
LUT
University
LAND
OF THE
CURIOUS

JANUARY 17, 2023

OPERATING SYSTEMS AND SYSTEMS PROGRAMMING (CT30A3370) 6 CREDITS

Venkata Marella

CHAPTER 6: PROCESS SYNCHRONIZATION

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

BACKGROUND

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 count that keeps track of the number of full buffers. Initially, count 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

```
while (true) {

        /* produce an item and put in nextProduced */
         while (count == BUFFER_SIZE)
                        ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

CONSUMER

```
while (true)  {
        while (count == 0)
              ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        /*  consume the item in nextConsumed
 }
```

RACE CONDITION

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

CRITICAL-SECTION PROBLEM

- To design a protocol that the processes can use to cooperate

```
Do {

    Entry section

    Critical section

    Exit section

    Remainder section

}while(TRUE);
```

General structure of a typical process Pj

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
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

SYNCHRONIZATION HARDWARE

- Many systems provide hardware support for critical section code
- 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-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TESTANDSET INSTRUCTION

- Definition:

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

SOLUTION USING TESTANDSET

- Shared boolean variable lock., initialized to false.
- Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

SEMAPHORE

- Synchronization tool that is less complicated
- Semaphore *S* – integer variable
- Two atomic standard operations modify *S: wait()* and *signal()*
  - Originally called *P()* and *V()*
- Can only be accessed via two indivisible (atomic) operations
  - *wait (S) {*

    while S <= 0

    ; // no-op

    S--;

    }
  - *signal (S)*

    { S++;

    }
- Can be implemented without busy waiting

USAGE AS GENERAL SYNCHRONIZATION TOOL

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - *Semaphore S;   // initialized to 1*
  - *wait (S);*

    Critical Section

    signal (S);

USAGE AS GENERAL SYNCHRONIZATION TOOL(2)

- P1 has a statement S1, P2 has S2
- Statement S1 to be executed before S2

P1

```
S1;
Signal(S);
```

Question: What's the initial value of S?

P2

```
Wait(S);
S2;
```

SEMAPHORE IMPLEMENTATION

- Must guarantee that no two processes can execute *wait ()* and *signal ()* on the same semaphore at the same time
- Thus, 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 semaphore has two data items:
  - value (of type integer)
  - pointer to a linked-list of PCBs.
  - Typedef struct{
    - Int value;
    - Struct process *list;

    } semaphone;

- Two operations (provided as basic system calls):
  - 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.

SEMAPHORE IMPLEMENTATION WITH NO BUSY WAITING (CONT.)

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block(); }
}
```

- Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P); }
}
```

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);
.	.
.	.
.	.
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.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

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

BOUNDED-BUFFER PROBLEM

- *N* buffers, each can hold one item
- *How many Semaphores do we need?*
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0, counting full items
- Semaphore empty initialized to the value N, counting empty items.

BOUNDED BUFFER PROBLEM (CONT.)

- The structure of the producer process

```
while (true) {

        //   produce an item
    wait (empty);
    wait (mutex);

        //  add the item to the buffer

    signal (mutex);
    signal (full);
}
```

BOUNDED BUFFER PROBLEM (CONT.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                // remove an item from buffer

        signal (mutex);
        signal (empty);

                // consume the removed item

}
```

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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1, to ensure mutual exclusion when readcount is updated.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

READERS-WRITERS PROBLEM (CONT.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

READERS-WRITERS PROBLEM (CONT.)

- The structure of a reader process

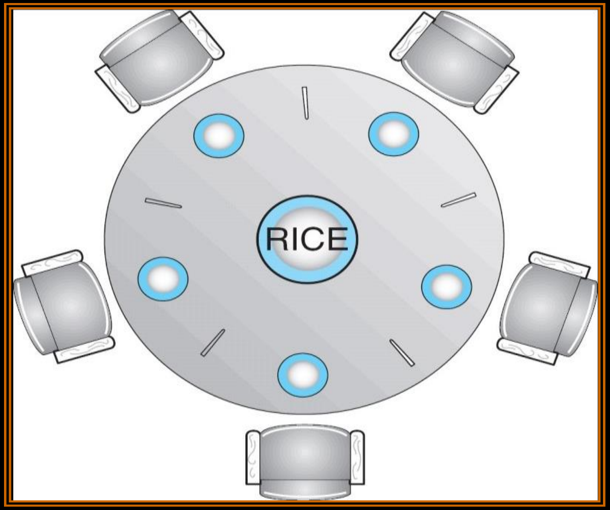
```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1) wait (wrt) ;
        signal (mutex)

                // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount == 0) signal (wrt) ;
        signal (mutex) ;
}
```

"Locking" the wrt semarphore, rather than "waiting"

Reason is that wrt is initialized to "1"

"Unlocking" the wrt semarphore, rather than "signaling"

DINING-PHILOSOPHERS PROBLEM



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

DINING-PHILOSOPHERS PROBLEM (CONT.)

