Chapter 4

PROCESS SYNCHRONIZATION

Part 2

Content

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors

Learning Outcomes

- Understand concurrency is an issue in operating systems and multithreaded applications
- Know the concept of a *critical region*.
- Understand how mutual exclusion of critical regions can be used to solve concurrency issues
- Including how mutual exclusion can be implemented correctly and efficiently.
- Be able to identify and solve a *producer consumer bounded buffer* problem.
- Understand and apply standard synchronization primitives to solve synchronisation problems.

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most n – 1 items in buffer at the same time. A solution, where all N buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer

Atomic operations

- In concurrent programming, an operation (or set of operations) is **atomic** if it appears to the rest of the system to occur at once without being interrupted.
- Other words used synonymously with atomic are: linearizable, indivisible or uninterruptible.
- Additionally, atomic operations commonly have a succeed-or-fail definition—they either successfully change the state of the system, or have no apparent effect.

Non-atomic operations

- When the compiler translates the **shared++** and **shared--** statements, these will be translated to a series of machine instructions (depending on the CPU architecture). On a load/store architecture (for example MIPS):
- **shared++**
  1) **load** shared from memory into CPU register
  2) **increment** shared and save result in register
  3) **store** result back to memory
- **shared--**
  1) **load** shared from memory into CPU register
  2) **decrement** shared and save result in register
  3) **store** result back to memory

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0,	0	0			
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	lw	$t0, BALANCE	1
			1	addi	$t0, $t0, -1	0
			0	sw	$t0, BALANCE	0

If the instructions are executed in this order, the increment and decrement cancel each other and the resulting **BALANCE** is **0**.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
			0	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1
sw	$t0, BALANCE	1	+1			

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **+1**.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **-1**.

Concurrency Example

- count is a global variable shared between two threads.After increment and decrement complete, what is the value of count?

```
void increment ()
{
      int t;
      t = count;
      t = t + 1;
      count = t;
}
```

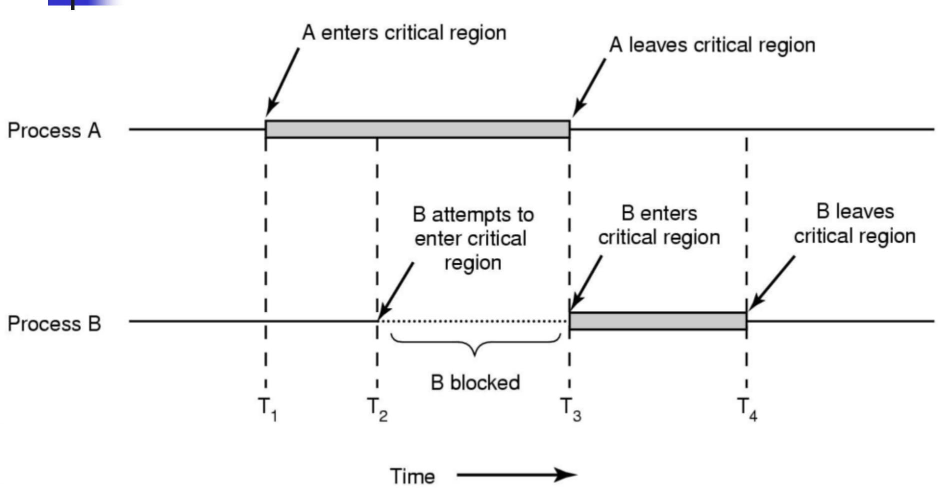
```
void decrement ()
{
  int t;
  t = count;
  t = t - 1;
  count = t;
}
```

Critical Region

- We can control access to the shared resource by controlling access to the code that accesses the resource. →A *critical region* is a region of code where shared resources are accessed.
  - Variables, memory, files, etc…
- Uncoordinated entry to the critical region results in a race condition → Incorrect behaviour, deadlock, lost work,…

Đinh Công Đoan 11

Critical Regions



Mutual exclusion using critical regions

Đinh Công Đoan 12

Identifying critical regions

- Critical regions are regions of code that:
  - ✓ Access a shared resource,
  - ✓ and correctness relies on the shared resource not being concurrently modified by another thread/ process/entity
- Example critical regions

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;
while (1) {
while (counter == BUFFER_SIZE)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```


