Synchronization in Cooperating Processes - I

Concurrent and Parallel Execution

- Threads of execution of different processes can run on different cores of a modern processor concurrently or in parallel.
- A running thread may be interrupted at any point. And another thread may be scheduled in its place on the processing core.

Cooperating Process

- Two processes[a] are said to be cooperating if one can affect or gets affected by the computation of the other.
- Cooperating processes share data either by sharing the address space, or by sharing file or by exchange of messages[b].

---

[a] This is true for different threads within a process.

[b] Threads within a process share the global data space.

Cooperating Process

- Concurrent and/or parallel execution of cooperating processes or threads may lead to concurrent access to shared data.
- If such an access is unrestricted, it may lead to data inconsistency.
- Different methods have been designed to maintain data integrity by putting restriction on concurrent access of data.

Producer-Consumer

- Following example is the classic producer-consumer problem of cooperating processes.
- Producer and consumer processes share data through a bounded buffer.
- The producer generates data that is used by the consumer.

Producer-Consumer

- The bounded buffer is organized as a finite size circular queue.
- The queue resides in the shared memory.
- The producer can add data as long as the queue is not full.
- The consumer can consume so long as the queue is not empty.

A Queue: `queue.h`

```
/*
  header file for queue.h
*/
#ifndef _QUEUE_H
#define _QUEUE_H

#define MAX 5
#define ERROR 1
#define OK 0
```

```
class queue {
  private:
      int data[MAX];
      int front, rear, count ;
  public:
      queue();
      int addQ(int);
      int deleteQ(void);
      int frontQ(int &);
      int isEmptyQ();
```

```
        int isFullQ();
} ;
#endif
```

A Queue: `queue.c++`

```
/*
 * queue.c++ implementation of int
 * queue
 * $ g++ -Wall -c queue.c++
*/
#include "queue.h"

queue::queue() {
    front = rear = 0; count = 0;
```

```
}

int queue::isFullQ(){
    return count == MAX;
}

int queue::isEmptyQ(){
    return count == 0;
}

int queue::addQ(int n){
```

```
    if(isFullQ()) return ERROR;
    rear = (rear + 1) % MAX;
    data[rear] = n;
    count = count+1;
    return OK;
}


int queue::frontQ(int &v){
    if(isEmptyQ()) return ERROR;
    v = data[(front+1)%MAX] ;
    return OK;
```

```
}

int queue::deleteQ(){
    if(isEmptyQ()) return ERROR;
    front = (front+1)%MAX ;
    count = count - 1;
    return OK;
}
```

Producer-Consumer: `prodCon1.c++`

```
/*
 * prodCon1.c++ Producer-Consumer Problem on
 *           shared memory
 * $ g++ -Wall prodCon1.c++ queue.o
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include "queue.h"

void producer(queue *);
void consumer(queue *);
int *countP, *countC; // global counters
```

```
int main() {
    int shmID, chID1, chID2, status ;
    struct shmid_ds buff ;
    queue *qP ;

    shmID = shmget(IPC_PRIVATE,
                   sizeof(queue),
                   IPC_CREAT | 0777);
    if(shmID == -1) {
       perror("Error in shmget\n") ;
       exit(1) ;
```

```
}
qP = (queue *) shmat(shmID, 0, 0777);
countP = (int *)(qP+1);
countC = countP+1;
*countP = *countC = 0; // counter init
if((chID1 = fork()) != 0) { // Parent
   if((chID2 = fork()) != 0) { // Parent now
       waitpid(chID1, &status, 0);
       waitpid(chID2, &status, 0);
       cout << *countP << " data produced\n";
       cout << *countC << " data consumed\n";
```

```
       shmdt(qP) ;
       shmctl(shmID, IPC_RMID, &buff);
   }
   else producer(qP);
     // consumer(qP);
     // Child 2: producer
   }
else  consumer(qP);
    // producer(qP);
    // Child I: consumer
return 0 ;
```

```
}
void producer(queue *qP){
     int added = 1, i;
     for(i=1;i<=500000;++i) {
        int data, err;

        if(added) {
           data = rand() ;
           added = 0 ;
        }
        err = qP->addQ(data) ;
```

```
            if(err == OK) {
               added = 1 ;
               cout << "Produced Data "
                    << ++(*countP)
                    << " " << data << "\n" ;
            }
         }
}
void consumer(queue *qP) {
         int i;
         for(i=1; i<= 500000; ++i) {
```

```
   int data, dataOK;

   dataOK = qP -> frontQ(data);
   qP -> deleteQ(); // where to put this cod
   if(dataOK == OK){
      // qP -> deleteQ() ;
      cout << "\tConsumed Data "
           << ++(*countC)
           << " " << data << "\n" ;
   }
}
```

```
}
```

Producer-Consumer: output_1

```
Produced Data 379 1858721860
Produced Data 380 1548348142
Produced Data 381 105575579
Produced Data 382 964445884
Produced Data 383 2118421993
383 data produced
309 data consumed
```

Funny output, Note that the queue size is only 5. Is it due to race?

