Interprocess Communication (Part 2)

Operating Systems

Assoc. Prof. Milos Jovanovik, PhD

Semaphores

- Solutions with busy waiting are hard to generalize for complex problems
- **Semaphores are used as a general synchronization tool**
- A Semaphore consists of:
  - Variable: **N**
  - List of processes: **queue**
  - The semaphore is accessed with:
    - Method **wait** (**down**, **sleep, acquire**): checks N, and depending on its value, blocks a given process (inserts it in the waiting queue) or permits access to the critical section;
    - Method **signal** (**up**, **wakeup, release**): the value of N is updated and a sleeping process from queue is awaken;
- Example: Semaphore s(5);

Semaphore Functionality

- The process calls **wait** before the critical region and either receives a permission to continue or gets blocked, depending on the value of the variable **N** in the semaphore.
  - If N is 0, it blocks;
  - If N is not 0, it decrements value by 1 and exits the wait call --> entering the critical region;
- The process calls **signal** after the critical region and frees one process to continue through the semaphore.
  - The variable N of the semaphore is incremented;
  - The blocked process becomes ready and it is positioned in the ready queue (after that, the CPU is assigned by the scheduler);

Semaphore Properties

- The **variable N** determines how many processes can work simultaneously – semaphore bandwidth
  - Example: 3 processes for accessing the printer;

- These methods are **executed atomically**, i.e. when one instruction is executed in the semaphore, no other process is granted access to it

Semaphore Implementation

```
typedef struct {
 int N;
 struct process *queue;
} semaphore;

void wait(semaphore S){
 //down, sleep, acquire
 S.N--;
 if (S.N < 0) {
  add process to S.queue;
  block();
 }
}

void signal(semaphore S){
 //up, wakeup, release
 S.N++;
 if (S.N ≤ 0) {
  remove a process P from S.queue;
  wakeup(P);
 }
}
```

Using Semaphores as a General Synchronization Tool

- Count semaphores:
  - Integer values
  - Unbounded
- Binary semaphores:
  - Integer values 0 and 1
  - Easier to implement
  - a.k.a mutex locks

- A count semaphore can be implemented as a binary semaphore

Using Semaphores

- **N** processes are sharing the same semaphore **mutex**.
- Each $\mathbf{P_i}$ is organized as:

```
do {
  wait(mutex);
  // critical section
  signal(mutex);
  // non-critical section
} while(1)
```