Bounded-Buffer

- Consumer process

```
item nextConsumed;
while (1) {
while (counter == 0)
; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
}
```

Bounded-Buffer

- The statements
  counter++;
  counter--;
  must be performed atomically.
- Atomic operation means an operation that completes in its entirety without interruption
- The statement "**count++**" may be implemented in machine language as:
  - **register1 = counter**
    register1 = register1 + 1
    counter = register1

- The statement "**count - -**" may be implemented as:
  - **register2 = counter**
    register2 = register2 – 1
    counter = register2
- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

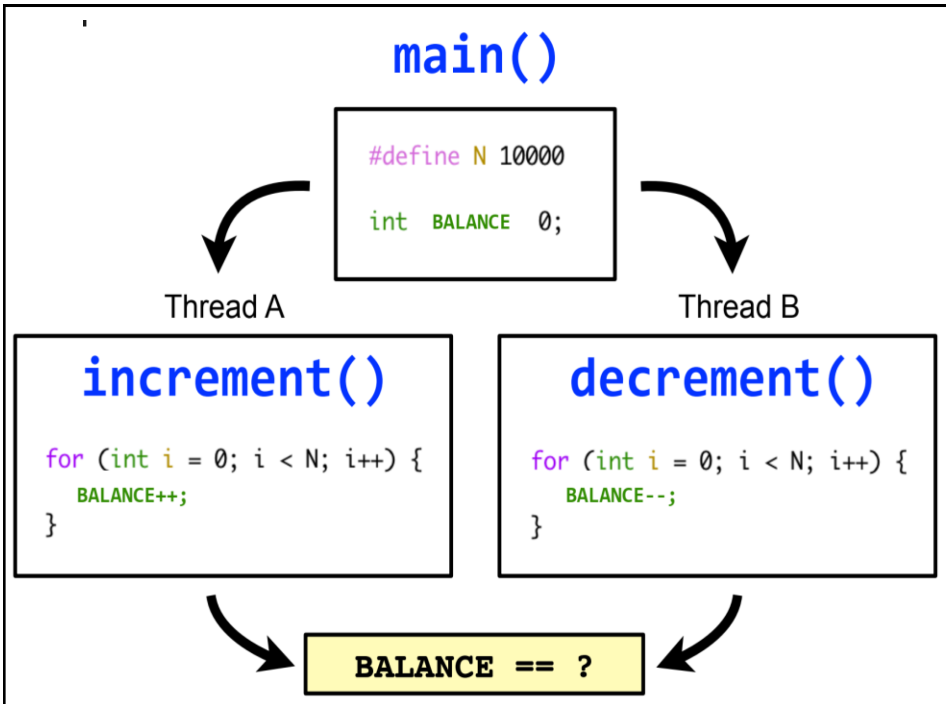
- Assume **counter** is initially 5. One interleaving of statements is
  - producer: **register1 = counter** (register1 = 5)
  - producer: **register1 = register1 + 1** (register1 = 6)
  - consumer: **register2 = counter** (register2 = 5)
  - consumer: **register2 = register2 – 1** (register2 = 4)
  - producer: **counter = register1** (counter = 6)
  - consumer: **counter = register2** (counter = 4)
- The value of **count** may be either 4 or 6, where the correct result should be 5.