Race Condition

- The consumer deletes some data without reporting.
- Two concurrent processes producer and consumer update the data structure queue.
- The outcome depends on the order of access to the shared data by their components e.g. `addQ()`, `deleteQ()` etc.

Race Condition

- Even if the queue is empty while executing `qP -> frontQ(data)`, it may not be empty while executing the next method `qP -> deleteQ()`.
- The method deletes the item without reporting.
- This is known as race condition.

Race Condition

- This can be avoided by shifting `qP -> deleteQ()` within the guard of `if(dataOK == OK)`.
- But a similar race condition may arise at a finer granularity as the high-level language constructs are translated to a sequence of machine instructions.

Race Condition

- The increment and the decrement of the count of queue elements, `qP -> count` (not public), may not be done by a single machine instruction. So the increment operation is interruptible.
- It is possible that the increment operation of the producer process is interrupted immediately after the `count` is read from the memory to a CPU register.

Race Condition

- Before the data update takes place in the memory, `count` is also read for the decrement operation by the consumer process[a].
- Both the processes have the same `qP -> count` value say n. One will increment it to $n+1$ and the other will decrement it to $n-1$.

---

[a] There is a context switch or the consumer is running in parallel.

Race Condition

- Finally both the processes will write in the memory location. The value of the one who writes last will remain.
- But both the situations[a] are incorrect.
- After producing a new data and consuming an old data the `qP -> count` should remain at n.

---

[a] The consumer is writing last or the producer is writing last.

Race Condition

- Normally a machine instruction is uninterruptible i.e. an interrupt is processed only after the completion of the current instruction.
- On a uniprocessor system if increment and decrement of a memory location is performed by one machine instruction, the previous race condition will not arise.

Race Condition

- But on a multi core/processor system the race condition cannot be avoided even if data update is done by one instruction, unless the memory location is locked.
- The producer and the consumer processes running in parallel[a] can access the same memory location concurrently, and read the data on two different memory cycles.

---

[a] On different cores/processors

Race Condition

- The breakdown of atomicity of instruction execution can be restored by locking the memory location implicitly[a] or explicitly by using a memory lock.
- Essentially, the memory update should be mutually exclusive.

---

[a] Any memory access instruction will lock the location.

Race Condition

- A race condition may occur in a user program having multiple threads or processes. It may also take place while executing the kernel code.
- There may be several processes running the kernel mode in a system.

Race Condition

- A race conditions in kernel mode may occur while modifying the kernel data structures. Such data structures are related to memory allocation, list of PCBs etc.
- As an example two different processes may give fork() call, enter the kernel mode and update same set of data structures concurrently.

Preemptive Kernel

- A kernel may be preemptive or non-preemptive.
- On a non-preemptive kernel, a user process running in the kernel mode cannot be interrupted (interrupts are disabled).
- So it should be free from race condition on a uniprocessor system.

Preemptive Kernel

- But in an SMP or multi-core system disabling interrupt for all processors or cores may not be possible or even if it is, it will be costly.
- In such a situation hardware supported locking will be used to protect integrity of kernel data structures.

Race in Producer-Consumer

- To show how the race condition is still present in our producer-consumer problem, we amplify it.
- We inject delay within the increment and the decrement operations of data count in `addQ()` and `deleteQ()` methods.

Delay in `addQ()`

```
int queue::addQ(int n){
    int temp, i;
    if(isFullQ()) return ERROR;
    rear = (rear + 1) % MAX;
    data[rear] = n;
//      count = count+1;
    temp = count;
    for(i=1; i<= 500000; ++i); // Delay
    temp = temp+1;
    count = temp;
```

```
        return OK;
}
```

Delay in `deleteQ()`

We drop the `frontQ()` function and modify the `deteteQ()` function.

```
int queue::deleteQ(int &v){
    int temp, i;
    if(isEmptyQ()) return ERROR;
    v = data[(front+1)%MAX] ;
    front = (front+1)%MAX ;
//     count = count - 1;
    temp = count;
```

```
    for(i=1; i<= 500000; ++i); // Delay
    temp = temp-1;
    count = temp;
     return OK;
}
```

Race Again

```
$ $ g++ -Wall prodCon1a.c++ queue1a.o
$ ./a.out
Produced Data 17 1365180540
Produced Data 18 1540383426
Produced Data 19 304089172
Produced Data 20 1303455736
Produced Data 21 35005211
21 data produced
11 data consumed
```

Race Again

```
Consumed Data 9 596516649
Consumed Data 10 1189641421
Consumed Data 11 424238335
Consumed Data 12 719885386
Consumed Data 13 1649760492
10 data produced
13 data consumed
```

As the count value is incorrect, the consumer is reading old data.

Critical Section of Code

- A set of cooperating processes are running on a system.
- In the code of each process there may be sequences of instructions that update a shared data.
- These sequences are called critical sections of code.

Critical Section of Code

- No two process should run their critical sections of code related to a shared data concurrently. It is essential for the purpose of data integrity.
- Each cooperating process must follow a protocol before entering a critical section and also after leaving it.

