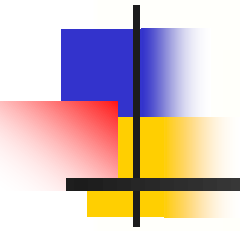


SWENG3043

Operating Systems

Software Engineering Department

AASTU

Nov 2018

Chapter Three

Process Synchronization

Objectives

- After completing this chapter you should be able to:
  - Understand the concept of process synchronization
  - Understand critical-section problem, whose solutions can be used to ensure the consistency of shared data
  - Explain both software and hardware solutions of the critical-section problem
  - Examine several classical process-synchronization problems

Topics Included

- Race Conditions
- The Critical-Section Problem
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Classic Problems of Synchronization

Concurrency

- Processes can execute concurrently or in parallel
- The fundamental design issue in the management of multiple processes is *concurrency*: simultaneous execution of multiple processes.
- Concurrency arises in three different contexts
  - *Multiple applications*: concurrently running applications
  - *Structured applications*: an application structured as a set of concurrent processes (threads)
  - *OS structure*: OS implemented as a set of processes or threads.

Concurrency (cont'd)

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- E.g. Consider two process, P3 and P4, that share two variables b=1 and c=2
  - P3 executes b= b+ c and P4 executes c= b + c
    - If P3 executes first, value of b=3 and c=5.
    - If P4 executes first, value of b=4 and c=3