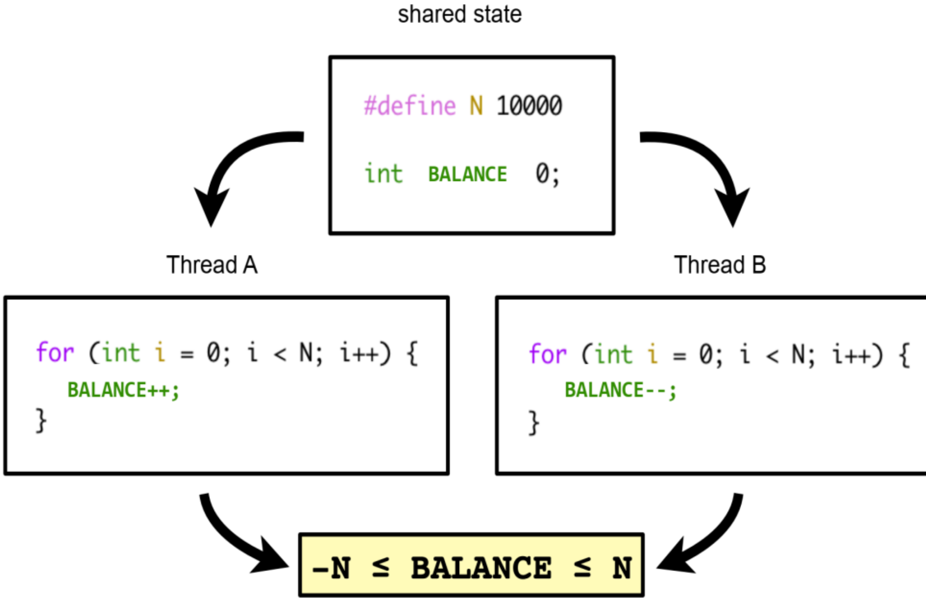
Race condition

- A race condition or race hazard is the behaviour of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events.
- It becomes a bug when events do not happen in the intended order.
- The term originates with the idea of two signals racing each other to influence the output first

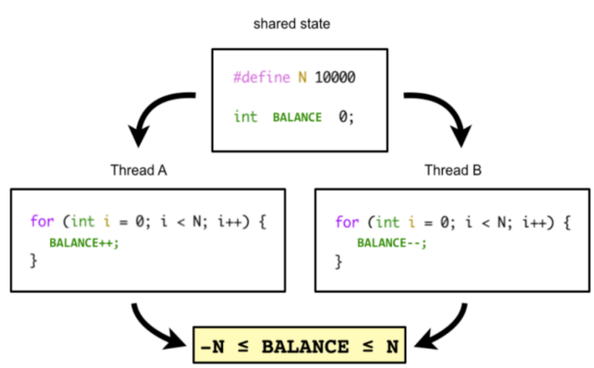
Data race

- A data race occurs when two instructions from different threads access the same memory location and:
  - ✓ at least one of these accesses is a write
  - ✓ and there is no synchronization that is mandating any particular order among these accesses





Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results if executed several times.



Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results if executed several times.

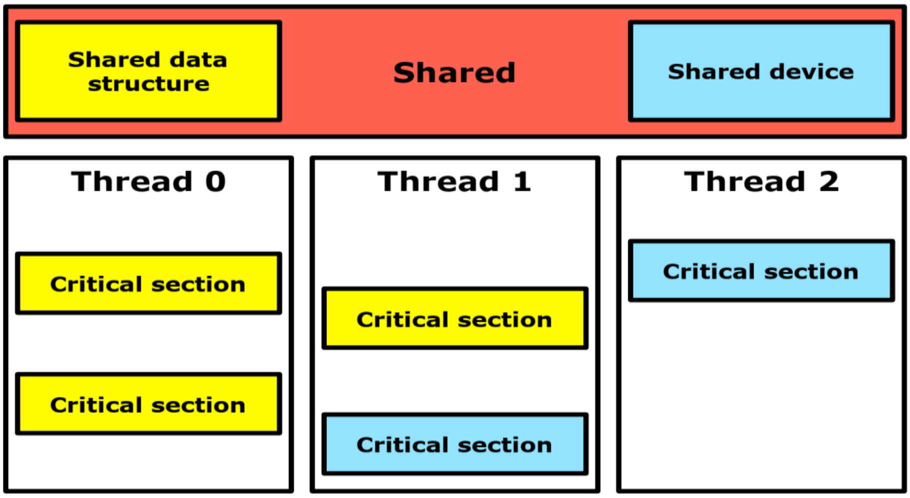
Testing and debugging	When a program is executing concurrently, there are many different execution paths. Testing and debugging concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Race condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.
-

Critical section

- Part of a program that should not be concurrently executed by more than one of the program's concurrent processes or threads at a time.
- Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that does not allow multiple concurrent accesses.
- By carefully controlling which variables are modified inside and outside the critical section, concurrent access to that state is prevented.



Three threads accessing a shared data structure (yellow) and a shared device (blue).

