + 1 ) % N );
        eat();
        put_fork( ( i + 1 ) % N );
        put_fork( i );
    }
}
```

Incorrect solution – possible deadlock occurence.

The Dining Philosophers Problem (III)

```
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
```

```
#define LEFT ( i + N - 1) % N
#define RIGHT ( i + 1 ) % N
int state[N];
semaphore mutex = 1;
semaphore s[N];
```

```
void philosopher(int i) {
     while (TRUE) {
          think( );
          take_forks(i);
          eat( );
          put_forks(i);
     }
}

void take_forks(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```

```
void put_forks(i) {
     down(&mutex);
     state[i] = THINKING;
     test(LEFT);
     test(RIGHT);
     up(&mutex);
}

void test(i) {
     if (state[i] == HUNGRY && \\
          state[LEFT] != EATING && \\
          state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```


The Readers-Writers Problem

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void
writer( void )
{
  while( TRUE )
  {
    think_up_data( );
    down( &db );
    write_data_base( );
    up( &db );
  }
}
```

```
void
reader(void)
{
  while( TRUE )
  {
    down( &mutex );
    rc = rc + 1;
    if( rc == 1 )
      down( &db );
    up( &mutex );
    read_data_base( );
    down( &mutex );
    rc = rc - 1;
    if( rc == 0 )
      up(&db);
    up(&mutex);
    use_data_read( );
  }
}
```


The Sleeping Barber Problem

```
#define CHAIRS 5
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber( void )
{
  while( TRUE )
  {
    down( &customers );
    down( &mutex );
    waiting = waiting - 1;
    up( &barbers );
    up( &mutex );
    cut_hair();
  }
}
```

```
/* how many sitting on chairs */
/* how many sleep without work */

void customer( void )
{
  down( &mutex );
  if( waiting < CHAIRS )
  {
    waiting = waiting + 1;
    up( &customers );
    up( &mutex );
    down( &barbers );
    get_haircut();
  } else {
    up( &mutex );
  }
}
```