Example

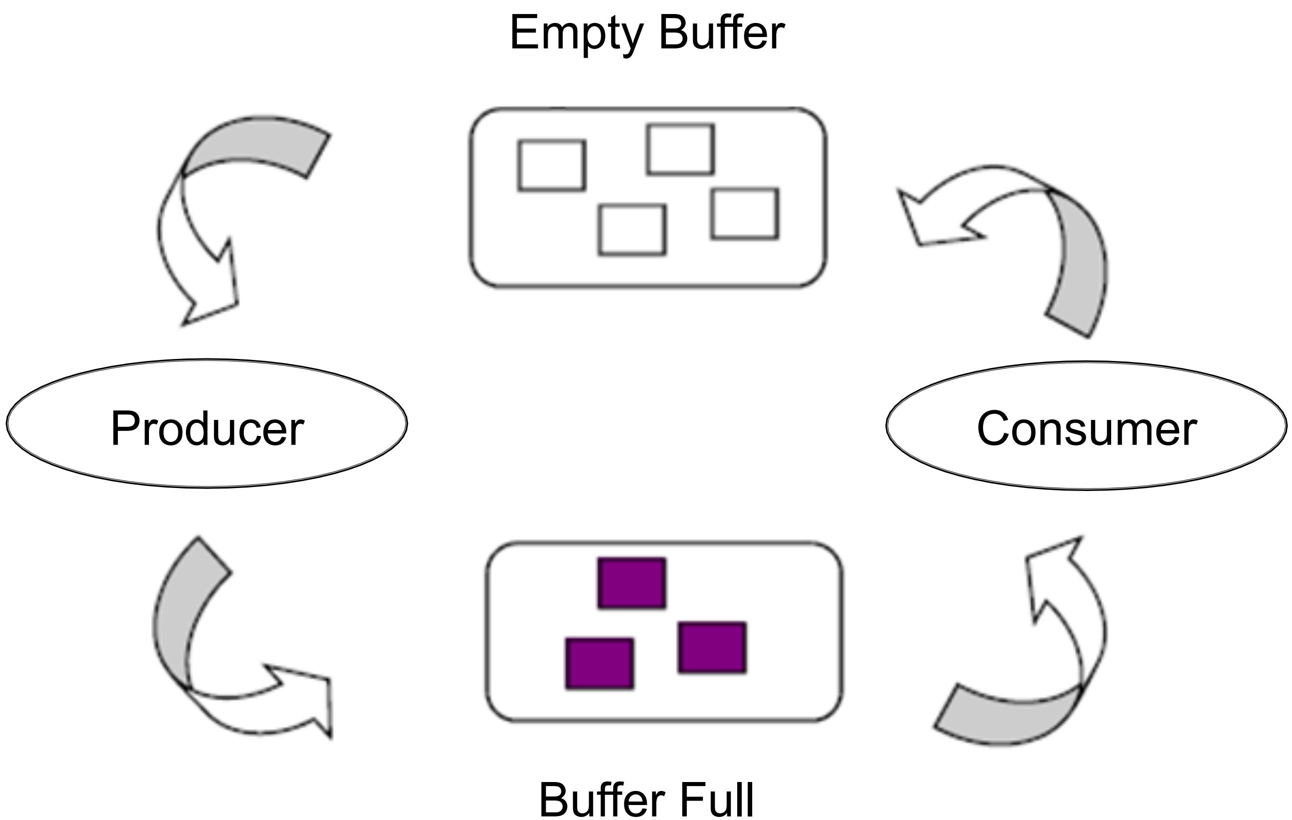
- Two concurrent processes:
  - P_1 has a statement block SB_1, a P_2 has SB_2
- Suppose that SB_2 should be executed after SB_1
  - This should work regardless of which process executes first in the CPU;
- P_1 and P_2 use a shared semaphore (sync=0) in their code:

```
// P1 code
SB1;
signal(sync);
```

```
// P2 code
wait(sync);
SB2;
```

- Since sync==0, P_2 will execute S_2 only after P_1 calls signal(sync), which is after S_1

Producer – Consumer



Producer – Consumer: Semaphores

```
#include <prototypes.h>
#define N   100     // number of slots
typedef int semaphore;
semaphore mutex=1, empty=N, full=0;

void producer (void)  {
   int item;
   while (TRUE)  {
    produce_item(&item);   // next item
    Down (&empty);  //  full? decrement empty
    Down (&mutex);  // enter cr. segment
    enter_item (item);
    Up (&mutex);    // exit cr. segment
    Up (&full);        // increment full
    }   }

void consumer (void)  {
   int item;
   while (TRUE)  {
    Down (&full);  //  full? decrement
    Down (&mutex);  // cr. segment
    remove_item(&item);   // take item
    Up (&mutex);   // cr. segment
    Up (&empty);    //increment empty
    consume_item(item);   // print item
    }   }
```

4. Higher-Level Synchronization Primitives

- They come as high-level language support, i.e. they have to be supported by the compiler
- It is assumed that processes have local data and code that operates on these data
- The local data is accessed only by the program encapsulated by the process, i.e. one process cannot directly access the local data of some other process
- Processes can share global data

Monitors

- A set of procedures, variables and data structures grouped in special types of modules

- A process can call the monitor at any moment, but it cannot directly access the internal data structures of the monitor with procedures declared outside of the monitor

Monitor Description

- A Monitor is a construction similar to objects in high-level program languages, such as C++, Java, ...
- It consists of:
  - Variables (a, b, c)
  - Methods (fn_1(…), fn_2(…); fn_m(…);)
  - Initial code (constructor)
  - Conditional variables (x,y,z)

```
Monitor M {
    int a, b, c ;
    condition x, y, z ;
    fn1(…);
    fn2(…);
    …
    fnm(…);
    { init code }
}
```

