# Tasks for the Problem Session on RTSD 1.06.2020

**Definition of semaphore operations on counting semaphore S(value, queue, max-value, initial-value):**

**P(S): if (--S.value < 0) process is blocked and en-queued into S.queue;**

**V(S): if (S.value < max-value) then S.value++; if (S.value <= 0) then**

Assume MaxSem is a maximal value of the counting semaphore, CountSem – binary semaphore initialized by 0 (closed), GuardSem – binary **activate one of the processes in S.queue (delete it from S.queue and en-queue into dispatching queue);**

**Task 1.** Implement counting semaphore with the help of binary (binary semaphore is a counting semaphore with max-value=1).

semaphore initiated by 1 (open), shared by multiple processes. Then

Const

Struct counting\_sema{

GuardSem:TBinSem=1;//global binary semaphore, initialized by 1, guards counter

CountSem:TBinSem=0;//global binary semaphore to simulate counting semaphore

R:integer=MaxSem;//global integer, representing current value of counting

//semaphore; negative values give the number of suspended processes

} Count\_sema;

CP(counting\_sema &count\_sema){//counter P operation

With (count\_sema){

P(GuardSem); //lock counter

R:=R-1; //decrement semaphore value

If (R<0){//none available?

V(GuardSem);//release counter

P(CountSem);//wait for free resource

}

V(GuardSem);//release counter

}

}

CV(counting\_sema &count\_sema){//counter V operation

With (count\_sema){

P(GuardSem);//lock counter

If (R<MaxSem) R:=R+1;//free resource, if possible

If (R<=0) //any task waiting for free resource

V(CountSem);//give that task to go ahead

Else V(GuardSem);//release counter

}

}

For V operation, releasing of GuardSem in the case of R<=0 is made by activated process (last V(GuardSem) in P operation).

**Task 2.** (Laplante, 2nd Ed., p. 187, Ex. 3 ; 3rd Ed., Ex. 3.21, p. 159). Why it is not wise to disable interruptions before the while loop in the following implementation of P-operation on binary semaphore S:

Procedure P(var S:Boolean);

Begin

While s=false do;

S:=false

End;

Procedure V(var s:Boolean);

Begin

S:=true

End;

If to disable interruptions before *while* loop:

Procedure P(var S:Boolean);

Begin

Asm

Cli

End;

While s=false do;

S:=false;

Asm

Sti

end

End; //It is a Pascal code

then in the case of false value of *s* it couldn’t be set to 1 because control yielding will not be allowed, and deadlock could happen. From the other hand, if to disable interrupts after while loop and before assignment, it is not reasonable because control yielding to a next process can happen before disabling of interrupts, and the next process may also see true value of s. So, disable interruptions after while is useless. But without such disabling we can get several processes watching the same not-modified yet value of s. What to do? Solution follows:

Procedure P(var S:Boolean);

Begin

While true do begin //infinite loop

While s=false do;//wait for true value of s

//this loop is made with enabled interrupts!

Asm //disable interruptions

Cli

End;

If s then begin //again check for true value

S:=false; //reset if s was true

Asm

Sti //enable interruptions

End;

Break //exit infinite loop

End //end of then branch

Asm

Sti //enable interruptions

End; //here we come if s became false in between while loop and interrupt

//disabling

End //end of infinite loop

End;//end of P

Simpler solution, but requiring more interruption switches:

Procedure P(var s:boolean);

Var

Temp:Boolean;

Begin

Repeat

Disable interrupts;//cli

Temp:=s;

S:=false;

Enable interrupts;//sti

Until temp=true;

End;

Procedure V(var s:boolean);//for both cases above of P operation

Begin

S:=true

End;

**Task 3.** Implement a pipe mechanism of inter-processes communications using circular buffer and semaphores. Consider non-blocking interactions (Similar to Laplante, 2nd Ed., p. 188, Ex. 15)



CircularBuffer has N places for information portions (bytes, sectors, blocks, etc.). We use 2 pointers associated with CircularBuffer: Head, Tail. Tail points to a position where a next portion of information is to be written, Head points to a position from which a next portion of information is to be read. After reading a portion of information, we assume it is not available any more for other requests, and its place may be reused after read operation termination. Circular means that after writing into the 1st position we write into the 2nd and so on, and after writing in the last N-th position we shall try to write in the 1st position; similar for reading. Places for the next write and read operations are pointed by Tail and Head respectively. For circulating, we use (x mod N) operation (remainder after integer division by N).

Simultaneous reading and writing are prohibited, for this sake we use a binary semaphore, BufferSem, initialized by 1 (open).