- The structure of Philosopher *i*:

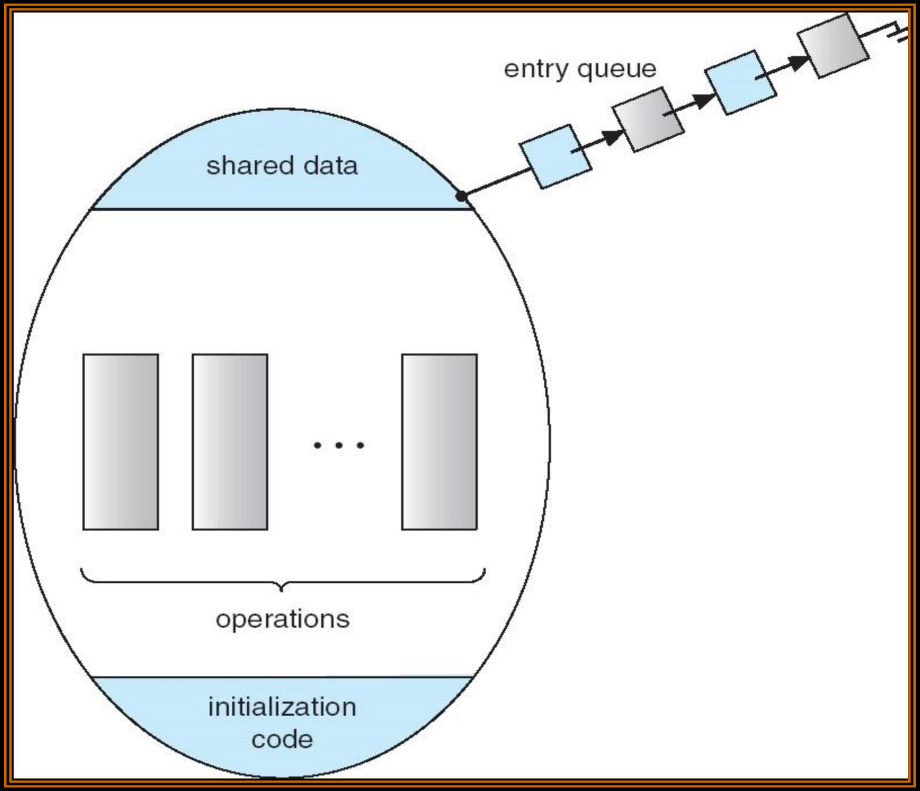
```
While (true)  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

             //  eat

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


             //  think


}
```