The critical – section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Đinh Công Đoan 28

- N processes all competing to use shared data.
  - Structure of process Pi ---- Each process has a code segment, called the critical section, in which the shared data is accessed.

repeat
  entry section　　　/* enter critical section */
  critical section　　/* access shared variables */
  exit section　　　　/* leave critical section */
  remainder section　/* do other work */
until false

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Solution to Critical-Section Problem

- 1. **Mutual Exclusion**. If process Pi 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.

Properties of critical sections

- Assume that each process/thread executes at a nonzero speed. No assumption concerning relative speed of the N processes. In the following discussion the term **task** is used to mean a concurrent unit of execution such as a process or thread

Mutual Exclusion

If task T_i is executing in its critical section, then no other tasks can be executing in their critical sections.

Bounded Waiting

A bound must exist on the number of times that other tasks are allowed to enter their critical sections after a task has made a request to enter its critical section and before that request is granted.

Progress

If no task is executing in its critical section and there exist some task that wish to enter their critical section, then the selection of the task that will enter the critical section next cannot be postponed indefinitely.

More?

Deadlock

Fairness

Starvation

We will come back to this …

Solution of the Critical Problem

- Software based solutions
- Hardware based solutions
- Operating system based solution
- Structure of Solution
  - do {

    entry section

    critical section

    exit section

    reminder section
  - } while (1);

Initial attempts to Solve Problem

- Only 2 processes, P0 and P1
- General structure of process Pi (other process Pj)

do {
entry section
(Primitive) critical section
exit section
reminder section
} **while (1)**;

- Processes may share some common variables to synchronize their actions

Algorithm 1 (Dekker 1)

- Shared variables:
  int turn;
  initially **turn** = **0**
  turn = **i** → Pi can enter its critical section
- Process Pi

do {
while (turn != i) ;
critical section
turn = **j**;
reminder section
} **while (1)**;

- Satisfies mutual exclusion, but not progress

Algorithm 1

- Satisfies mutual exclusion
  - ✓ The turn is equal to either i or j and hence one of Pi and Pj can enter the critical section
- Does not satisfy progress
  - ✓ Example: Pi finishes the critical section and then gets stuck indefinitely in its remainder section. Then Pj enters the critical section, finishes, and then finishes its remainder section. Pj then tries to enter the critical section again, but it cannot since turn was set to i by Pj in the previous iteration. Since Pi is stuck in the remainder section, turn will be equal to i indefinitely and Pj can't enter although it wants to. Hence no process is in the critical section and hence no progress.
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 2 (Dekker 2)

- Shared variables
  boolean flag[2];
  initially **flag [0] = flag [1] = false.**
  flag [i] = true → Pi ready to enter its critical section
  Process Pi

```
do {
        flag[i] := true;
        while (flag[j]) ;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but no progress and no bounded waiting requirement

Algorithm 2

- Satisfies mutual exclusion
  - ✓ If Pi enters, then flag[i] = true, and hence Pj will not enter.
- Does not satisfy progress
  - ✓ Can block indefinitely…. Progress requirement not met
  - ✓ Example: There can be an interleaving of execution in which Pi and Pj both first set their flags to true and then both check the other process' flag. Therefore, both get stuck at the entry section
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorith 3

- Shared Variables
  - ✓ **var** flag: **array** (0..1) **of** boolean;
    initially flag[0] = flag[1] = false;
  - ✓ flag[i] = true ⇛ Pi ready to enter its critical section
- Process Pi

```
repeat
    while flag[j] do no-op;
    flag[i] := true;
    critical section
    flag[i]:= false;
    remainder section
until false
```

Algorithm 3

- ***Does not satisfy mutual exclusion***
  - ✓ Example: There can be an interleaving of execution in which both first check the other process' flag and see that it is false. Then they both enter the critical section.
  - ✓ We don't need to discuss/consider progress and bounded wait when mutual exclusion is not satisfied

Algorithm 4 (Peterson)

- Combined shared variables of algorithms 1 and 2.
- Process Pi

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) ;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

- Meets all three requirements; solves the critical section problem for two processes.

Algorithm 4

- Satisfies mutual exclusion
  - If one process enters the critical section, it means that either the other process was not ready to enter or it was this process' turn to enter. In either case, the other process will not enter the critical section
- Satisfies progress
  - If one process exits the critical section, it sets its ready flag to false and hence the other process can enter. Moreover, there is no interleaving in the entry section that can block both.
- Satisfies bounded wait
  - If a process is waiting in the entry section, it will be able to enter at some point since the other process will either set its ready flag to false or will set to turn to this process.

Bakery Algorithm

- Critical section for n processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
  - If processes Pi and Pj receive the same number,
    - if i <= j, then Pi is served first; else Pj is served first.
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation –
  - Lexicographic order(ticket#, process id#)
    - (a,b) < (c,d) if (a<c) or if ((a=c) and (b < d))
    - max(a0,….an-1) is a number, k, such that k >=ai for i = 0,…,n-1
- Shared Data

  var choosing: **array**[0..n-1] **of** Boolean ;(initialized to false)

  number: **array**[0..n-1] **of** integer ; (initialized to 0)

Bakery Algorithm (cont.)