Pascal code follows:

Const

N=20; //buffer on 20 places

Head:integer=1;

Tail:integer=1;//Head and Tail at first show to the same position, that corresponds to

//empty buffer state

Type

Buffer=array[1..N] of Elements;//type of buffer

Var

BufferSem: BinSemaphore=1;//global semaphore variable shared by several processes

CircularBuffer:Buffer;

function WriteBuff( Info:element):integer;//return -1 if no place to write

begin

P(BufferSem);

If Tail mod N+1<>Head then begin //there is space for next write

CircularBuffer[Tail]:=info; //write into tail position

Tail:=Tail mod N+1;//shift tail pointer

result:=0;//successfully wrote

End

Else

result:=-1;//no space in buffer

V(BufferSem);

End;

function ReadBuff( var Info:element):integer;//return -1 if buffer is empty

begin

P(BufferSem);

If Head<>Tail then begin //buffer is not empty

Info:=CircularBuffer[Head];//read from head

Head:= Head mod N+1;//shift head pointer

result:=0;//successfully read

End

Else

result:=-1;//empty buffer

V(BufferSem);

End;

Producer uses

If WriteBuff(information)=0 then begin

Work in the case of successful write operation

End

Else begin

Work in the case of not writing successfully

End

**Task 4.** Assume that a system has 5 processes, and resources of 3 types: processors (total available number is 10), memory modules (10 modules available), disks (10 disks available). Processes’ resources used/required are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Processors | Memory | Disks |
| 1 | 1 / 0 | 1 / 1 | 0 / 3 |
| 2 | 2 / 1 | 1 / 2 | 1 / 1 |
| 3 | 1 / 0 | 2 / 0 | 1 / 5 |
| 4 | 2 / 0 | 1 / 3 | 1 / 0 |
| 5 | 2 / 4 | 1 / 0 | 2 / 0 |

Draw a resource graph and decide using reduction procedure, whether a deadlock exists. Can Habermann’s (Banker) algorithm be applied here?

On the graph above, only given processors are shown (with respective number of given processors). Requests on processors are shown without required quantities of processors. If to show all details, graph will be too complicated for viewing.

To decide on deadlock existence, we apply the reduction procedure, in which we analyze, whether it is possible to delete all the edges incident to a particular process. If yes, then the edges are deleted, and occupied by them resources are freed.

Currently, number of free resources: P (processors) – 2; M (memory) – 4; D (disks) – 5.

Start with P1. P1 wants 1 memory (4 available) and 3 disks (5 available). This request can be satisfied. Hence we delete all edges incident to P1. After that number of free resources: P (processors) – 3; M (memory) – 5; D (disks) – 5.

P2 wants 1 processor (3 available), 2 memory units (5 available), 1 disk (5 available). All requests of P2 can be satisfied. Hence, we delete all edges incident to P2. After that number of free resources: P (processors) – 5; M (memory) – 6; D (disks) – 6.

P3 wants 5 disks (6 available). All requests of P3 can be satisfied. Hence, we delete all edges incident to P3. After that number of free resources: P (processors) – 6; M (memory) – 8; D (disks) – 7.

P4 wants 3 memory units (8 available). All requests of P4 can be satisfied. Hence, we delete all edges incident to P4. After that number of free resources: P (processors) – 8; M (memory) – 9; D (disks) – 8.

P5 wants 4 processor (8 available). All requests of P5 can be satisfied. Hence, we delete all edges incident to P5. No edges are in the graph now. It means that there is no deadlock.

Habermann’s algorithm here is not applicable because maximal requests of the processes are not known.

**Task 5.**  In the conditions of Task 4, build task waiting graph.



Process 5 waits for all other processes: when will appear necessary to it 4 processors. Other processes also wait for Dispatcher’s decision. There are no loops, and hence, there are no deadlocks.