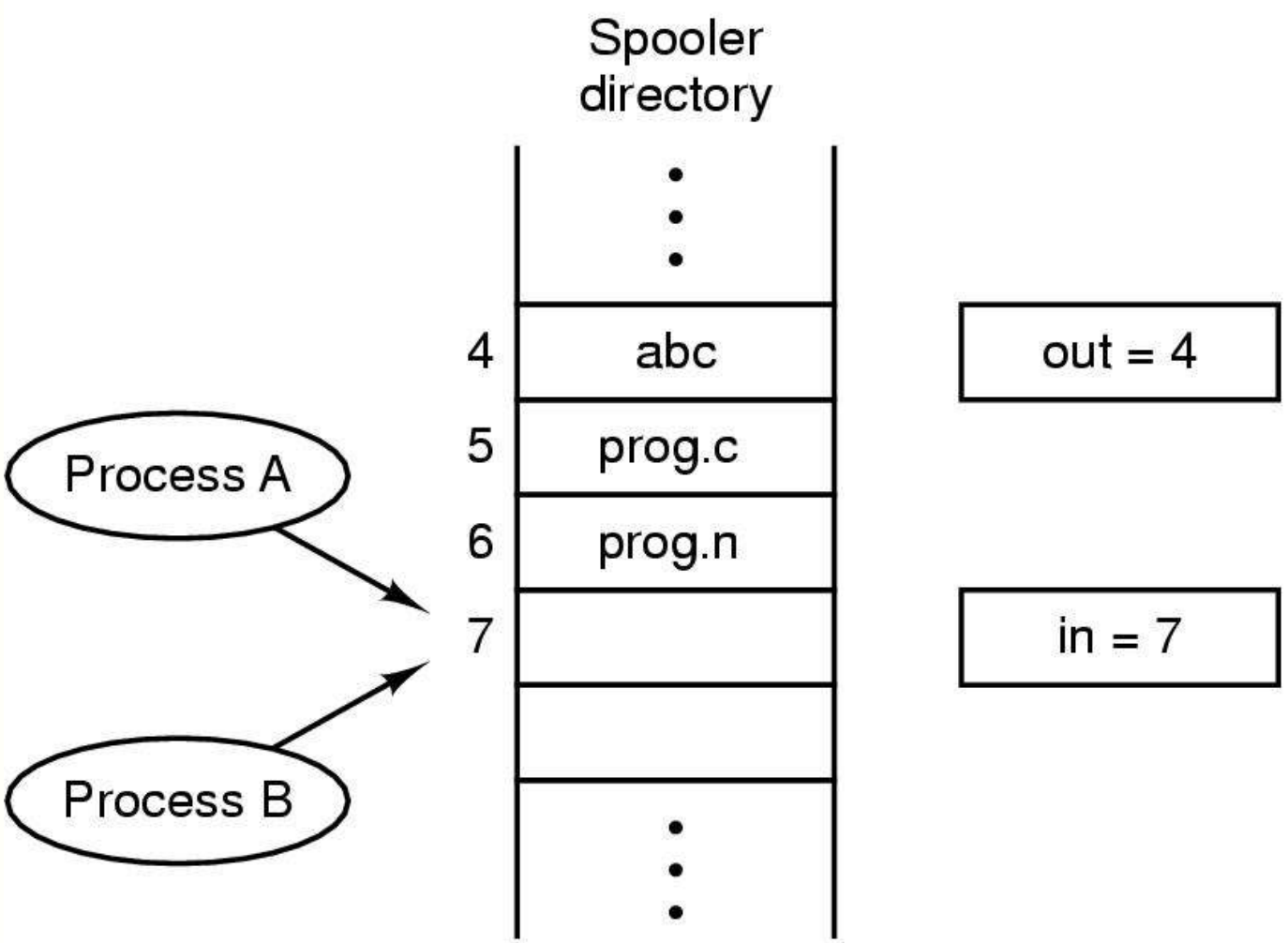
Race Condition

- A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.
  - E.g. *Printer Spooler*: When a process wants to print a file, it enters the file name in a special spooler directory. Assume that the spooler directory has a large number of slots, numbered 0,1,2,… *Printer Daemon* checks if there are any files to be printed.
  - There are two globally shared variables
    - *Outfile*: points to the next file to be printed
    - *Infile*: points to the next free slot in the directory
  - There were some files in the spooler directory and assume the current value of infile is 7 and that of outfile is 4

Race Condition(cont'd)

- Assume that simultaneously process A and process B decide they want to queue a file for printing
  - Process A reads infile and stores the value 7 in its local variable (x=infile)
  - An interrupt occurs and the CPU decides that process A has run long enough so it switches to process B. Process B reads infile and stores the value 7 in its local variable (y=infile)
  - Process B stores the name of its file in slot 7 and adjusts infile to be 8
  - Eventually process A runs again, it checks x and finds 7 there, and writes its file name in slot 7, erasing the name process B just put there, it updates infile to be 8
  - The printer will now print the file of process A, process B file will never get any output

Race Condition(cont'd)