```
repeat
   choosing[i] := true;
   number[i] := max(number[0], number[1],…,number[n-1]) +1;
   choosing[i] := false;
   for j := 0 to n-1
     do begin
          while choosing[j] do no-op;
          while number[j] <> 0
              and (number[j] ,j) < (number[i],i) do no-op;
     end;
      critical section
    number[i]:= 0;
      remainder section
until false;
```

Bakery Algorithm

Process	Number
P0	3
P1	0
P2	7
P3	4
P4	8

Bakery Algorithm

P0	P2	P3	P4
(3,0) < (3,0)	(3,0) < (7,2)	(3,0) < (4,3)	(3,0) < (8,4)
Number[1] = 0	Number[1] = 0	Number[1] = 0	Number[1] = 0
(7,2) < (3,0)	(7,2) < (7,2)	(7,2) < (4,3)	(7,2) < (8,4)
(4,3) < (3,0)	(4,3) < (7,2)	(4,3) < (4,3)	(4,3) < (8,4)
(8,4) < (3,0)	(8,4) < (7,2)	(8,4) < (4,3)	(8,4) < (8,4)

Bakery Algorithm

- P1 not interested to get into its critical section □ number[1] is 0
- P2, P3, and P4 wait for P0
- P0 gets into its CS, get out, and sets its number to 0
- P3 get into its CS and P2 and P4 wait for it to get out of its CS
- P2 gets into its CS and P4 waits for it to get out
- P4 gets into its CS
- Sequence of execution of processes: <P0, P3, P2, P4>

Bakery Algorithm

- Meets all three requirements:
  - ✓ Mutual Exclusion: (number[j], j) < (number[i], i) cannot be true for both Pi and Pj
  - ✓ Bounded-waiting: At most one entry by each process (n-1 processes) and then a requesting process enters its critical section (First-Come-First-Serve)
  - ✓ Progress: Decision takes complete execution of the ‘for loop’ by one process. No process in its ‘Remainder Section’ (with its number set to 0) participates in the decision making

Supporting Synchronization

Programs	***Shared Programs***
Higher-level API	***Locks Semaphores Monitors Send/Receive CCregions***
Hardware	***Load/Store Disable Ints Test&Set Comp&Swap***

- We are going to implement various synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide inherent support for synchronization at the hardware level
  - Need to provide primitives useful at software/user level

Synchronization Hardware

- Normally, access to a memory location excludes other accesses to that same location.
- Extension: designers have proposed machine instructions that perform two operations atomically (indivisibly) on the same memory location (e.g., reading and writing).
- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).
- They can be used to provide mutual exclusion but other mechanisms are needed to satisfy the other two requirements of a good solution to the CS problem.

Hardware Solutions for Synchronization

- Load/store - Atomic Operations required for synchronization
  - Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!
- Mutual exclusion solutions presented depend on memory hardware having read/write cycle.
  - If multiple reads/writes could occur to the same memory location at the same time, this would not work.
  - Processors with caches but no cache coherency cannot use the solutions
- In general, it is impossible to build mutual exclusion without a primitive that provides some form of mutual exclusion.
  - How can this be done in the hardware???
  - How can this be simplified in software???

Synchronization Hardware

- Test and modify the content of a word atomically - Test-and-set instruction
- **function** Test-and-Set (**var** target: boolean): boolean;
  begin
  Test-and-Set := target;
  target := true;
  end;
- Similarly “SWAP” instruction

Mutual Exclusion with Testand-Set

- Shared data: var lock: boolean (initially false)
- Process Pi

repeat
    while Test-and-Set (lock) **do** no-op;
    critical section
    lock := false;
    remainder section
until false;

Bounded Waiting Mutual Exclusion with Test-and-Set

- **var** j : 0..n-1;
  key : boolean;
  repeat
  waiting [i] := true; key := true;
  while waiting[i] **and** key **do** key := Test-and-Set(lock);
  waiting [i] **:=** false**;**
  critical section
  j := i+1 mod n;
  while (j <> i) **and** (**not** waiting[j]) **do** j := j + 1 **mod** n;
  if j = i **then** lock := false;
  else waiting[j] := false;
  remainder section
  until false;

Solution with TSL

- Is the TSL-based solution good? No
- Mutual Exclusion: Satisfied
- Progress: Satisfied
- Bounded Waiting: Not satisfied

Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b)
{
boolean temp = a;
a = b;
b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):
  boolean lock;
  boolean waiting[n];
- 'key' is local and set to false
- Process Pi