Critical Section Protocol

- Before entering a critical section a process must check and ensure that no other process is running in its corresponding critical section of code.
- After leaving a critical section the process must signal its departure.
- Any critical section protocol should satisfy the following conditions.

Critical Section Protocol

- Mutual exclusion (safety): no two process should execute their related critical sections concurrently.
- Progress (liveness): Each process requesting to enter its critical section eventually must get its chance.

Critical Section Protocol

- Bounded waiting (weak fairness): A requesting process P_i may have to wait to enter its critical section as other processes are entering (and leaving) their critical sections. But there should be a bound on the number of entries by other processes before the entry is granted to P_i.

Critical Section Protocol

Following is a software based critical section protocol for two cooperating processes.

Peterson's Algorithm

- Peterson's algorithm[a] was proposed as a software solution of the critical section problem of two processes[b].
- The algorithm allows alternate execution of critical section of codes by two processes P_0 and P_1 if both of them wishes to enter simultaneously.

---

[a] An improvement over Dekker's algorithm.

[b] But it cannot guarantee correctness on a modern architecture.

Peterson's Algorithm

- Two boolean variables C_0 and C_1 are used to register requests of two process P_0 and P_1 to enter the critical sections.
- The variable `turn` $\in \{0, 1\}$ indicates the turn of the process to enter the critical section when both have registered their requests.

Peterson's Algorithm

For process P_i, $i \in \{0, 1\}$:

```
while (1) {
    $C_i$ = true
    turn = 1-i
    while ($C_{1-i}$ == true and turn == 1-i) wait
          ⋮
        critical section of code
          ⋮
    $C_i$ = false
        non-critical code
}
```

Peterson's Algorithm

The process $P_i,\ i \in \{0, 1\}$ does the following to enter its critical section of code.

- It sets C_i `= true` to register its request.
- It gives priority to the other process P_{1-i} to enter its critical section by setting `turn = 1-i`.

Peterson's Algorithm

- If P_{1-i} (other process) has already registered its request to enter the critical section and the value of `turn` remains to `1-i`[a], then P_i waits on the while-loop.
- If only P_i requests, then C_{1-i} is `false` and P_i enters its critical section.

---

[a] The process P_{1-i} performed `turn = i` before P_i performs `turn = 1-i`.

Peterson's Algorithm

- While P_i is in its critical section, the other process P_{1-i} cannot enter in its critical section as both $C_{1-(1-i)} = C_i$ `== true` and `turn = i (= 1-(1-i))`, set by P_{1-i}.
- P_i leaves its critical section by withdrawing its request, C_i `= false`.
- If P_{1-i} is waiting on while-loop, it can now enter the critical section.

Peterson's Algorithm

- When both processes try to enter, the last one that updates `turn = 1-i` is stopped and P_{1-i} enters its critical section.
- But once P_{1-i} comes out of its critical section and withdraws its request, it cannot enter again in its critical section before P_i.
- Usually C_0 and C_1 are replaced by a 2-element array `flag[2]`.

Mutual Exclusion

- Both P_i and P_{1-i} cannot cross the while-barrier concurrently as that require C_0, C_1 `== true` and also `turn == 0` and `turn == 1`, which is impossible.
- So the mutual exclusion of entering critical sections is guaranteed (not really).

Liveness and Fairness

- Even when both P_0 and P_1 request to enter the critical section and one enters first. The next turn will be for the other one.
- The wait time for P_i is the time of execution of critical section of P_{1-i}.

Note

- What will happen if we exchange the order of assignments of C_i and `turn`?

  C_i `= true`

  `turn = 1-i`

  is replaced by

  `turn = 1-i`

  C_i `= true`

- Sequential semantics of these two codes are not different as C_i and `turn` are independent.

Note

- But when two processes P_0 and P_1 are running concurrently, the mutual exclusion of entering critical section of codes are not guaranteed by the second version.
- There may be the following sequence of execution of code in P_0 and P_1.

Note

Initially both C_0 and C_1 are `false`. The code for P_i is

`turn = 1-i`

C_i `= true`

`while (`C_{1-i} `== true and turn == 1-i) wait`

Following is an execution sequence:

1. P_1: `turn = 0`
2. P_0: `turn = 1`

Note

5. P_0: C_0 `= true`

6. P_0: C_1 `== true and turn == 1` is `false`.
   P_0 enters its critical section.

7. P_1: $C_1 =$ true.

8. P_1: C_0 `== true and turn == 0` is `false`.
   P_1 enters its critical section.

Out of Order Execution

- Machine instructions are reordered by the compiler and also by the hardware for better performance.
- This may create trouble for Paterson’s algorithm.

Shared Variables

- The variables C_0, C_1 and `turn` are accessed by both the process P_0 and P_1.
- So these variables are bound to some shared memory region.
- Following is an implementation of Peterson's algorithm on producer-consumer problem.