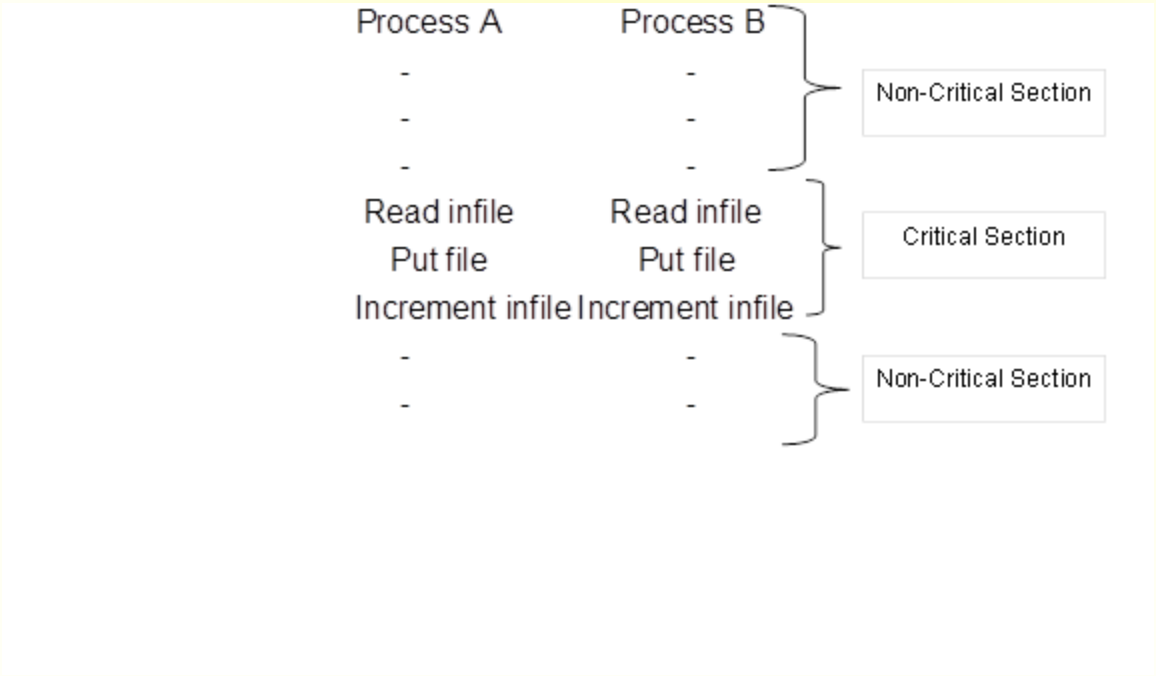


Two processes want to access shared memory at same time

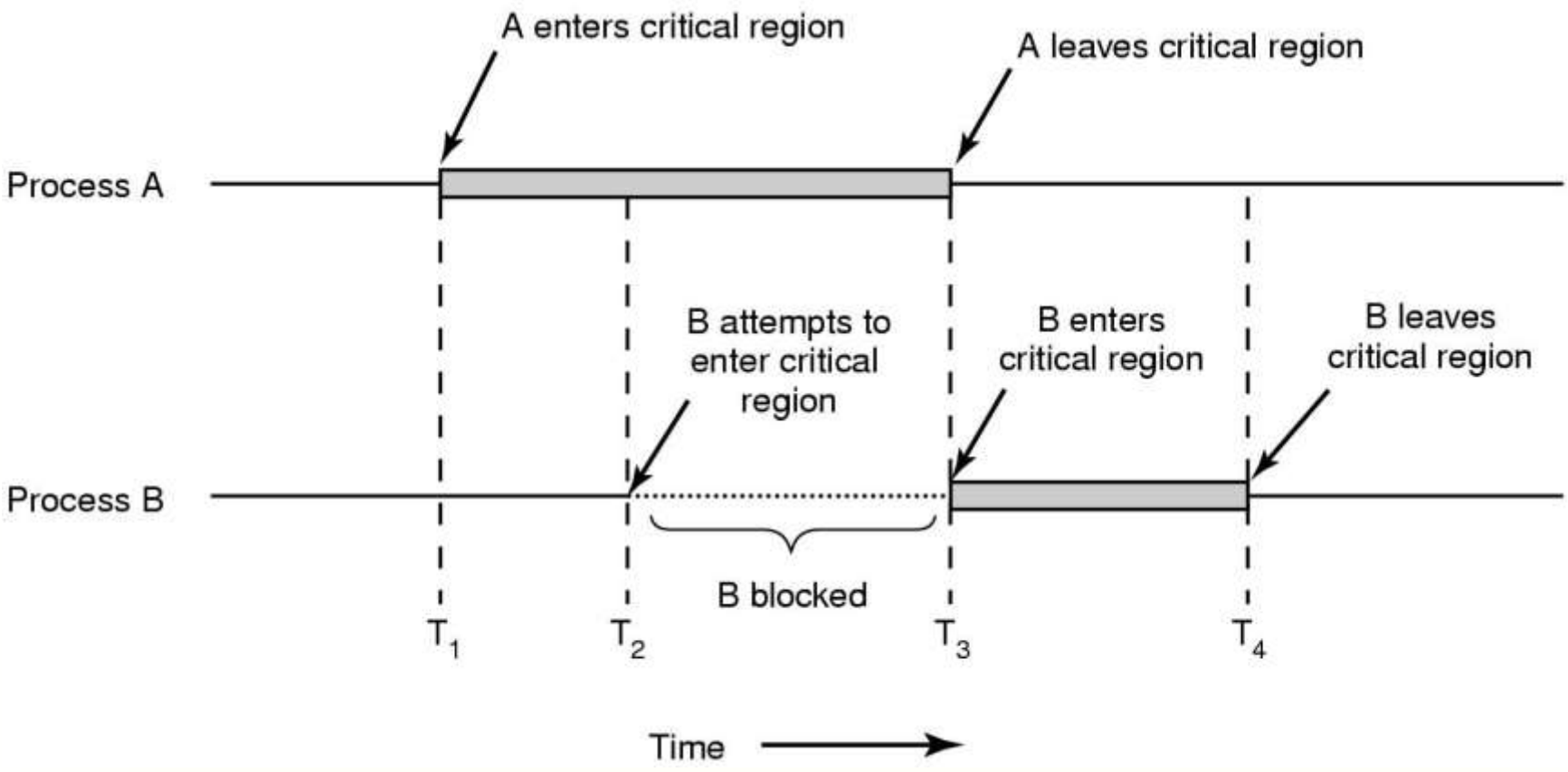
Critical Region

- The key to prevent race condition is to enforce mutual exclusion - it is the ability to exclude (prohibit) all other processes from using a shared variable or file while one process is using it.
- Part of a program where shared resource is accessed is called ***critical region*** or ***critical section.***

Critical Region(cont'd)



Critical Region(cont'd)



Mutual exclusion using critical regions

Critical Region(cont'd)

- Four conditions to provide mutual exclusion:
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region.
- Mutual Execution with Busy Waiting
  - while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.
  - Types:
    - Disabling Interrupts
    - Lock Variables
    - Strict Alternation
    - Peterson's Solution
    - TSL Instruction

Disabling Interrupts

- The simplest solution
  - In this technique, each process disables all interrupts just after entering its critical section and re-enables them just before leaving it.
- With interrupts turned off the processor will not be switched to another process
- Thus, once a process is in critical region, any other process will not intervene.

Disabling Interrupts(cont'd)

- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.
  - For instance, a process can turn off interrupts but never turn them on again in which the whole system freezes.
  - If the system is multiprocessor, disabling interrupts affects only the processor that executed the disable instruction. The other ones will continue running and can access the shared memory.
- Disabling interrupts is often a useful technique within the operating system itself.
- But, is not appropriate as a general mutual exclusion mechanism for user processes.

Lock Variables

- It is a software solution.
- Assume we have a single shared (lock) variable initially set to 0
- A process enters its critical section if this variable is 0, when it enters it sets it to 1. If the lock is already 1, the process waits until it becomes 0
- Thus, 0 means that no process is in its critical region, and a 1 means that some process is in its critical region
- Disadvantage:
  - Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs and sets the lock to 1.
