Operating Systems Concepts

Process Synchronization: Semaphores, Mutex

CS 4375, Fall 2025
Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu
November 05, 2025

Summary

- Process Synchronization
    - Multiprogramming: multiple processes or threads may require consensus to work together
        - How do we achieve that? What issues may occur for shared data?
    - Race Condition
    - Critical-Section Problem
        - Different Solutions

Agenda

- Semaphores
- Classic Problems of Synchronization
    - Bounded Buffer Problem
    - Readers and Writers Problem
    - Dining Philosophers Problem
- Monitors
- Condition Variables
- Sleeping Barber Problem

Solution to Critical-Section Problem

- Summary of implementations of mutual exclusion
  - Implementation 1 - disabling hardware interrupts
    - NO: race condition avoided, but can crash the system!
  - Implementation 2 - simple lock variable (unprotected)
    - NO: still suffers from race condition
  - Implementation 3 - indivisible lock variable (TSL)
    - YES: works, but requires hardware support

    This will be the basis for “mutexes”
  - Implementation 4 - no-TSL toggle for two threads
    - NO: race condition avoided inside, but lockup outside
  - Implementation 5 - Peterson’s no-TSL, no-alternation
    - YES: works in software, but has processing overhead

Solution to Critical-Section Problem

- Problem: All implementations (2-5) rely on busy waiting
  - "Busy waiting" means that the process/thread continuously executes a tight loop until some condition changes
  - Busy waiting is bad ☹️
    - **Waste of CPU time**- the busy process is not doing anything useful, yet remains "READY" instead of "BLOCKED"
    - **Paradox of inversed priority**- by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and liberating B! (B is working against its own interest)

- We need for the waiting process to block, not keep idling!

Semaphores

- Synchronization tool for critical section problem
- Semaphore *S* - Integer variable
- Can only be accessed through two standard operations:
  - wait() and signal()
  - P() and V()
    - Proberen/test ; Verhogen/increase (in Dutch)
- Classical implementation (using busy-waiting): (Without interruption)

`wait (S) {`
`  S--;`
`  while (S <= 0); // loop`
`}`
	`signal (S) {`
	` S++;`
	`}`

Semaphores Without Busy-Waiting

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

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

- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.

Semaphores Without Busy-Waiting

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


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

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
    // remove process P from the
    // waiting queue S->list
    wakeup(P);
    }
}
```

Semaphores as Synchronization Tool

- Counting semaphore - Integer value can range over an unrestricted domain
- Binary semaphore - Integer value can range only between 0 and 1
  - Also known as **mutex locks**
- Provides mutual exclusion

```
Semaphore S; //initialized to N
wait(S);
/* critical section */
signal(S);
```

```
Lock S; //initialized to 1
acquire(S);
/* critical section */
release(S);
```

Bounded Buffer Problem

- Shared buffer with ***N* slots** to store at most ***N* items**
- Producer processes data items and puts into the buffer
- Consumer gets the data items from the buffer
- Variable **`empty`** keeps number of empty slots in the buffer
- Variable **`full`** keeps number of full items in the buffer

Bounded Buffer Problem - Solution 1

- **Implementation 1** - 1 semaphore

Producer Process

```
int empty = N, full = 0;
while (true) {
    /* produce an item */
    while (empty == 0); // loop
    wait(mutex);
    // add the item to the buffer
    empty--; full++;
    signal(mutex);
}
```

Consumer Process

```
while (true) {
    wait(mutex);
    if (full > 0) {
    // remove item from buffer
    full--; empty++;
    }
    signal(mutex);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 1

- **Implementation 1** - 1 semaphore

Producer Process

```
int empty = N, full = 0;
while (true) {
     /* produce an item */
     while (empty == 0); // loop
     wait(mutex);
     // add the item to the buffer
     empty--; full++;
     signal(mutex);
}
```

Consumer Process

```
while (true) {
     wait(mutex);
     if (full > 0) {
     // remove item from buffer
     full--; empty++;
     }
     signal(mutex);
     /* consume the item */
}
```

Bounded Buffer Problem - Solution 1

- **Implementation 1** - 1 semaphore

Consumes non-existing item!