Shared Variable

In `prodConPeterson.c++`

```
.............
int *turnP;       // Peterson var
bool *c0P, *c1P; // Peterson vars
.............
turnP = countC+1;              // Peterson
c0P = (bool *)(turnP+1);       // variables
c1P = c0P+1;                   // in shared memory
*c0P = *c1P = false;
```

Queue Header File

In `queue1b.h`

```
extern int *turnP;
extern bool *c0P, *c1P;
```

Critical Sections

- Critical section of code are in two methods of the queue.
- The producer uses `addQ()` and the consumer uses `deleteQ()`.
- In `addQ()` the critical section is increment of the counter.
- In `deleteQ()` the critical section is decrement of the counter.

Critical Sections

- Peterson's entry and exit codes are used to make these two operations logically atomic with respect to producer and consumer processes.
- Following are the modified codes of `addQ()` and `deleteQ()`

addQ()

In queue1b.c++

```
int queue::addQ(int n){
    int temp, i;
    if(isFullQ()) return ERROR;
    rear = (rear + 1) % MAX;
    data[rear] = n;
//     count = count+1;
    *c0P=true; *turnP=1;
    while(*c1P && *turnP == 1);
```

```
        temp = count;
        for(i=1; i<= 500000; ++i); // Delay
        temp = temp+1;
        count = temp;
    *cOP=false;
    return OK;
}
```

deleteQ()

```
int queue::deleteQ(){
    int temp, i;
    if(isEmptyQ()) return ERROR;
    front = (front+1)%MAX ;
//     count = count - 1;
  *c1P=true; *turnP=0;
  while(*c0P && *turnP == 0);


      temp = count;
```

```
       for(i=1; i<= 500000; ++i); // Delay
       temp = temp-1;
       count = temp;
  *c1P=false;
    return OK;
}
```

Architectural Support

- A more general solutions of critical section problem uses architectural support provided by the CPU.
- In a single processor system a critical section of code can be made atomic or uninterruptible if the interrupt is disabled before entering the critical section and enabled after leaving it.

Architectural Support

- But this cannot be used in user mode as disabling interrupt is a privileged instruction.
- This technique can be used on a single processor system with a non-preemptive kernel.
- Preemption is not allowed when a process is running in kernel mode to update kernel data structures.

Architectural Support

- Even in the kernel mode, keeping interrupts disabled for a long time may affect the response in a time critical application.
- It can be used for a short critical section.
- But on a multiprocessor system, it may be complicated and costly to disable interrupt for all processors.

Architectural Support

- Execution of a machine instruction is atomic on a uniprocessor. An interrupt is not serviced before the completion of the current instruction.
- This feature can be used to create a lock for critical sections using some special instructions.

Special Machine Instructions

- Two such instructions are test and set and exchange.
- Usually a memory access by an instruction is not atomic on a multiprocessor system. But they can be made so by locking the memory location.

Test and Set

```
boolean tAs(boolean *lockP){
        boolean temp = *lockP
        *lockP = True
        return temp;
}
```

- A variable `lock` is initialized to False.
- `tAs(&lock)` returns its stored value and sets it to True.

Test and Set

- If more than one processors try to execute `tAs` on the same memory location simultaneously, the hardware ensures that they are executed in some sequence, making the access and update of the location atomic.
- The memory location `lock` is locked during the execution of test and set.

Bit-Test and Set in x86-64

- The instruction '`bts bitPos, bitWord`' (bit-test and set) tests and set the bit specified by the bitPos in the bitWord[a].
- It copies the specified bit of the bitWord in the carry flag (CF) of the program status word (PSW), and sets the bit.

---

[a] There are similar instructions e.g. `bt` (bit test), `btc` (bit-test and complement), `btr` (bit-test and reset).

Bit-Test and Set in x86-64

- On a single CPU the instruction is atomic.
- On a multiprocessor the lock prefix can make it atomic by locking the memory location of `bitWord`.
- If two such instructions are fetched for execution on two different processors in parallel, they will be executed sequentially in some order.

Semantics of Bit-Test and Set

Let the memory location be L and the bit position be i. The semantics of Bit-Test and Set is

`CF =` L_i
L_i `= 1;` where CF is the carry flag.

`bts` and Mutual Exclusion

- Let `bitWord` be the memory location lock initialized to zero (0).
- The critical section entry code for a process P_i is `bts 0, lock`.
- It sets the bit-0 of lock to one (1), and stores the original value of bit-0 of lock in the carry-flag (`CF`).

`bts` and Mutual Exclusion

- The value stored in `CF` is zero (0) if there is no other process in its critical section.
- But if there is a process in the critical section, it has already set the bit-0 of lock to one (1), so `CF` gets set (value 1) after execution of the instruction.
- The process P_i enter its critical section if `CF=0`.

`bts` and Mutual Exclusion

- But P_i loops on its bit-test and set, if `CF=1`.
- The variable `lock` is reset to zero (0) as the exit code of the critical section.
- The mutual exclusion is guaranteed by the atomicity of `bts`.
- The variable `lock` should be in the shared memory.

