

Synchronization-III

Semaphores

Didem Unat

Lecture 11

COMP304 - Operating Systems (OS)

Mutex Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
  
release() {  
    available = true;  
}
```

- Software interface for locks
- Calls to **acquire()** and **release()** must be atomic
 - implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - And also called a **spinlock**

Semaphores

- We want to be able to write more complex constructs
 - need a language to do so. We define semaphores which we assume are atomic operations.
- Semaphores are more general synchronization tools
 - Operating System Primitive
 - Two standard atomic operations modify semaphore variable S: wait() and signal()

WAIT (S):

```
while ( S <= 0 );  
S = S - 1;
```

SIGNAL (S):

```
S = S + 1;
```

- As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

Critical Section for n Processes

- Shared semaphore:
semaphore mutex = 1; //initial value
- Process P_i :

```
do {  
    wait(mutex)  
    critical section  
    signal(mutex)  
    remainder section  
} while (true);
```

Shared Account Balance Example

```
semaphore mutex = 1;
```

```
proc_0() {  
    . . .  
    /* Enter the CS */  
    wait(mutex);  
    balance += amount;  
    signal(mutex);  
    . . .  
}
```

```
proc_1() {  
    . . .  
    /* Enter the CS */  
    wait(mutex);  
    balance -= amount;  
    signal(mutex);  
    . . .  
}
```

//CS stands for critical section

Semaphore as a general synchronization tool

- Provides **mutual exclusion**

Semaphore S = 1; // initialized to 1 or initialized to # of resources

wait (S);

Critical Section

signal (S);

- **Counting** semaphore – integer value can range over an unrestricted domain
 - For example: resources in the hardware: semaphore is initialized to number of printers
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - This is the same as **mutex locks**

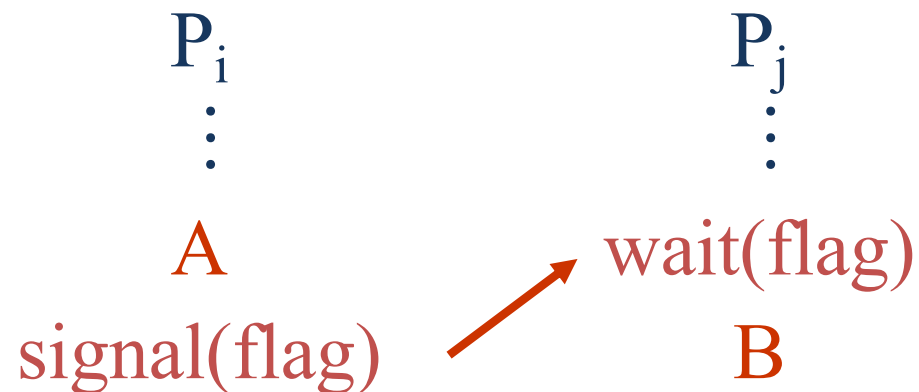
Semaphore as a general synchronization tool

- Semaphores can be used to force synchronization (precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at beginning.

For example, here we want P1 to execute before P2.

- Execute B in P_j **only after** A is executed in P_i
- Use semaphore flag initialized to **0**

Code:



Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

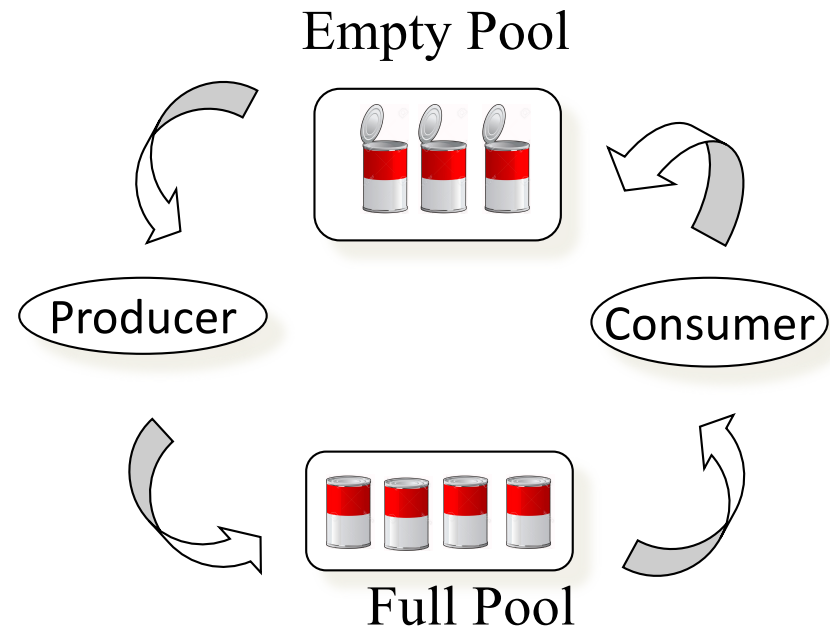
These classical problems are used for testing newly proposed synchronization methods.

Bounded-Buffer Problem

- Buffer size: buffer can hold **n** items
- Binary semaphores
semaphore mutex;
- Counting semaphores
semaphore full, empty;

Initially:

full = 0, empty = n, mutex = 1



This is the same producer / consumer problem as before. But now we'll do it with signals and waits. Remember: a **wait decreases** its argument and a **signal increases** its argument.

Bounded-Buffer Problem

- Empty and Full are counting semaphores