  - When the first process runs again, it will also set the lock to 1 and the two processes will be in their critical region at the same time causing race condition.

Strict Alternation

- This solution requires that the two processes strictly alternate in entering their critical regions.
- The integer turn, initially 0, keeps track of whose turn it is to enter the critical region and examines or updates the shared memory.
- Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.

Strict Alternation(cont'd)

```
While (TRUE) {
   While(turn!=0) /*wait*/;
   Critical_region();
   turn=1;
   Noncritical_region();
}
```

(a) Process 0

A proposed solution to the critical region problem

Strict Alternation(cont'd)

```
While (TRUE) {
   While(turn!=1) /*wait*/;
   Critical_region();
   turn=0;
   Noncritical_region();
}
```

(b) Process 1

Strict Alternation(cont'd)

- Taking turns is not a good idea when one of the processes is much slower than the other. This situation violates condition 3 of implementing mutual exclusions: process 0 is being blocked by a process not in its critical region.
- Continuously testing a variable waiting for some value to appear is called **busy waiting**. This technique wastes processor time

Peterson's Solution

- It combines the idea of taking turns with the idea of lock variables.
- Before entering its critical region each process calls enter_region with its own process number, 0 or 1, as parameter
  - Will cause to wait, if need to be, until it safe to enter
  - When done, calls leave_region to allow other process to enter
- Initially, neither process is in its critical section.
  - Now process 0 calls enter_region. It indicates its interest by setting its array element & sets turn to 0.
  - Since process 1 is not interested, enter_region returns immediately.
  - If process 1 now calls enter_region, it will hang there until interested[0] goes to false, an event that only happens when process 0 calls leave_region to exit the critical region.

Peterson's Solution(cont'd)

- Now consider the case that both processes call enter_region almost simultaneously
  - Both will store their process number in turn.
  - Whichever store is done last is the one that counts; the first one is lost.
  - Suppose that process 1 stores last, so turn is 1.
  - When both processes come to the while statement, process 0 executes it zero times and enters its critical section. Process 1 loops and does not enter its critical section.

