Synchronization

Last modified: 11.03.2020

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
  …
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to **0**. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

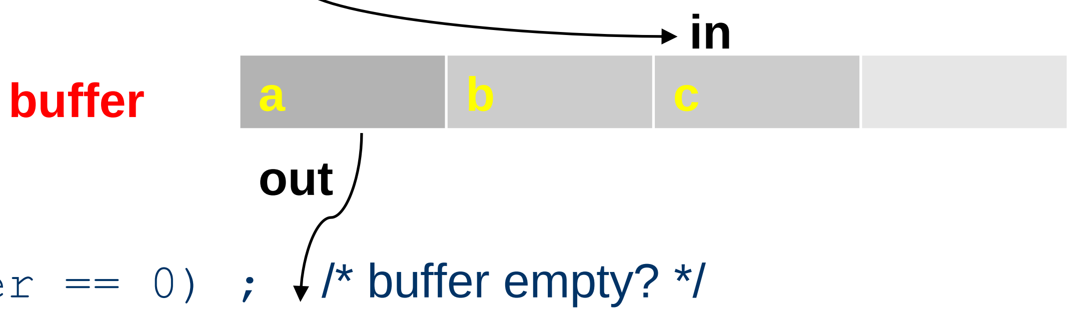
Producer-consumer (and a buffer)

- **Shared data**

```
#define BUFFER_SIZE 4
typedef struct {        . . . } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0, counter = 0;
```

- **Producer**

```
while (true) {
        /* produce an item in next_produced  */
        while (counter == BUFFER_SIZE)  ;/* buffer full? */
        buffer [in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```



- **Consumer**

```
while (true) {
        while (counter == 0)  ;  /* buffer empty? */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next_consumed */
}
```

Producer-consumer - implementation

- For the presented solution of the producer-consumer problem the following operations :

  ```
  counter++;
  counter--;
  ```

  have to be performed **atomically**, i.e. their execution has to be carried out without interruption.

- Example implementations of the operations **counter++** and **counter-**

```
register1 = counter                 register2 = counter
register1 = register1 + 1           register2 = register2 - 1
counter = register1                 counter   = register2
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

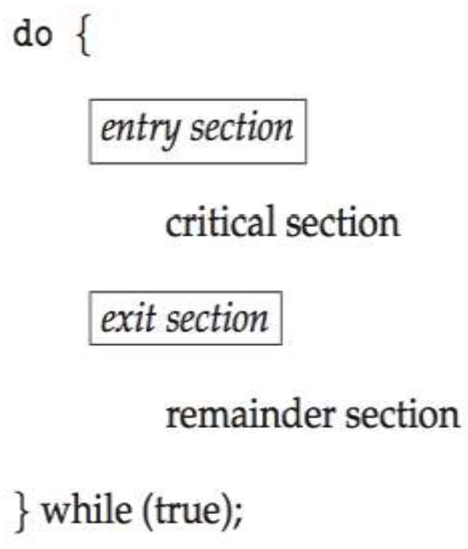
- Consider this execution interleaving with “count = 3” initially:

```
S0: producer execute register1 = counter          {register1 = 3}
S1: producer execute register1 = register1 + 1    {register1 = 4}
S2: consumer execute register2 = counter          {register2 = 3
S3: consumer execute register2 = register2 – 1    {register2 = 2}
S4: producer execute counter = register1          {counter = 4}
S5: consumer execute counter = register2          {counter = 2}
```

- Outcome depends on the order of writes to `counter` (**race condition**).
- To prevent uncertainty synchronization is used.

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Two-process solution
- Assumed: the **`load`** and **`store`** machine-language instructions are atomic
- The two processes share two variables:
  - **`int turn; /* indicates whose turn it is to enter CS */`**
  - **`Boolean flag[2]; /* flag[i] = `*`true`*** implies that process $\mathbf{P_i}$ is ready */
- **The algorithm**

```
flag[i] = true;
turn = j;
while (flag[j] && turn = = j);

        critical section

flag[i] = false;
        remainder section
```

- One can prove that the three critical section requirements are met.

Synchronization hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **atomic** == **non-interruptible**
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

- This is a correct solution to the critical-section problem only for 2 processes, and when acquire can be successfully executed only by one of them.
- For more than 2 processes the code provides mutual exclusion only.

Mutual exclusion using swap

- **Definition**:

```
int swap ((boolean &a, boolean &b) {
    boolean temp = a; a = b; b = temp;
}
```

Atomic change of value for two variables a and b

- **Solution**

  Shared data: `boolean lock=false;`

  Process P_i

```
    boolean key= true; /* local variable */
            . . .
    do {
        swap(lock,key);
                } while(key) ;
      /* critical section */
    . . .
    lock = false;
    /* remainder section*/
```

Mutual exclusion using test_and_set

- **Definition**:

```
boolean test_and_set (boolean *target){
            boolean rv = *target;
            *target = TRUE;   return rv:
     }
```

  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to “**TRUE**”.