```
producer:
do {
    /* produce an item in nextp */
    wait (empty);          /* Do action      */

    /* add nextp to buffer */

    signal (empty);

} while(TRUE);
```

```
consumer:
do {
    wait (full);

    /* remove an item from buffer to nextc */

    signal (full);

    /* consume an item in nextc */
} while(TRUE);
```

Signals are wrong!
Producer (consumer)
needs to wake up the
consumer (producer)!

Bounded-Buffer Problem

- **Mutex** is binary semaphore, Empty and Full are counting semaphores

```
producer:
do {
    /* produce an item in nextp */
    wait (empty);          /* Do action      */

    /* add nextp to buffer */

    signal (full);

} while(TRUE);
```

Does this work for multiple producers and consumers?

Only works for one producer and consumer. Need the mutex to prevent multiple producers writing into the same buffer.

```
consumer:
do {
    wait (full);

    /* remove an item from buffer to nextc */

    signal (empty);

    /* consume an item in nextc */
} while(TRUE);
```

Bounded-Buffer Problem

- **Mutex** is binary semaphore, Empty and Full are counting semaphores

```
producer:
do {
    /* produce an item in nextp */
    wait (empty);          /* Do action */
    wait (mutex);          /* Buffer guard*/

    /* add nextp to buffer */

    signal (mutex);
    signal (full);

} while(TRUE);
```

```
consumer:
do {
    wait (full);
    wait (mutex);

    /* remove an item from buffer to nextc */

    signal (mutex);
    signal (empty);

    /* consume an item in nextc */
} while(TRUE);
```

We need a mutex in addition to empty and full semaphores because multiple producers (hence consumers) are operating on the same buffer

Semaphores

- **Spinlocks** (mutexes) are useful in a system since no context switch is required
- A disadvantage of mutex solutions so far:
 - they all require **busy waiting**.
- To overcome busy waiting → **blocking a process**

No busy waiting (blocking) Semaphores

- With each semaphore there is an associated waiting queue:
 - Keeps list of processes waiting on the semaphore
- Two operations:
 - **Block** – place the process invoking the operation on the appropriate waiting queue if semaphore == false
 - **Wakeup** – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Blocking Semaphores

```
typedef struct {  
    int    value;  
    struct process *list; /* list of processes waiting on S */  
} SEMAPHORE;
```

```
SEMAPHORE s;  
wait(s) {  
    s.value = s.value - 1;  
    if ( s.value < 0 ) {  
        add this process to s.list;  
        block ();  
    }  
}
```

```
SEMAPHORE s;  
signal(s) {  
    s.value = s.value + 1;  
    if ( s.value <= 0 ) {  
        remove a process P from s.list;  
        wakeup(P);  
    }  
}
```

Block – place the process invoking the operation on the appropriate waiting queue if semaphore is not available

Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Deadlock and Starvation

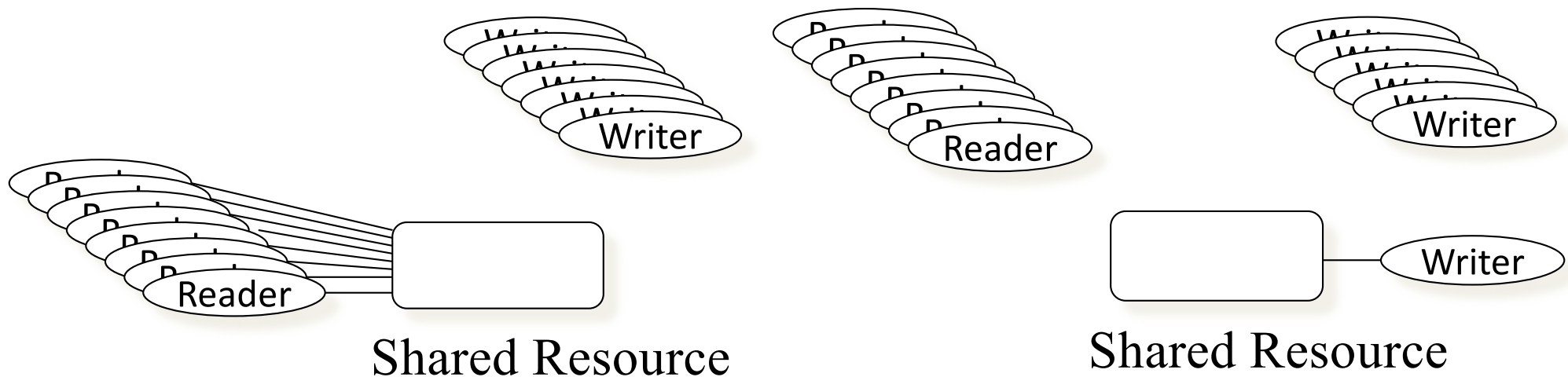
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
\vdots	\vdots
signal(S);	signal(Q);
signal(Q)	signal(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- May occur if we add and remove processes from the list in LIFO order or based on priority.

Readers and Writers Problem

- A data set is shared among a number of concurrent processes
 - **Readers** – only read the data set; they do **not** perform any updates
 - **Writers** – can both read and write.
- Problem
 - Allow multiple readers to read at the same time.
 - Only one single writer can access the shared data at a time.



Readers/Writers Problem

Locks:

- are **shared** (for the readers) and **exclusive** (for the writer).

Two possible (contradictory) guidelines can be used:

- No reader is kept waiting unless a writer holds the lock (the readers have precedence).
 - First readers problem
- If a writer is waiting for access, no new reader gains access (writer has precedence).
 - Second readers problem

Can starvation occur on either of these rules?

First-Readers Problem

Favoring readers over writers