Monitor Semantics

- Only one method (process) in the monitor can be active at a given moment – mutual exclusion
- The programmer does not have to use explicit synchronization – this is now left to the compiler
- By defining all critical sections as procedures (functions) of the monitor, there is no concurrent execution in them

Conditional Variables

- By adding conditional variables, we enrich the process management

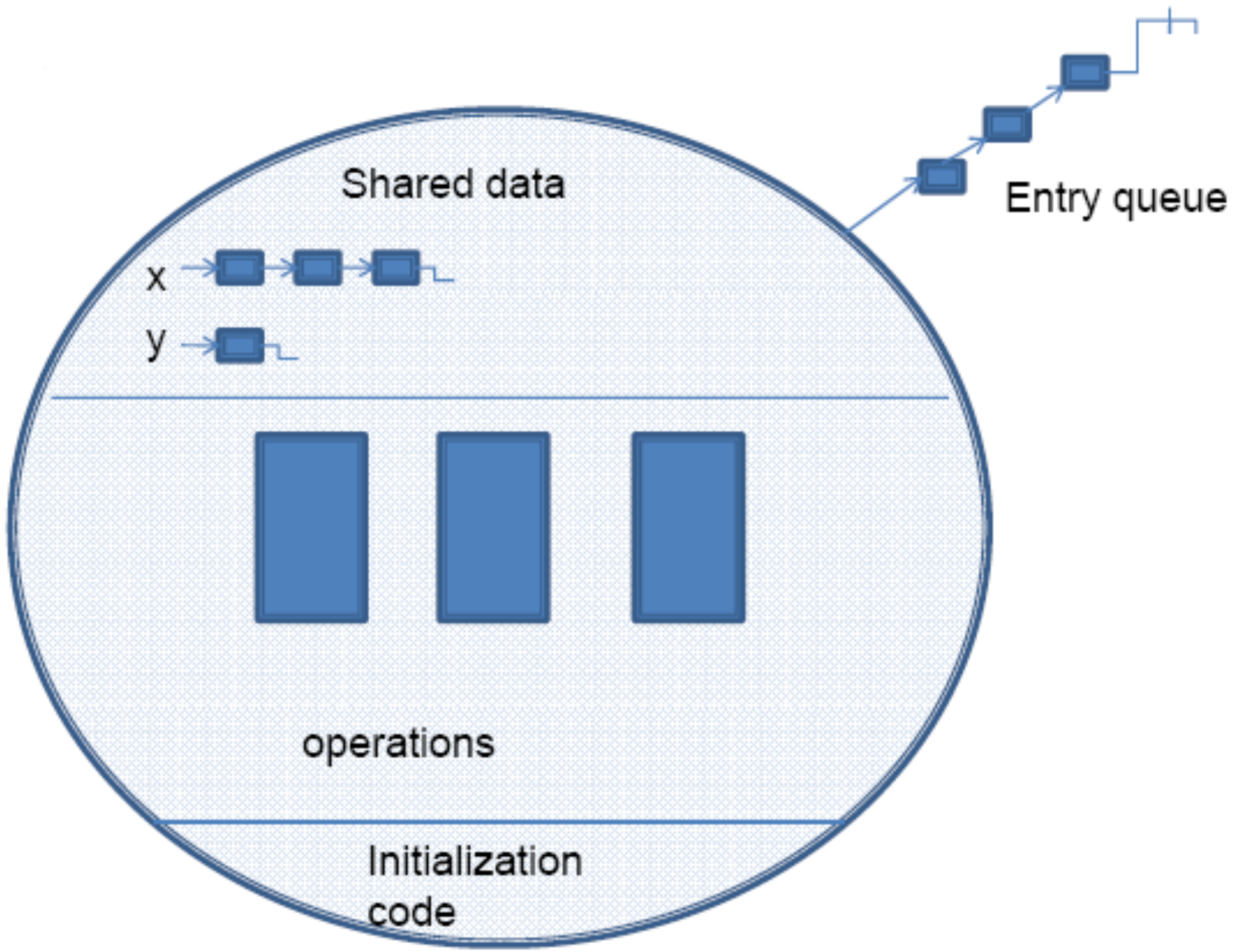
- These variables can have two operations:
  - **x.wait** – the calling process is blocked (waits in a queue), until some other method (process) signals the variable with ...
  - **x.signal** – unblocks the process that is waiting in a given variable (... but, there will be two active processes in the monitor ...)