Peterson's Solution(cont'd)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

TSL Instruction

- This technique requires a little help from the hardware.
- Many multiprocessor computers have an instruction:

  TSL RX, LOCK
  - TSL (Test and Set Lock) is an indivisible atomic instruction that copies the content of a memory location into a register and stores a non-zero value at the memory location.
    - Indivisible mean no other processor can access the memory word until the instruction is finished.
- To implement mutual exclusion with TSL instruction, we use a shared variable, lock, to coordinate access to shared memory. When lock is 0 the process may set it to 1 using TSL instruction and executes its critical section. When it is done, the process sets lock back to 0.

TSL Instruction(cont'd)

- Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in lock.

TSL Instruction(cont'd)

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

Sleep and Wakeup

- Both Peterson's solution and the solution using TSL are correct.
  - But both have the defect of requiring *busy waiting* which wastes CPU time and which can also have unexpected effects, like the *priority inversion problem*.
- Priority inversion problem:
  - consider a computer with two processes, H with high priority and L with low priority.
  - The scheduling rule: H runs whenever it is in the ready state.
  - At a certain moment, with L in its critical region, H becomes ready to run(e.g. I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops.

Sleep and Wakeup(cont'd)

- Now let us look at some inter-process communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions.
  - **Sleep**: It is a system call that causes the caller to block, i.e. be suspended until another process wakes it up.
  - **Wakeup**: It is a system call that causes the process specified by the parameter to wake up.
- As an example of how these primitives can be used let us consider the producer-consumer problem (also known as the bounded buffer problem)

Producer-Consumer Problem

- Two processes share a common fixed-size buffer. One of, the producers, puts information in the buffer, and the other one, the consumer, takes it out.
- When the producer wants to put a new item in the buffer, it checks the buffer, if it is full, it goes to sleep, to be awakened when the consumer has removed one or more items if it is not, a producer will add an item and increment *count*.
- When the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. If it is nonzero, remove an item and decrement *count*.
- Let us see the producer-consumer problem using C programming.

Producer-Consumer Problem(cont'd)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Producer-consumer problem with fatal race condition

Producer-Consumer Problem(cont'd)

- Problem: Race condition can occur because access to count is unconstrained.
  - Consider the following situation:
    - The buffer is empty and the consumer has just read count to see if it is 0.
    - At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
    - The producer enters an item in the buffer, increments count, and notices that it is now 1.
    - Reasoning the count was just 0, and thus the consumer must be sleeping, and the producer calls wakeup to wake the consumer up.
    - Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.

Producer-Consumer Problem(cont'd)

- When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
- Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

- The problem arises because the wakeup signal is lost.
- A quick fix is to add to the rules by adding **wakeup-waiting bit**.
  - When a wakeup is sent to a process that is still awake, this bit is set.
  - Later, when the process tries to go to sleep, if the wakeup-waiting bit is on, it will be turned off, but the process will stay awake.
  - The wakeup waiting bit **cannot be a general solution**, especially for any random number of processes.

Semaphores

- Semaphores solve the lost-wakeup problem
- A semaphore is a new integer variable type that counts the number of wakeups saved for future use.
- A semaphore could have the value 0, indicating that no wakeups were saved or some positive value if one or more wakeups were pending.
- Two operations were proposed to implement semaphores: up and down (which are generalizations of wakeup and sleep, respectively).
  - DOWN operation
    - It checks the value of the semaphore to see if the value is greater than 0. If so it decrements the value and just continues. If the value is 0, the process is put to sleep without completing the DOWN operation for the moment.

Semaphores(cont'd)

- Checking the value, changing it and going to sleep is all done as a single, indivisible atomic operation. Once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.
- This atomicity is **essential to solving synchronization problems and avoiding race conditions.**

- UP operation
  - It increments the value of the semaphore. If one or more processes were sleeping on that semaphore, unable to complete an earlier DOWN operation, one of them is chosen by the system and is allowed to complete its DOWN operation
  - The process of incrementing the semaphore and waking up one process is also indivisible.