`bts` and Producer-Consumer

- We introduce `bts` instruction as inline assembly code in the critical sections of counter increment and counter decrement in `addQ()` and deleteQ() respectively.

Shared Variable `lock`

In `prodConTaS.c++`

```
int *lockP; // Global
.....................
lockP = countP+1;
*lockP = 0; // lock initialized to 0
```

Shared Variable `lock`

In `queue1c.h`

```
extern int *lockP;
```

Inline Assembly Code `addQ()`

In `queue1c.c++`

```
__asm__ __volatile__ (
  ".MyLbl1:     \n\t"
  "lock          \n\t"
  "btsl $0, 0(%%rbx) \n\t"
  "jc .MyLbl1     \n\t"
  :
  :"b"(lockP)
  :
```

```
 );
      temp = count;
      for(i=1; i<= 500000; ++i); // Delay
      temp = temp+1;
      count = temp;
 *lockP = 0 ; // make lock = 0
```

The code for `deleteQ()` is also similar.

Exchange or Swap

This instruction may have different form and we take one of them.

```
void swap(int *srcP, int *dstP){
    int temp
    temp = *srcP
    *srcP = *dstP
    *dstP = temp
```

This also is executed atomically.

Exchange or Swap

- A global variable `lock` is initialized to zero (0).
- The first process that executes the instruction before entering its critical section sets the `lock (*dstP)` to one (1)[a] and gets the old content of the `lock` in *srcP.

---

[a] Stored in its local variable `*srcP`.

Exchange or Swap

- When there is already a process (P_i) in its critical section, and another process P_j attempts to enter its critical section, it gets the value one (1) from the lock in its local variable `*srcP`. It waits (spin lock) until the value become zero (0).
- At the end of the critical section the process sets `lock` to zero (0).

Exchange in x86-64

- The instruction '`xchg src, dst`', exchanges the contents of `src` and `dst`.
- In case of memory operand (destination), the instruction implicitly includes a memory lock. So it is atomic even on a multiprocessor system.

Exchange in x86-64

- If two such instructions with the same destination in memory are fetched for execution on two different processors in parallel, they will be executed sequentially in some order.

`xchg` and Mutual Exclusion

- Let the `lock` be a shared memory location among processes. It is initialized to zero (0).
- We load a CPU register say `eax` with one (1) and execute the instruction '`xchg eax, lock`' in the entry codes of the critical sections of concurrent processes.

`xchg` and Mutual Exclusion

- The contents of `lock` and the register `eax` are exchanged. one (1) will be stored in the shared variable `lock`.
- Only the first process P_i executing this instruction, will get a zero (0) in its register `eax`.

`xchg` and Mutual Exclusion

- All other concurrent processes that have executed ‘`xchg eax, lock`’ after P_i and before P_i leaves its critical section will get one (1) in `eax`.
- P_i enters the critical section and other concurrent processes will loop (busy wait) on ‘`xchg eax, lock`’.

Inline Assembly Code `addQ()`

In `queue1d.c++`

```
__asm__ __volatile__ (
    "movl $1, %%eax \n\t"
    ".MyLbl1:      \n\t"
    "xchgl %%eax, 0(%%rbx) \n\t"
    "cmpl $0, %%eax \n\t"
    "jne .MyLbl1      \n\t"
    :
    :"b"(lockP)
```

```
    :"%eax"
);
      temp = count;
      for(i=1; i<= 500000; ++i); // Delay
      temp = temp+1;
      count = temp;
*lockP = 0 ; // make lock = 0
```

`LL/SC` Instruction Pairs

- Load-link (`LL`) and store-conditional (`SC`) are a pair of machine instructions used for process or thread synchronization.
- They do not lock the memory.
- `LL` loads a value from a memory location (M) to a CPU register (R).

`LL/SC` Instruction Pairs

- `SC` is interesting - it updates a value in M and returns one (1) provided M has not been updated since `LL` has read it.
- Otherwise it does not update M and returns zero (0).
- Architecture like Alpha, PowerPC, MIPS, ARM supports it.

Busy Wait

- In both the hardware methods we have discussed, a process asks for a lock before entering its critical section.
- It enters the critical section only if it can acquires the lock.
- Otherwise it loops and tests the lock again until it is released by the process holding it.

Busy Wait

- This is called a busy wait and such a mutex (mutual exclusion) lock is known as a spinlock. A busy wait wastes CPU time.
- Moreover It does not satisfy the fairness (bounded wait) property[a].
- It also requires little bit of low-level programming.

---

[a] One can write algorithm to satisfy this property [SGG].

Busy Wait

- As an alternative a process that waits for the lock may be suspended.
- But that requires intervention by the kernel[a] and the overhead of context switching.
- A busy wait of a process on a uniprocessor system is a waste of the whole time slice of the waiting process.

---

[a] User level thread does not require kernel intervention.

Busy Wait

- Let there be n processes ready to enter their critical sections on a uniprocessor system.
- One of them is preempted in the middle of its critical section.
- Other $n - 1$ processes will waste their entire time slices by busy wait on the lock.