Consumer Process

```
while (true) {
    wait(mutex);
    if (full > 0) {
    // remove item from buffer
    full--; empty++;
    }
    signal(mutex);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 2

- **Implementation 2** - 1 semaphore

Producer Process

```
int empty=N, full=0;
while (true) {
    /* produce an item */
    while (empty == 0); // loop
    wait(mutex);
    // add the item to the buffer
    empty--; full++;
    signal(mutex);
}
```

Consumer Process

```
while (true) {
    while (full == 0); // loop
    wait(mutex);
    // remove item from buffer
    full--; empty++;
    signal(mutex);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 2

- **Implementation 2** - 1 semaphore

Producer Process

```
int empty=N, full=0;
while (true) {
    /* produce an item */
    while (empty == 0); // loop
    wait(mutex);
    // add the item to the buffer
    empty--; full++;
    signal(mutex);
}
```

Consumer Process

```
while (true) {
    while (full == 0); // loop
    wait(mutex);
    // remove item from buffer
    full--; empty++;
    signal(mutex);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 2

- **Implementation 2** - 1 semaphore

Mutual exclusion is not preserved!

Consumer Process

```
while (true) {
    while (full == 0); // loop
    wait(mutex);
    // remove item from buffer
    full--; empty++;
    signal(mutex);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 3

- **Implementation 3** - 2 semaphore

Producer Process

```
int empty = N, full = 0;
while (true) {
    /* produce an item */
    wait(empty);
    // add the item to the buffer
    signal(full);
}
```

Consumer Process

```
while (true) {
    wait(full);
    // remove item from buffer
    signal(empty);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 3

- **Implementation 3** - 2 semaphore

Producer Process

```
int empty = N, full = 0;
while (true) {
    /* produce an item */
    wait(empty);
    // add the item to the buffer
    signal(full);
}
```

Consumer Process

```
while (true) {
    wait(full);
    // remove item from buffer
    signal(empty);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 3

- **Implementation 3** - 2 semaphore

Mutual exclusion is not preserved!

Consumer Process

```
while (true) {
    wait(full);
    // remove item from buffer
    signal(empty);
    /* consume the item */
}
```

Bounded Buffer Problem - Solution 4

- **Implementation 4** - 3 semaphore
  - Semaphore mutex to access the buffer
    - Initialized to 1
  - Semaphore full (number of full buffers)
    - Initialized to 0
  - Semaphore empty (number of empty buffers)
    - Initialized to N

Bounded Buffer Problem - Solution 4

- **Implementation 4** - 3 semaphore

Producer Process

```
int empty = N, full = 0;
while (true) {
    /* produce an item */
    wait(empty);
    wait(mutex);
    // add the item to the buffer
    signal(mutex);
    signal(full);
}
```

Consumer Process

```
while (true) {
    wait(full);
    wait(mutex);
    // remove item from buffer
    signal(mutex);
    signal(empty);
    /* consume the item */
}
```

Bounded Buffer Problem

- **Implementation 1: Using 1 semaphore**
- **Implementation 2: Using 1 semaphore**
- **Implementation 3: Using 2 semaphores**
- **Implementation 4: Using 3 semaphores**

Readers and Writers Problem

- Multiple ***Readers*** and ***Writers*** concurrently accessing the same database.
- Multiple ***Readers*** accessing at the same time → OK
- When there is a ***Writer*** accessing, there should be no other process accessing at the same time

Readers and Writers Problem

Reader process

```
while (true) {
     wait(mutex);
     readercount++;
     if (readercount == 1)
          wait (wrt_mutex);
     signal(mutex);
     /* read from database */
     wait(mutex);
     readercount--;
     if (readercount == 0)
          signal (wrt_mutex);
     signal(mutex);
}
```

Writer Process

```
while (true) {
     wait(wrt_mutex);
     /* write to database */
     signal(wrt_mutex);
}
```

Dining Philosophers Problem

- Five philosophers spend their time eating and thinking
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and