- **Solution**

Shared Boolean variable **lock**, initialized to **FALSE**

```
          . . .
  while (test_and_set(&lock))
      ; /* do nothing */

/* critical section */

  lock = FALSE;
            /* remainder section */
          . . .
```

Bounded-waiting Mutual Exclusion with test_and_set

```
/* entry section */
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
...
 /* exit section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

/* remainder section */
...
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than mutex locks) for process to synchronize their activities.
- Semaphore ***S*** state – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S.value <= 0)
       ; // busy wait
    S.value--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S.value++;
}
```

Critical section for *n* processes

- Shared
  - `semaphore mutex;`
  - Initially `mutex.value = 1`
- Process P_i

```
      …

     wait(mutex);

  /* critical section */
      …

     signal(mutex);

     /* remainder section */
  …
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to the next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

Implementation with no Busy waiting (cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      /* add this process to S->list; */
       . . .
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      /* remove a process P from S->list; */
       . . .
      wakeup(P);
   }
}
```

Semaphore Usage

- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`sync`" initialized to 0

```
P1:
    S1;
    signal(sync);

P2:
    wait(sync);
    S2;
```

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

```
        P0                          P1
    wait(S);                    wait(Q);
    wait(Q);                    wait(S);
      ...                         ...
    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
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Two types of semaphores

- **Binary semaphore** – integer value can range only between 0 and 1
- **Counting semaphore** – integer value can range over an unrestricted domain. Counting semaphore **S** can be implemented with two binary semaphores:

Data structures:

```
binary-semaphore S1, S2;
int C;
```

Initialization: `S1.value = 1;` `S2.value = 0;` `C =` initial value of ***S***

Code:

```
wait(S){
    wait(S1);
    C--;
    if (C < 0) {
        signal(S1);
        wait(S2);
    }
    signal(S1);
}
```

```
signal(S){
        wait(S1);
         C ++;
         if (C <= 0) signal(S2);
         else signal(S1);
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded Buffer Problem

Producer sends data to consumer using a shared buffer that can hold **n** data items

- Shared data

```
semaphore full, empty, mutex;
SemInit(full,0); SemInit(empty,n); SemInit(mutex,1);
```

Producer code

```
do {
/* create data item*/
. . .
  wait(empty);
  wait(mutex);
/* store data item in buffer */
  ...
  signal(mutex);
  signal(full);
} while (1);
```

Consumer code

```
 do {
wait(full);
wait(mutex);
  /* remove data item from the buffer */
                ...
        signal(mutex);
              signal(empty);
/* use data item retrieved from the buffer*/
        . . .
} while (1);
```

Readers-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 the same time
- Several variations of how readers and writers are considered:
  - **First** variation – no reader kept waiting unless writer has permission to use shared object
  - **Second** variation – once writer is ready, it performs the write ASAP
  - Both may have starvation leading to even more variations
  - Problem is solved on some systems by kernel providing reader-writer locks

Readers-Writers Problem (Cont.)

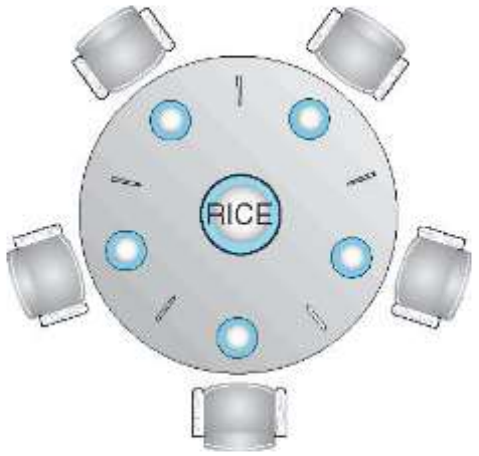
- **Shared Data**
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0
- **The structure of a writer process**

```
do {
     wait(rw_mutex);
          ...
  /* writing is performed */
          ...
     signal(rw_mutex);
 } while (true);
```

- **The structure of a reader process**

```
do {
      wait(mutex);
       read_count++;
       if (read_count == 1)
        wait(rw_mutex);
      signal(mutex);
          ...
       /* reading is performed */
          ...
     wait(mutex);
       read_count--;
       if (read_count == 0)
             signal(rw_mutex);
     signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
      wait (chopstick[i] );
       wait (chopStick[ (i + 1) % 5] );

                 //  eat

       signal (chopstick[i] );
       signal (chopstick[ (i + 1) % 5] );

                  //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- Note: deadlock prevention might not prevent starvation

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation **are possible**.

Monitors (*)

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```

- For more information – see the textbook

Synchronization implementations

- Windows
- Linux
- Pthreads

POSIX conforming systems

- UNIX System V interface –**semaphore sets**
- Newer interface of “**POSIX semaphores**
  - **named semaphores**
  - **unnamed semaphores**

- **Mutexes**
- **Condition variables**
- **Barriers**
- **Read-write locks**

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer locks
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

The goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is **write-ahead logging**
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - **<T_i starts>** written to log when transaction T_i starts
  - **<T_i commits>** written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record **<checkpoint>** to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms , e.g. two-phase locking protocol, provide serializability control

T0	**T1**
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T0	**T1**
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)