```
do {
        key = true;
        while (key == true)
        Swap(lock,key);
        critical section
        lock = false;
        remainder section
        }
```


Solution with swap

- Is the swap-based solution good? No
- Mutual Exclusion: Satisfied
- Progress: Satisfied
- Bounded Waiting: Not satisfied

A Good Solution

- ‘key’ local; ‘lock’ and ‘waiting’ global
- All variables set to false

```
do {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
            key = TestAndSet(lock);
        waiting[i] = false;
            Critical Section
```

A Good Solution

```
        j = (i+1) % n;
        while ( (j != i) && !waiting[j] )
                j = (j+1) % n;
        if (j == i)
                lock = false;
        else
                waiting[j] = false;
            Remainder Section
}
```

Đinh Công Đoan 60

Solution with Test-And-Set

- Is the given solution good? Yes
- Mutual Exclusion: Satisfied
- Progress: Satisfied
- Bounded Waiting: Satisfied

Semaphore

- Semaphore S - integer variable (non-negative)
  - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations

```
wait (S):     while S <= 0 do no-op
              S := S-1;
signal (S):   S := S+1;
```

  - P or wait used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
  - V or signal releases a resource and increments the semaphore by 1, waking up a waiting P, if any
  - If P is performed on a count <= 0, process must wait for V or the release of a resource.
- **P():"proberen" (to test) ; V() "verhogen" (to increment) in Dutch**

Example: Critical Section for n Processes

- Shared variables

  var mutex: semaphore
  initially mutex = 1
- Process Pi

  repeat
  wait(mutex);
  critical section
  signal (mutex);
  remainder section
  until false

Semaphore as a General Synchronization Tool

- Execute B in Pj only after A execute in Pi
- Use semaphore flag initialized to 0
- Code

P_i	P_j
⋮	⋮
A	*wait(flag)*
signal(flag)	*B*

Problem...

- Locks prevent conflicting actions on shared data
  - ✓ Lock before entering critical section and before accessing shared data
  - ✓ Unlock when leaving, after accessing shared data
  - ✓ Wait if locked
- All Synchronization involves waiting
  - ✓ **Busy Waiting,** uses CPU that others could use. This type of semaphore is called a spinlock.
    - Waiting thread may take cycles away from thread holding lock (no one wins!)
    - OK for short times since it prevents a context switch.
    - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
  - ✓ Should sleep if waiting for a long time
- For longer runtimes, need to modify P and V so that processes can block and resume

Semaphore Implementation

- Define a semaphore as a record

  type semaphore = **record**

  value: integer;

  L: **list of** processes;

  end;
- Assume two simple operations
  - ✓ block suspends the process that invokes it.
  - ✓ wakeup(P) resumes the execution of a blocked process P

Semaphore Implementation(cont.)

- Semaphore operations are now defined as