- The solution uses three semaphores:
  - *full*, initially 0, to count the number of full slots,
  - *empty*, initially equal to number of slots in buffer, to count the number of empty slots and
  - *mutex*, initially 1, to make sure the producer and the consumer do not access the buffer at the same time

Semaphores(cont'd)

- Mutex is used for mutual exclusion
  - It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables.
- Full/empty are used for synchronization
  - ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty.
- The producer stops running when the buffer is full, and the consumer stops running when it is empty.
- Semaphores that are initialized to 1 and are used by two or more processes to ensure that only one of them can enter its critical region at the same time are called *binary semaphores*

Semaphores(cont'd)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

The producer-consumer problem using semaphores

Semaphores(cont'd)

- Problem:
  - Semaphores are too low-level and error prone. If you are not careful when using them, errors like race condition, deadlocks and other forms of unpredictable and irreproducible behavior can occur.
  - Suppose that the two downs in the producer's code were interchanged or reversed in order and suppose also the buffer was full and mutex is 1

    down (&mutex);

    down (&empty);

    - The producer does a down on mutex and mutex becomes 0 and then the producer does a down on empty.
      - The producer would block since the buffer is full.
      - Next time the consumer does a down on mutex and it blocks since mutex is 0.
      - Therefore both processes would block forever and hence deadlock would occur.

Mutex

- Simplified version of semaphore: used when count is not needed
- Are good only for managing mutual exclusion to some shared resource or piece of code.
- Useful in thread packages that are implemented entirely in user space
- Mutex is a variable that can be in one of two states: unlocked(critical region is available) or locked.
  - Integer value 0 represent unlocked and all other values represent locked
- Thread(process) calls *mutex_lock to* access critical region and calls *mutex_unlock* when exit the critical region.
  - If the mutex is unlocked, the calling thread can enter to the critical region otherwise it will be blocked
  - If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Mutex(cont'd)

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

Monitors

- A monitor is a higher level of synchronization primitive proposed by Hoare and Branch Hansen to make writing a correct program easier.
- A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package.
- **Rules associated with monitors**
  - Processes may call the procedures in a monitor whenever they want to, but they can not directly access the monitor's internal data structures
  - Only one procedure can be active in a monitor at any instant. Monitors are programming language construct, so the compiler knows that they are special and can handle calls to a monitor procedures differently from other procedure calls

Monitors(cont'd)

- Compiler implements mutual exclusion. The person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all critical regions into monitor procedure, no two processes will ever execute their critical regions at the same time.

- Monitors use condition variables, along with two operations on them, WAIT and SIGNAL to block and wake up a process.
  - ***WAIT***
    - When a monitor procedure discovers that it can not continue(e.g., the producer finds the buffer full), it does a WAIT on some condition variable that causes the calling procedure to block.
    - It also allows another process that had been previously prohibited from entering the monitor to enter now.

Monitors(cont'd)

- ***SIGNAL***
  - A process can wake up its sleeping partner by doing a SIGNAL on the condition variable that its partner is waiting on
  - A process doing a SIGNAL statement must exit the monitor immediately, i.e. SIGNAL will be the final statement in a monitor procedure. This avoids having two active processes in a monitor at the same time.
  - If a SIGNAL is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler is revived.

Monitors(cont'd)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitors(cont'd)

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

Monitors(cont'd)

- Limitations:
  - Monitors are a programming language concepts.
    - Most languages do not have monitors(E.g. C)
  - They were designed for CPUs that will have access to a common memory – can not be implemented in distributed systems.