Busy Wait

- On a multiprocessor system if the execution time of a critical section is short, then one process may busy wait on a processor for a short duration while the process holding the lock can finish its critical section on another processor.
- Busy wait is acceptable if the wait time is shorter than the time to switch context.

Kernel Uses Spinlock

- On a multi processor system processes running on different processors may enter the kernel mode and modify a data structure.
- Restricting concurrent access to the data by disabling interrupt is not possible (or costly) in such a situation.
- The kernel (e.g. Linux) may use spinlock in such a situation.

Bounded-Wait Using Exchange

Following algorithm ensures bounded-wait using exchange.

- Let there be n processes $\{P_0, \cdots, P_{n-1}\}$.
- There is an array of global variables `waiting[n]` and a `lock` variable. All are initialized to zero (0).
- `key` is local to each process.

Bounded-Wait Using Exchange

```
while(1){
      waiting[i] = key = 1
      while(waiting[i] && key)
            swap(&key, &lock)
      waiting[i] = 0

      // Critical section of code

      j = (i+1) mod n
      while(j≠i && ¬waiting[j])
            j = (j+1) mod n
      if(j == i) lock = 0
      else waiting[j] = 0
}
```

Bounded-Wait Using Exchange

- If the process P_i wants to enter its critical section, it initializes `waiting[i]` and `key` to one (1) - line-1.
- The process, among those waiting to enter the critical section, that first executes `swap()` gets zero (0) in its `key` and writes one (1) in the the global variable `lock`.

Bounded-Wait Using Exchange

- Let that process be P_i. Its while loop is terminated (line-2-3), and it enters its critical section after changing `waiting[i]` to zero (0).
- No other waiting process (P_j) can enter their critical sections as their respective `key` get one (1) from `lock` and `waiting[j]` is also one (1).

Bounded-Wait Using Exchange

- The process P_i after coming out of its critical section searches for the next process (if there is any) that is waiting to enter the critical section.
- If one such process P_j is found, P_i changes `waiting[j] = 0` (line-10). This enables to terminate the while-loop (line 2-3) of P_j and enter its critical section.

Bounded-Wait Using Exchange

- If there is no other process waiting to enter the critical section the while-loop (line-7-8) is terminated at `j==i`.
- The `lock` is reset to zero (0).
- The bounded-wait is guaranteed as requesting processes get chance in a round-robin manner.

Mutex Lock

- The simplest software tool (API) available for mutual exclusion of critical section is a mutex lock.
- It is a boolean variable that has two states, locked and unlocked.
- A process or a thread acquires a mutex lock before entering the critical section. It releases the lock after coming out of it.

Mutex Lock

- A thread that locks a mutex becomes its owner. Only the owner can unlock a mutex. No other thread can acquire it before it is released.
- It is necessary to initialize a mutex before any use.

Creating Mutex Lock

- We already know how to create a mutex lock.
- We put our inline assembly code in wrapper functions.

myMutex.h

```
// header file for myMutex.c++
#ifndef _MYMUTEX_H
#define _MYMUTEX_H

void myMutexInit(int &);
void myMutexLock(int *);
void myMutexUnlock(int &);

#endif
```

myMutex.c++

```
/*
 * myMutex.c++
 * $ g++ -Wall -c myMutex.c++
 */
#include "myMutex.h"

void myMutexInit(int &lock){
     lock = 0;
}
```

```
void myMutexLock(int *lockP){
    __asm__ __volatile__ (
        "movl $1, %%eax \n\t"
        ".MyLbl1:      \n\t"
        "lock          \n\t"
        "xchgl %%eax, 0(%%rbx) \n\t"
        "cmpl $0, %%eax \n\t"
        "jne .MyLbl1     \n\t"
        :
        :"b"(lockP)
```

```
        :"%eax"
    );
}

void myMutexUnlock(int &lock){
     lock = 0;
}
```

queue1e.c++

```
/*
 queue1e.c++ implementation of int
           queue
*/
#include "queue1d.h"
#include "myMutex.h"
............
int queue::addQ(int n){
    int temp, i;
```

```
    if(isFullQ()) return ERROR;
    rear = (rear + 1) % MAX;
    data[rear] = n;
    myMutexLock(lockP);
        temp = count;
       for(i=1; i<= 500000; ++i); // Delay
       temp = temp+1;
       count = temp;
    myMutexUnlock(*lockP);
     return OK;
} ......
```

Note

- It is possible to build a lock that will avoid busy wait.
- A process that does not get the lock will be suspended.
- If there are more than one such processes, they will be put in a queue.
- One or all of the suspended processes will be ready once the lock is released.

Semaphore

- A semaphore is a synchronization mechanism suggested by Edsger W. Dijkstra in 1962-63.
- It is a shared integer variable that can be initialized and accessed through two atomic operations.
- Often it is maintained by the kernel and system calls are required to perform the operations.

Operations on Semaphore

- A semaphore is initialized to non-negative integer value and the related data structure is prepared.
- P() or wait() operation: the name P() comes from the Dutch word proberen(to test).
- V() or signal() operation: the origin of the name V() is from the Dutch word verhogen (to increase).