```
semaphore rdwrt = 1;  
semaphore cntmutex = 1;  
int readcount = 0;
```

Writer:

```
do {  
    wait( rdwrt );  
    /*    writing is performed    */  
    signal( rdwrt );  
  
} while(TRUE);
```

Reader:

```
do {  
    wait( cntmutex );           /* Allow 1 reader in entry*/  
    readcount = readcount + 1;  
    if readcount == 1 then  
        wait(rdwrt );           /* 1st reader locks rdwrt */  
    signal( cntmutex );  
  
    /*    reading is performed    */  
  
    wait( cntmutex );  
    readcount = readcount - 1;  
    signal( cntmutex );  
    if readcount == 0 then  
        signal(rdwrt );         /*last reader frees writer */  
  
} while(TRUE);
```

First-Readers Problem

Favoring readers over writers

```
semaphore rdwrt = 1;
semaphore cntmutex = 1;
int readcount = 0;
```

Writer:

```
do {
    wait( rdwrt );
    /* writing is performed */
    signal( rdwrt );

} while(TRUE);
```

Reader:

```
do {
    wait( cntmutex );
    readcount = readcount + 1;
    if readcount == 1 then
        wait( rdwrt );
        signal( cntmutex );

        /* reading is performed */

    wait( cntmutex );
    readcount = readcount - 1;
    signal( cntmutex );
    if readcount == 0 then
        signal( rdwrt );
        /*last reader frees writer */
} while(TRUE);
```

A reader might be
reading while a writer is
writing at the same
time or two writers can
perform their writes!

How can that happen?

First-Readers Problem

Correct Implementation, Favoring readers over writers

```
semaphore rdwrt = 1;
semaphore cntmutex = 1;
int readcount = 0;
```

Writer:

```
do {
    wait( rdwrt );
    /*    writing is performed    */
    signal( rdwrt );

} while(TRUE);
```

Reader:

```
do {
    wait( cntmutex );
    readcount = readcount + 1;
    if readcount == 1 then
        wait( rdwrt );
    signal( cntmutex );

    /*    reading is performed    */

    wait( cntmutex );
    readcount = readcount - 1;
    if readcount == 0 then
        signal( rdwrt );
    signal( cntmutex );

} while(TRUE);
```

/* Allow 1 reader in entry*/

Assume there are two readers left (r1 and r2). r1 may be interrupted before it checks readcount == 0. The second reader(r2) decrements the readcount from 1 to 0 and both will check readcount==0 and get true, then both send a signal(wrt) allowing two writers or one reader and one writer in the system.

Question

- Many events on campus take place in SGKM. Assume that the show room can only allow N many people. As soon as N audience are in the room, the staff will not accept another incoming audience until an existing audience leaves the event.
- Explain how semaphores can be used by the staff to limit the number of people in the SGKM event room.
- Show your pseudo-code.

Solution

- Semaphore is initialized to the number of allowable people in SGKM. When a reservation is accepted, the `acquire()` method is called; when someone leaves the event, the `release()` method is called. If the system reaches the number of allowable people, subsequent calls to `acquire()` will block until an existing person leaves the event and the `release` method is invoked.

Reading

- Read Chapter 6
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley
 - Jerry Breecher