```
wait (S): S.value := S.value -1;
    if S.value < 0
    then begin
            add this process to S.L;
            block;
    end;
signal (S): S.value := S.value +1;
    if S.value <= 0
    then begin
    remove a process P from S.L;
    wakeup(P);
    end;
```


Block/Resume Semaphore Implementation

- If process is blocked, enqueue PCB of process and call scheduler to run a different process.
- Semaphores are executed atomically;
  - no two processes execute wait and signal at the same time.
  - Mutex can be used to make sure that two processes do not change count at the same time.
    - If an interrupt occurs while mutex is held, it will result in a long delay.
    - Solution: Turn off interrupts during critical section.

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 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.

Two Types of Semaphores

- Counting Semaphore - integer value can range over an unrestricted domain.
- Binary Semaphore - integer value can range only between 0 and 1; simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

Classical Problems of Synchronization

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

Bounded Buffer Problem

- Shared data

  type item = ….;
  var buffer: **array**[0..n-1] **of** item;
  full, empty, mutex : semaphore;
  nextp, nextc :item;
  full := 0; empty := n; mutex := 1;

Bounded Buffer Problem

- Producer process - creates filled buffers
  repeat
  …
  produce an item in nextp
  …
  wait (empty);
  wait (mutex);
  …
  add nextp to buffer
  …
  signal (mutex);
  signal (full);
  until false;

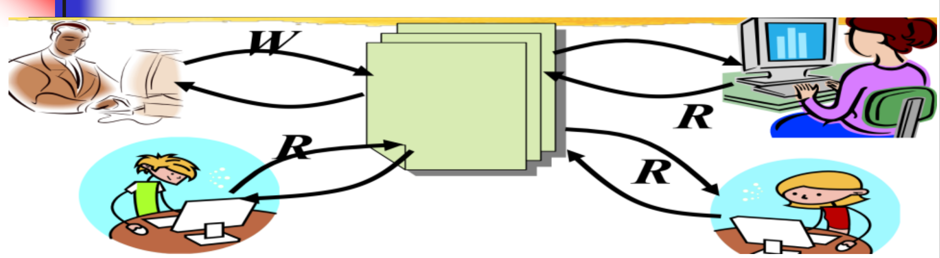
Bounded Buffer Problem

- Consumer process - Empties filled buffers
  repeat
  wait (full);
  wait (mutex);
  …
  remove an item from buffer to nextc
  ...
  signal (mutex);
  signal (empty);
  …
  consume the next item in nextc
  …
  until false;

Discussion

- ASymmetry?
  - Producer does: P(empty), V(full)
  - Consumer does: P(full), V(empty)
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

Readers-Writers Problem

- Shared Data

  var mutex, wrt: semaphore (=1);

  readcount: integer (= 0);

- Writer Process

  wait(wrt);

  …

  writing is performed

  ...

  signal(wrt);

Readers-Writers Problem

- Reader process

  wait(mutex);

  readcount := readcount +1;

  if readcount = 1 **then** wait(wrt);

  signal(mutex);

  ...

  reading is performed

  ...

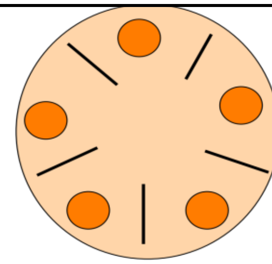
  wait(mutex);

  readcount := readcount - 1;

  if readcount = 0 **then** signal(wrt);

  signal(mutex);

Dining-Philosophers Problem



- Shared Data

var chopstick: **array** [0..4] **of** semaphore (=1

- Philosopher i :
  repeat
  wait (chopstick[i]);
  wait (chopstick[i+1 **mod** 5]);
  …
  eat
  ...
  signal (chopstick[i]);
  signal (chopstick[i+1 **mod** 5]);
  …
  think
  …
  until false;

Higher Level Synchronization

- Timing errors are still possible with semaphores
- Example 1
  - signal (mutex);
    …
    critical region
    ...
    wait (mutex);
- Example 2
  - wait(mutex);
    …
    critical region
    ...
    wait (mutex);
- Example 3
  - wait(mutex);
    …
    critical region
    ...
    Forgot to signal

Motivation for Other Sync. Constructs

- Semaphores are a huge step up from loads and stores
  - Problem is that semaphores are dual purpose:
    - They are used for both mutex and scheduling constraints
    - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Idea: allow manipulation of a shared variable only when condition (if any) is met – **conditional critical region**
- Idea : Use locks for mutual exclusion and condition variables for scheduling constraints
- **Monitor:** a lock (for mutual exclusion) and zero or more condition variables (for scheduling constraints) to manage concurrent access to shared data
  - Some languages like Java provide this natively

Conditional Critical Regions

- High-level synchronization construct
- A shared variable v of type T is declared as:
  var v: **shared** T
- Variable v is accessed only inside statement
  region v when B **do** S
  - where B is a boolean expression.
  - While statement S is being executed, no other process can access variable v

Critical Regions (cont.)

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement,

  region v when B **do** S

  - ✓ the Boolean expression B is evaluated.
  - ✓ If B is true, statement S is executed.
  - ✓ If it is false, the process is delayed until B becomes true and no other process is in the region associated with v

Example - Bounded Buffer

- Shared variables

  var buffer: **shared record**
  pool:**array**[0..n-1] **of** item;
  count,in,out: integer;
  end;

- Producer Process inserts nextp into the shared buffer

  region buffer **when** count < n
  do begin
  pool[in] := nextp;
  in := in+1 **mod** n;
  count := count + 1;
  end;

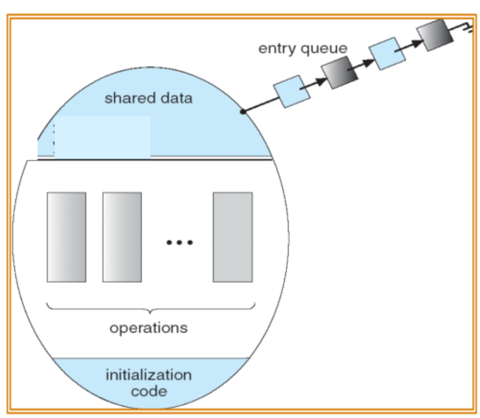
Bounded Buffer Example

- Consumer Process removes an item from the shared buffer and puts it in nextc
- **region** buffer **when** count > 0