Operations on Semaphore

When a thread (process) P_i performs the P() operation on a semaphore s, the value of the semaphore is decremented by one (1) and then it is tested. If the value is less than zero (0), the thread (process) P_i is blocked on s.

Operations on Semaphore

When a thread (process) P_i performs the V() operation on a semaphore s, the semaphore value is incremented by one (1). If the value is ≤ 0, then there must be some other threads (processes) P_j blocked on the semaphore. It is brought to ready state.

Operations on Semaphore

- View a semaphore as a resource counter.
- A thread requests for a copy of the resource by the P() or wait() operation.
- The semaphore value is decremented to indicate that the resource will be allocated to the thread.
- The thread is blocked if the resource is currently not available.

Operations on Semaphore

- If no resource is available i.e. the initial value of the semaphore is ≤ 0, the requesting thread is blocked.
- For each semaphore, a queue of blocked threads is maintained. The magnitude of the negative value of the semaphore indicates the number of blocked threads on it.

Operations on Semaphore

- When a thread relinquishes the resource, it performs a V() operation. If there is a suspended thread on the semaphore, that is brought to ready queue.
- Atomicity of P() and V() operations are not difficult to implement. But suspension and wakeup of thread requires OS kernel intervention.

Binary and Counting Semaphores

- If the values taken by a semaphore ranges over $\{0, 1\}$, it is called a binary semaphore.
- A counting semaphore can take integer values.
- A binary semaphore can be used as a mutex lock.

An Example

- Let there be two copies of some resource and the counting semaphore s is initialized to two (2).
- There are two processes P_1 and P_2. They perform semaphore operations as follows:

```
P1:                 P2:
  s.wait();           s.wait(); s.wait();
  s.signal();         s.signal(); s.signal();
```

An Example

Let one execution sequence be as follows:

Execute	`s.val`	Queue	P_1	P_2
Init	2	[]	ready	ready
P_1 :`s.wait()`	1	[]	running	ready
P_2 :`s.wait()`	0	[]	running	running
P_2 :`s.wait()`	-1	$[P_2]$	running	blocked
P_1 :`s.signal()`	0	[]	running	ready
P_2 :`s.signal()`	1	[]	finished	running
P_2 :`s.signal()`	2	[]	finished	running

Semaphore Data Structure

A conceptual data type of counting semaphore is as follows:

```
typedef struct {
        int count; // boolean for binary sem
        semQ queue;
} semaphore;
semaphore s;
```

The queue is for the blocked processes on the semaphore.

Initialization of Semaphore

```
void initSem(semaphore *sP, int val) {
          sP -> count = val;
          sP -> queue = EMPTY;
}
```

Binary Semaphore Operations

These operations are atomic.

```
void P(semaphore *sP) {
     if(sP->count==1) sP->count=0;
     else {
        addQ(sP->queue, process);
        block the process;
     }
}
```

```
void V(semaphore *sP) {
     sP->count = 1;
     if(!isEmpty(sP->queue)) {
        addQ(readyQ, frontQ(sP->queue)) ;
        deleteQ(sP->queue) ;
     }
}
```

Counting Semaphore Operations

These operations are atomic.

```
void P(semaphore *sP) {
     sP->count--;
     if(sP->countr < 0) {
        addQ(sP->queue, process);
        block the process;
     }
}
```

```
void V(semaphore *sP) {
     sP->count++;
     if(sP->count <= 0) {
        addQ(readyQ, frontQ(sP->queue)) ;
        deleteQ(sP -> queue);
     }
}
```

Different Usage of Semaphore

There are two essential usage of semaphore.

- Guarding critical sections of code for execution in mutual exclusion.
- Creating synchronization point or a barrier in the path of execution.

Mutual Exclusion of Critical Sections

A simple example is as follows where a semaphore is used as a mutex lock.

```
Initialization: semaphore s = 1;
Proc-I              Proc-II
....                ....
s.P()               s.P()
Critical section    Critical section
s.V()               s.V()
....                ....
```

Synchronization Point-I

```
Initialization: semaphore s = 0;
Proc-I          Proc-II
....            ....
  s.P()           s.V()
l1:             l2:
```

Process-I cannot cross `l1` unless process-II crosses `l2`.
Typical examples are join of thread, `waitpid` of process.

Synchronization Point

```
Initialization: semaphore s1 = 0, s2 = 0;
Proc-I          Proc-II
....            ....
s1.V()          s2.V()
s2.P()          s1.P()
l1:             l2:    ... synch-point ...
```

Process-I cannot go beyond `l1` unless process-II crosses `l2` and vice versa.

Semaphore on Linux

- Two types of semaphore implementations and their APIs are available on Linux platform.
- They are old System V semaphore and POSIX (Portable Operating System Interface) semaphore
- We start with the API of System V semaphore.

System V Semaphore: Basic Operations

- Get[a] a semaphore set: `semget()`.
- Initialize the elements of the semaphore set using `semctl()`.
- Perform semaphore operations on the set elements: semop().
- At the end remove the semaphore set: `semctl()`.