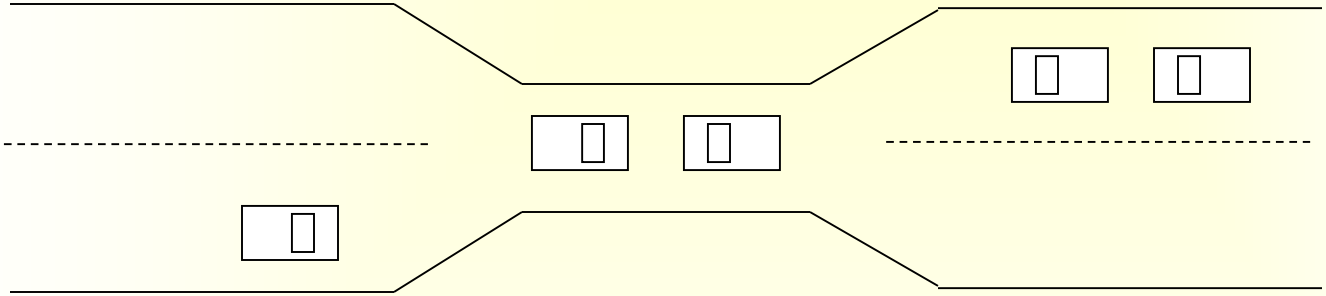
Concurrency Problems

- The enforcement of mutual exclusion creates additional control problems:
  - **Deadlock**
  - **Starvation**
- **Deadlock**
  - It is the permanent blocking of a set of processes compete for system resources or communicate with each other.
  - It refers to a situation in which a set of two or more processes are waiting for other members of the set to complete an operation in order to proceed, but none of the members is able to proceed.
  - It is a difficult phenomenon to anticipate and there are no easy general solutions to this problem

Concurrency Problems(cont'd)

- Example 1:
  - Consider two processes, P1 and P2, and two resources, R1 and R2.
  - Suppose that each process needs access to both resources to perform part of its function.
  - Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1.
  - Each process is waiting for one of the two resources.
  - Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources.
  - The two processes are deadlocked.

Concurrency Problems(cont'd)



Deadlock: Bridge Crossing Example

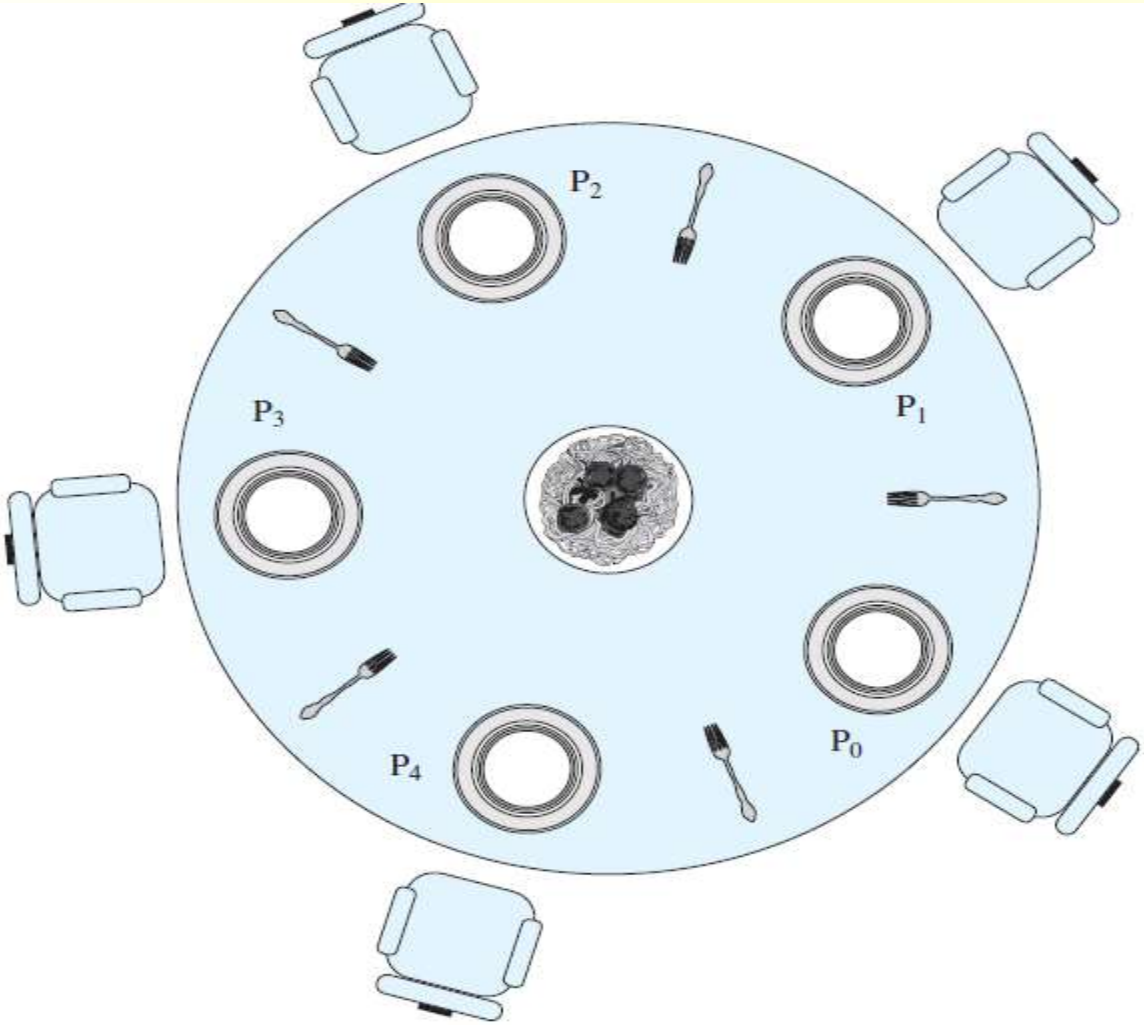
Concurrency Problems(cont'd)