```
    do begin
        nextc := pool[out];
        out := out+1 mod n;
        count := count -1;
    end;
```

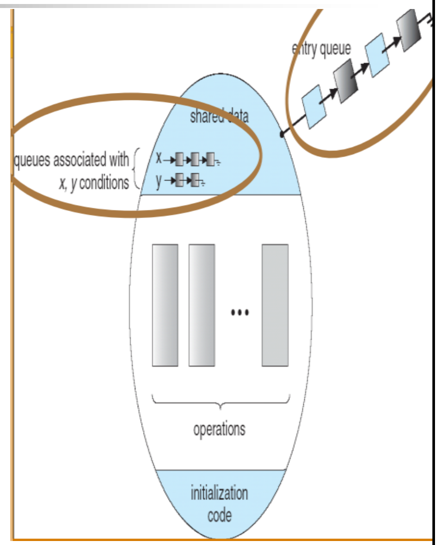

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes
- **type** monitor-name = **monitor**
  variable declarations
  procedure entry P1 (…);
  begin … **end**;
  procedure entry P2 (…);
  begin … **end**;
  .
  .
  .
  procedure entry Pn(…);
  begin … **end**;
  begin
  initialization code
  end



Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep



Monitors with condition variables

- To allow a process to wait within the monitor, a condition variable must be declared, as:
  var x,y: condition
  - Condition variable can only be used within the operations wait and signal. Queue is associated with condition variable.
  - The operation
    x.wait;
    means that the process invoking this operation is suspended until another process invokes
    x.signal;
  - The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect

Dining Philosophers

- **type** dining-philosophers= **monitor**
  var state: **array**[0..4] **of** (thinking, hungry, eating);
  var self: **array**[0..4] **of** condition;
  // condition where philosopher I can delay himself when hungry
  but // is unable to obtain chopstick(s)
  procedure entry pickup (i :0..4);
  begin
  state[i] := hungry;
  test(i); //test that your left and right neighbors are not eating
  if state [i] <> eating **then** self [i].wait;
  end;
  procedure entry putdown (i:0..4);
  begin
  state[i] := thinking;
  test (i + 4 **mod** 5); // signal one neighbor
  test (i + 1 **mod** 5); // signal other neighbor
  end;

Dining Philosophers (cont.)

```
procedure test (k :0..4);
begin
      if state [k + 4 mod 5] <> eating
      and state [k ] = hungry
      and state [k + 1 mod 5] <> eating
then
begin
      state[k] := eating;
      self [k].signal;
end;
end;
begin
      for i := 0 to 4
            do state[i] := thinking;
end;
```