**Task 6.** Assume that a system has 5 processes and resources of 1 type: processors (total available number is 10). Processes’ resources required and maximal required are as follows:

|  |  |  |
| --- | --- | --- |
| Process | Processors required  (C) | Processors maximal required (B) |
| 1 | 1 | 3 |
| 2 | 2 | 2 |
| 3 | 1 | 2 |
| 4 | 2 | 4 |
| 5 | 2 | 3 |

Use Habermann’s (Banker) algorithm to decide safety of granting required processors

Available processors after granting: r=2

B1-c1=2<=r=2 ? yes -> S={P1}

B2-c2=0<=r+c1=3? Yes -> S={p1,p2}

B3-c3=1<=r+c1+c2=2+1+2=5? Yes -> S={p1,p2,p3}

B4-c4=2<=r+c1+c2+c3=5+1=6? Yes -> S={p1,p2,p3,p4}

B5-c5=1<=r+c1+c2+c3+c4=6+2=8? Yes -> S={p1,p2,p3,p4,p5}

Since all the processes are in the sequence, the state is safe, and required resources can be granted

**Task 7.**  Consider what data structures are necessary to manage stack?

We need maxAddress, minAddress, showing boundaries of stack segment, we are to have current state – StackPointer, pointing to the last written in the stack element. To request stack write, we give input parameter – required\_size\_of\_stack, and check, whether StackPointer - required\_size\_of\_stack >= minAddress. If this holds, StackPointer is decreased: StackPointer = StackPointer - required\_size\_of\_stack; otherwise query is rejected.

**Task 8.**  Consider what data structures are necessary to manage heap memory?

We may represent list of free regions by linked list of free regions:

|  |  |
| --- | --- |
| Free tag | Size |
| Address of previous block | Address of next block |
| Free region | |
| Size | Free tag |

Used regions of heap may be represented by

|  |  |
| --- | --- |
| Busy tag | Size |
| Used region | |
| Size | Busy tag |

where Size is the size of the whole region including used in it tags, sizes, etc.

We may have also HeapHeader, pointing to the first free region. When malloc(size) query comes, allocator may scan list of free regions, find fitting, and allocate (part of) it to application. Application will get pointer to the 1st byte after “Size”. When free(pointer) will be launched, before this pointed address, allocator will find size of allocated to application region, and by analysis of neighboring regions, it will be able to merge it with free regions.

**Task 9.** How, in C language, to query 100 byte of memory in the Data Segment?

Char a[100];

**Task 10.** How, in C language, to make global variable **a** to be not visible to function f()?

void f(){

..

}

int a;

**Task 11.** What are the benefits of multiprogramming with fixed partitions and absolute addresses of compilation?

Simplicity of implementation

**Task 12.** What problems do we have with variable partitions?

Fragmentation of memory, garbage collection, problem of allocation (FF, BF, LF algorithms)

**Task 13.** What is the difference between paged and segmented virtual memory (VM)?

The latter uses regions of different sizes, and paged virtual memory uses slots of the same size (usually, 4-8 K)

**Task 14.** How is achieved high performance in VM systems in spite of necessity to have 1 or 2 additional memory accesses on each memory access?

Fast cache memory is used to keep most recently used descriptors

**Task 15.**  What is it: page fault?

Interruption due to the absence of required virtual page in the physical memory

**Task 16.**  What is the reason of in –page fragmentation?

Size of region necessary to application may be not a multiple of the page size

**Task 17.**  What is the essence of First Fit algorithm?

It tries to allocate first met free region of memory to the requestor

**Task 18.**  What is the difference between LRU and FIFO?

Least Recently Used algorithm selects for replacement page which was not used for the longest period of time. FIFO replaces page, staying in memory for the longest period of time (oldest). For example, page P1 stays in memory from moment 00:00 (current time, suppose, is 01:24), it is old; P2 stays in memory from 01:15, it is young. If P1 last time was used at 01:22, and P2 – at 01:19, then LRU will replace P2, and FIFO will replace P1 in the case of physical memory with only 2 slots and necessity to load P3 at the current moment.

**Task 19.**  What is the essence of the elevator algorithm?

Queries are served in the order of meeting them while magnetic head moves inward or outward

**Task 20.**  (Laplante, 2nd Ed., Ex. to Ch. 5, No 6, p. 139) Build a Moore FSA to accept words cab, cob, cat, cot but no others from the alphabet L={a,b,c,o,t}

**Task 21.** (Laplante, 2nd Ed., Ex. to Ch. 5, No. 7, p. 139) Build a Moore FSA accepting all words except cab, cob, cat, cot, alphabet is L={a,b,c,o,t}

**Task 22.** (Laplante, 2nd Ed., Ex. to Ch. 5, No. 9, p. 139) Build a Moore FSA accepting all strings with three consecutive zeroes. Alphabet L={0,1}.

**Task 23.** Build a Mealy FSA for a coffee machine. Assume that the machine after reception of a coin dispenses a cup and asks for a choice of beverage: coffee, decaffeinated coffee, or tea. After that it dispenses beverage. Then, it asks for sugar and dispenses it, if necessary. Then it asks for milk, and dispenses it, if necessary. After that, it can serve the next client.