- **Starvation**
  - It referees to the situation in which a process is ready to execute but is continuously denied access to a processor in deference to other processes.
  - E.g. suppose that there are three processes P1, P2, and P3 and each require periodic access to resource R. If the operating system grants the resource to P1 and P2 alternately, P3 may indefinitely be denied access to the resource, thus starvation may occur.
  - In large part, it is a scheduling issue

Classical IPC Problems

- Three better-known problems:
  - Dinning-philosopher problem
  - The Readers and Writers problem
  - The sleeping barber problem
- **Dinning-philosopher problem**
  - Five philosophers are seated around a circular table.
  - Each philosopher has a plate of spaghetti.
  - Between each pair of plates is one fork.
  - The life of a philosopher consists of alternate periods of eating and thinking.
  - The procedure: take_fork waits until the specified fork is available and then seizes it.
  - Problem:
    - Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and their will be deadlock.

Dinning-philosopher problem



Dinning-philosopher problem(cont'd)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

A **nonsolution** to the dinning philosophers problem

Dinning-philosopher problem(cont'd)

- Problem:
  - Suppose that all five philosophers take their left fork simultaneously. None will be able to take their right forks, and their will be deadlock.
- Solution to dining philosophers problem
  - After taking the left fork, the program checks to see if the right fork is available.
  - If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process.
  - Still some problem. If all the philosophers could start the algorithm simultaneously
    - Picking up their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever.
    - This situation is called starvation.
  - Just wait a random time instead of the same time. But, this does not always works.

Dinning-philosopher problem(cont'd)

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1)

Dinning-philosopher problem(cont'd)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
     test(i);                           /* try to acquire 2 forks */
     up(&mutex);                        /* exit critical region */
     down(&s[i]);                       /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = THINKING;               /* philosopher has finished eating */
     test(LEFT);                        /* see if left neighbor can now eat */
     test(RIGHT);                       /* see if right neighbor can now eat */
     up(&mutex);                        /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Solution to dining philosophers problem (part 2)

The Readers and Writers Problems

- 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 treated
- Shared Data
  - Semaphore mutex initialized to 1
  - Semaphore db initialized to 1
  - Integer readcount initialized to 0
- The dining philosopher problem is useful for modeling process that are competing for exclusive access to a limited number of resources, such as I/O devices.
- The readers and writers problem, models access to database.

The Readers and Writers Problems(cont'd)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc - 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read( );             /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repeat forever */
          think_up_data( );             /* noncritical region */
          down(&db);                    /* get exclusive access */
          write_data_base( );           /* update the data */
          up(&db);                      /* release exclusive access */
     }
}
```

A solution to the readers and writers problem

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

The Sleeping Barber Problem

- The barber shop has one barber, one barber chair, and n chairs waiting for customers.
- If there are no customers present, the barber sits down in the barber chair and falls asleep.
- When a customer arrives, he has to wakeup the sleeping barber.
- If additional customers arrive, they either sit(if there are empty chairs) or leave the shop(if all chairs are full)
- The problem is to program the barber and the customers without getting into race conditions.

The Sleeping Barber Problem(cont'd)

The Sleeping Barber Problem

The Sleeping Barber Problem(cont'd)

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting - 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Solution to sleeping barber problem.