Conditional Variables

- The process that produced the signal operation, leaves the monitor (it blocks), while the process that waited on the conditional variable x, is activated

- If the conditional variable is signalled, but no other process waits on it, the signal is lost

Monitor


Shared data
x
y
Entry queue
operations
Initialization
code

Example

*fn*1(…)

…

x.wait // P1 blocks

*fn*4(…)

…

x.signal // P2 blocks

// P1 resumes

// P1 finishes

// P2 resumes

Producer – Consumer: Monitors

```
Monitor BoundedBuffer
char buf [N];
int count;
int tail, head, item;
condition empty, full;
// initialization
count = tail = head =0;
```


```
Remove(item) {
If (count == 0)
        empty.wait;
item = buf [head++ % N];
count--;
 full.signal;
}

void  consume() {
 while (TRUE) {
  Remove(item);
  consume_item(&item);
  }
}
```

```
Enter(item) {
if (count == N) full.wait;
buf [tail++ % N]=item;
count++;
empty.signal;
}

void produce () {
 while (TRUE) {
  produce_item(&item);
   Enter(item);
   }
}
```

Task 1

Assume that you have **three concurrent threads** which are part of a single process, and they execute the procedures A, B and C.

Provide the value of the **global variable x** (*write an integer value*), as well as the corresponding sequence of execution (*write the names of the procedures in a sequence of execution, without commas, e.g. BAC*), **after all three threads finish**.

```
#include <prototypes.h>
typedef int semaphore;
semaphore sA=1, sB=0, sC=0;
int x = 0;
void A()
{
 Down(&sA);
 x+=2;
 Up($sB);
}
```

```
void B()
{
 Down(&sB);
 x-=1;
 Up($sC);
}
void C()
{
 Down(&sC);
 x+=2;
 Up($sA);
}
```

The value of the global variable x is ☐, and the sequence of execution is ☐.

Task 1: Solution

Assume that you have **three concurrent threads** which are part of a single process, and they execute the procedures A, B and C.

Provide the value of the **global variable x** (*write an integer value*), as well as the corresponding sequence of execution (*write the names of the procedures in a sequence of execution, without commas, e.g. BAC*), **after all three threads finish**.

```
#include <prototypes.h>
typedef int semaphore;
semaphore sA=1, sB=0, sC=0;
int x = 0;
void A()
{
 Down(&sA);
 x+=2;
 Up($sB);
}
```

```
void B()
{
 Down(&sB);
 x-=1;
 Up($sC);
}
void C()
{
 Down(&sC);
 x+=2;
 Up($sA);
}
```

The value of the global variable x is 3, and the sequence of execution is ABC.

Task 2

What is the minimal value you can use to initialize the semaphore `s` in the code below, so that you avoid a **deadlock**? The two threads T1 and T2 call the functions in the given order:

T1: calls "A B C A B"
T2: calls "C B A A B"

The functions A, B and C are defined as follows:

```
...
semaphore s=[ ] ;

void A() {
  wait(&s);
  ...
}
void B() {
  signal(&s);
  ...
}
void C() {
  wait(&s);
  ...
}
...
```

The value of the semaphore after both threads finish is: [ ].

6	4	1	2	0	3	5

Task 2: Solution

What is the minimal value you can use to initialize the semaphore `s` in the code below, so that you avoid a **deadlock**? The two threads T1 and T2 call the functions in the given order:

T1: calls "A B C A B"
T2: calls "C B A A B"

The functions A, B and C are defined as follows:

```
...
semaphore s= 3 ;

void A() {
  wait(&s);
  ...
}
```

```
void B() {
  signal(&s);
  ...
}
void C() {
  wait(&s);
  ...
}
...
```

The value of the semaphore after both threads finish is: 1 .

6	4	1	2	0	3	5

Questions?