MONITORS

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

SCHEMATIC VIEW OF A MONITOR

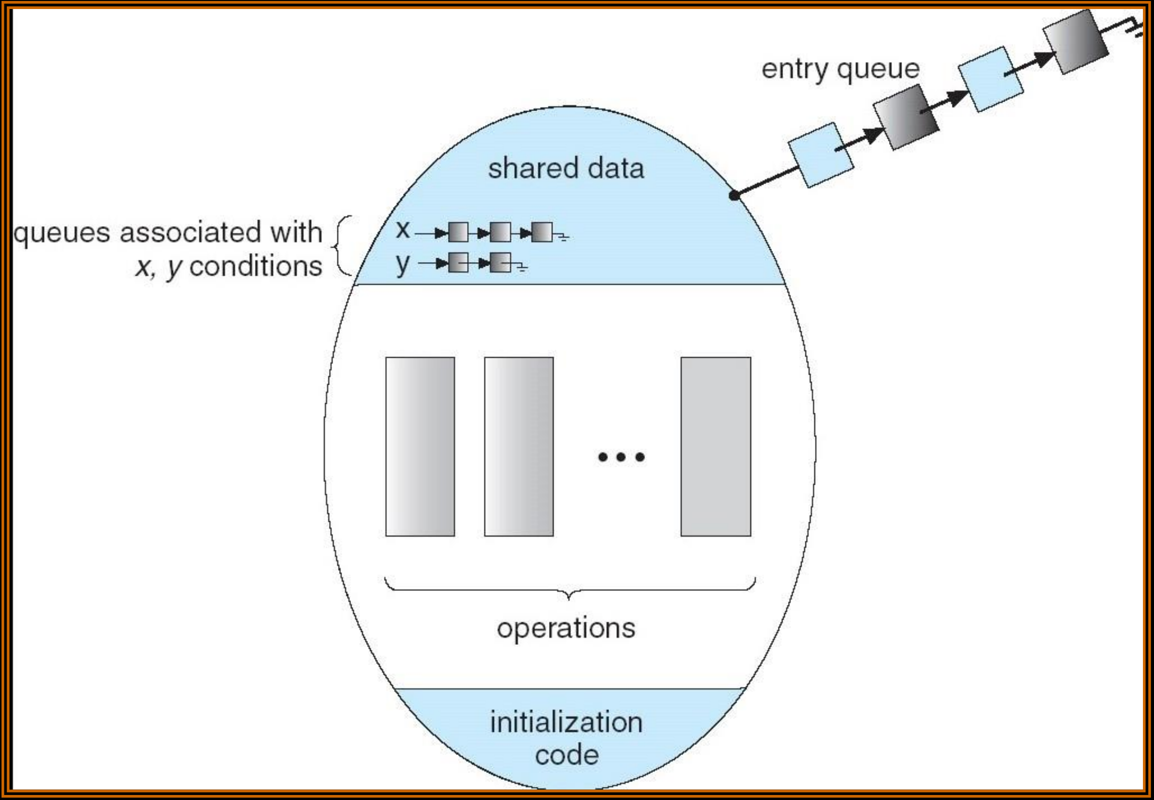


CONDITION VARIABLES

- *condition x, y;*

- Two operations on a condition variable:
- *x.wait ()* – a process that invokes the operation is suspended.
- *x.signal ()* – resumes one of processes (if any) that invoked *x.wait ()*

MONITOR WITH CONDITION VARIABLES



SOLUTION TO DINING PHILOSOPHERS

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5]; //philosopher i can delay herself when unable to get
    chopsticks
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
        state[i] = THINKING;
            // test left and right neighbors
         test((i + 4) % 5);
         test((i + 1) % 5);
     }
```

SOLUTION TO DINING PHILOSOPHERS (CONT)

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
              self[i].signal () ;
         }
 }

 initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
}
}
```

SOLUTION TO DINING PHILOSOPHERS (CONT)

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

  dp.pickup (i)

  EAT

  dp.putdown (i)

- When the left and right philosophers, **self[(i+4)%5]** and **self[(i+1)%5]** continue to eat, **self[i]** may starve.

MONITOR IMPLEMENTATION USING SEMAPHORES

- Variables

```
semaphore mutex;  // (initially  = 1), entry protection
semaphore next;  // (initially  = 0), signaling process may suspend themselves.

int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
        body of F;

    …
if (next-count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore "next" is introduced, on which the signaling process may suspend themselves.

MONITOR IMPLEMENTATION

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

If someone has been waiting, wake her up because I'll be entering the waiting state.

No one else waiting in the monitor. I'm going to block. Allow someone else to enter the monitor now.

MONITOR IMPLEMENTATION

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
      next-count++;
      signal(x-sem);
      wait(next);
      next-count--;
}
```

This is the signaling process. It will wait on the “next” semaphore.

SEMAPHORE VS. MONITOR

Semaphores	***Condition Variables***
Can be used anywhere in a program, but should not be used in a monitor	Can only be used in monitors
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	Wait() always blocks the caller.
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.
If Signal() releases a blocked thread, the caller and the released thread ***both*** continue.	If Signal() releases a blocked thread, the caller yields the monitor blocks(Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.

SYNCHRONIZATION EXAMPLES

- Solaris
- Windows XP
- Linux
- Pthreads

SOLARIS SYNCHRONIZATION

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments (page 218)
  - The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running.
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

WINDOWS XP SYNCHRONIZATION

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks (busy-waiting semaphore) on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events and timer
  - An event acts much like a condition variable
  - A timer is used to notify one thread if a specified amount of time has expired
- Dispatcher object from signaled state to nonsignaled state

LINUX SYNCHRONIZATION

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

PTHREADS SYNCHRONIZATION

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
  - read-write locks

- Non-portable extensions include:
  - spin locks
  - semaphores

Thank you

LUT University