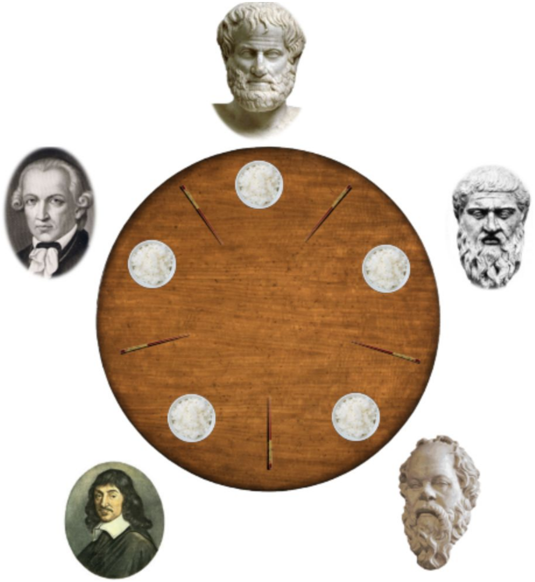
  five chopsticks set between the plates.

- Eating the spaghetti requires the use of

  two chopsticks, which the philosophers

  pick up one at a time.

- Philosophers do not talk to each other.
- Semaphores chopstick [5] are initialized to 1



Source: https://sphof.readthedocs.io/test2.html

Dining Philosophers Problem

Philosopher *i* process

```
while (true) {
     wait( chopstick[i] );
     wait( chopstick[(i+1)%5] );
     /* eat */
     signal( chopstick[(i+1)%5] );
     signal( chopstick[i] );
     /* think */
}
```

Dining Philosophers Problem

Philosopher *i* process

```
while (true) {
     wait( chopstick[i] );
     wait( chopstick[(i+1)%5] );
     /* eat */
     signal( chopstick[(i+1)%5] );
     signal( chopstick[i] );
     /* think */
}
```

Mutual exclusion is ensured, however it may result in deadlock!

Dining Philosophers Problem - Deadlock

- Deadlock happens when 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.

P1	P2
`wait(S);`	`wait(Q);`
`.`	`.`
`wait(Q);`	`wait(S);`
`.`	`.`
`.`	`.`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Dining Philosophers Problem

- To prevent deadlock:
  - Allow a philosopher to pick up his chopsticks only if both chopsticks are available (*i.e.* in critical section)
  - Use an asymmetric solution: An odd numbered philosopher picks up first his left chopstick and then his right chopstick; An even numbered philosopher first picks up his right chopstick, then his left chopstick.
  - Exercise: Write the algorithms for the above solutions

Semaphores - Wrong Use of Operations

- Semaphores A and B, initialized to 1

P1	P2
`wait(A);`	`wait(B);`
`wait(B);`	`wait(A);`

  - Deadlock

- signal(mutex) … wait(mutex)
  - Violation of mutual exclusion
- wait(mutex) … wait(mutex)
  - Deadlock
- Omitting of wait(mutex) or signal(mutex) (or both)
  - Violation of mutual exclusion or deadlock

Semaphores

- Semaphores are inadequate in dealing with deadlocks
- Do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken.
- Mostly used in low level code (e.g. operating systems)
- The trends in programming language development, is towards more structured forms of synchronizations such as **monitors**.

Monitor

- A high-level abstractions that provides 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(...){...}
    ...
}
```

- A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition.

Monitor Example

- As a simple example, consider a monitor for performing transactions on bank account

```
monitor account {
    int balance := 0
    function withdraw(int amount) {
        if (amount < 0) then error "Amount may not be negative"
        else if (balance < amount) then error "Insufficient funds"
        else balance := balance - amount
    }
    function deposit(int amount) {
        if (amount < 0) then error "Amount may not be negative"
        else balance := balance + amount
    }
}
```

Monitor Example

```
class Account {
      private lock myLock;
      private int balance := 0
      invariant (balance >= 0)
      public method boolean withdraw(int amount)
            precondition (amount >= 0) {
            myLock.acquire();
            try:
                  if (balance < amount)  then return false
                  else { balance := balance - amount ;  return
                  true }
            finally:
                  myLock.release();
            }
      public method deposit(int amount)
            precondition (amount >= 0) {
            myLock.acquire();
            try:
                  balance := balance + amount
            finally:
                  myLock.release();
}
```

By hiding the details of the synchronization code from the programmer, mutual exclusion is inherently provided by monitors, so programmer does not need to manage these locks manually.

Condition Variables

- For many applications, mutual exclusion is not sufficient
- A thread may need to wait until some condition holds true
- We don't want to use busy waiting…
- Condition variables queue threads until a certain condition is met (non-blocking, non-busy-waiting)
- Two operations on a condition variable x:
  - x.wait()– a thread invoking this operation is suspended
  - x.signal()– resumes one of the threads (if any) that invoked x.wait()

- If no thread was suspended, x.signal() operation has no effect (unlike semaphores)

Dining Philosophers Problem - Solution with Monitor

- Implementation using Monitors and condition variables

```
monitor DP {
     enum {THINKING; HUNGRY, EATING} state[5] ;
     condition self[5];    // to delay philosopher when he is hungry
                           // but unable to get chopsticks
     initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
     }
     void pickup (int i) {
          state[i] = HUNGRY;
          test(i);          // only if both neighbors are not eating
          if (state[i] != EATING) self[i].wait();
     }
}
```

Dining Philosophers Problem - Solution with Monitor

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

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

Dining Philosophers Problem - Solution with Monitor

- Main thread will run the initialization code

```
...
DiningPhilosophers.initialization_code();
...
```

- Each Dining Philosopher thread will run the `pickup` and `putdown` functions for Philosopher thread *i*:

```
...
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
...
```

Dining Philosophers Problem - Solution with Monitor

- No two philosophers eat at the same time
- No deadlock
- But starvation can occur!
    - Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
    - How can we prevent this? Exercise!

Sleeping Barber Problem

- Based upon a hypothetical barber shop with **one barber**, **one barber chair**, and **a number of chairs** for waiting customers
- When there are no customers, the barber sits in his chair and sleeps
- As soon as a customer arrives, he either **awakens** the barber or, if the barber is cutting someone else's hair, **sits down** in one of the vacant chairs
- If all of the chairs are **occupied**, the newly arrived customer simply **leaves**

Sleeping Barber Problem - Solution

- Use **three semaphores**: one for any **waiting customers**, one for the **barber** (to see if he is idle), and a **mutex**
- When a customer arrives, he attempts to **acquire** the **mutex**, and waits until he has succeeded.
- The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if **none** of these are **empty**, the customer checks again later (in a **loop**).
- Otherwise the customer takes a seat – thus **reducing** the number available (a **critical section**).

Sleeping Barber Problem - Solution

- The customer then signals the barber to **awaken** through his **semaphore**, and the mutex is released to allow other customers (or the barber) the ability to **acquire** it
- If the barber is not free, the customer then waits. The barber sits in a perpetual waiting loop, being awakened by any **waiting** customers. Once he is awoken, he signals the **waiting** customers through their **semaphore**, allowing them to get their hair cut one at a time.

Sleeping Barber Problem - Solution

```
Semaphore Customers                 // to wait for available customers
Semaphore Barber                    // to wait for available barber
Lock accessSeats                    // mutex lock, to change seat availability
int NumberOfFreeSeats
```

The Barber(Thread):

```
while (true) {               //runs in an infinite loop
    Customers.wait()         // tries to acquire a customer
                             // if none is available he's going to sleep
    accessSeats.acquire()// at this time he has been awaken
                             // -> want to modify the number of available seats
    NumberOfFreeSeats++     // one chair becomes free
    Barber.signal()         // the barber is ready to cut
    accessSeats.release()// we don't need the lock on the chairs anymore
    /* barber cuts hair */
}
```

Sleeping Barber Problem - Solution

The Customer(Thread):

```
needCut = true                              // the customer decides that they need haircut
while (needCut) {                           // as long as the customer is not cut
    accessSteats.acquire()                  // tries to get access to the chairs
    if (NumberOfFreeSeats > 0) {            // if there are any free seats
        NumberOfFreeSeats --                // sitting down on a chair
        Customers.signal()                  // notify the barber, who's waiting
                                            // for a customer
        accessSeats.release()               // don't need to lock the chairs anymore
        Barber.wait()                       // now it's this customers turn, but
                                            // needs to wait if the barber is busy
        needCut = false
    } else {                                // there are no free seats (tough luck!)
        accessSeats.release()               // but don't forget to
                                            // release the lock on the seats
    }
}
```

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo

Announcement

- Homework 4
  - Due on November 10th at 11.59PM
- Quiz 5
  - Released on black board
  - Due tomorrow November 6th, 11.59PM
- Please check the final exam schedule
  - December 12th, 10 AM - 12.45 PM