---

[a] Create a new one or open an existing one.

`semget()`

`int semget(key_t key, int n, int flag)`

- The system call to get a semaphore set.
- The first parameter specifies an IPC key returned by `ftok()` or often `IPC_PRIVATE`.
- The parameter n specifies the number of semaphores in the set ($\{0, \cdots, n-1\}$) when it is created[a].

---

[a] Less than or equal to the number of semaphores in an existing semaphore set.

- If the `flag` is not `IPC_CREAT`, no new semaphore is created. The flag also holds the read-write permission bits.
- A typical `semget()` call is,

```
#define NUM_SEMS 2
#define PERM (0644)
int semid ;
semid = semget(IPC_PRIVATE, NUM_SEMS,
                          IPC_CREAT | PERM);
```

`semctl()`

```
int semctl(int semid, int ind, int cmd,
...);
```

- This system call performs different control operations on the semaphores of the set.
- The first argument is the semaphore identifier of the semaphore set.
- The second argument is the index of a semaphore in the set.

- The third argument is a command and the fourth argument depends on it.
- The fourth argument:

```
union semun {
  int val;  /* Value for SETVAL */
  struct semid_ds *buf;
  unsigned short  *array;
  struct seminfo  *__buf;
};
```

`semctl()`

Following are two typical calls to `semctl()`.

- `semctl(semID, 0, SETVAL, 1);` - sets the value of the 0^{th} semaphore of semID to 1.
- `semctl(semID, 0, IPC_RMID) ;` - removes the semaphore set of `semID`.

`semop()`

```
int semop(int semid, struct sembuf
opsPtr[], unsigned int nOps);
```

- This system call is used to perform operations on the semaphores of the set identified by `semid`..
- It can perform one or more operations specified by `nOps` elements of the array of `struct sembuf` pointed to by `opsPtr`.

`semop()`

- The pointer `opsPtr` to an array of structures specify different operations on different semaphores of the set.
- The operations are performed `atomically` and in the order of the elements of `opsptr[]`.

`semop()`

- The fields of struct sembuf are

```
struct sembuf {
        ushort  sem_num ; // semaphor index
        short   sem_op ;  // operation
        short   sem_flg ; // operation flag
};
```

- Each array element specifies an operation on an element of the semaphore set.

Semaphore Values

- Each semaphore of the set is associated with following set of data:

```
unsigned short  semval; /* semaphore value */
unsigned short  semzcnt;/* # waiting for zero *
unsigned short  semncnt;/* # waiting for increa
pid_t           sempid; /* ID of process that
```

Semaphore Values

- `semval` is the value of the semaphore.
- `semzcount` is the count of threads waiting on the semaphore for its value to be zero (0).
- `semncnt` is the count of threads waiting on the semaphore for its value to increase.

`semop()`

- If `sem_op` $>$ `0` (resource released): its value is added to `semval`[a]. This may awaken a process waiting on the semaphore to decrease its value (to get resource).
- If `sem_op` $=$ `0`, the `semval` is checked for zero (0)[b]. If it is, the call is completed, else it waits-for-zero[c].

---

[a] The process must have write permission.

[b] The process must have read permission

[c] Increments semzcnt, count of threads waiting for semaphore value to be zero (0).

`semop()`

- If `sem_op` $<$ `0`: if `semval` $\geq$ `|sem_op|`, subtract `|sem_op|` from `semval`, and the operation is complete (required resource obtained). Otherwise the `semncnt` (count of threads waiting on this semaphore value to increase) is incremented by one, the operation (process) is blocked until `semval` becomes $\geq$ `|sem_op|`.

Binary `P()` Operation

```
static int P(int semID) {
     struct sembuf buff ;

     buff.sem_num = 0 ; // On the 0th element
     buff.sem_op  = -1 ;
     buff.sem_flg = 0 ;
     if(semop(semID, &buff, 1) == -1) {
       cerr << "semop P operation error\n" ;
       return -1;
```

```
     }
     return 0 ;
}
```

Binary V() Operation

```
static int V(int semID) {
     struct sembuf buff ;

     buff.sem_num = 0 ;  // On the 0th element
     buff.sem_op  = 1 ;
     buff.sem_flg = 0 ;
     if(semop(semId, &buff, 1) == -1) {
        cerr << "semop V operation error\n" ;
        return -1 ;
```

```
     }
     return 0 ;
}
```

System V Semaphore as Mutex Lock

```
/*
 * semSysV1.c++ shows the use of System V
 *               semaphore as mutex lock
 * $ g++ -Wall semSysV1.c++
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#define FLAGS (0644)

static int P(int semID) {
     struct sembuf buff ;

     buff.sem_num = 0 ; // On the 0th element
```

```
        buff.sem_op  = -1 ;
        buff.sem_flg = 0 ;
        if(semop(semID, &buff, 1) == -1) {
          cerr << "semop P operation error\n" ;
          return -1;
        }
        return 0 ;
}

static int V(int semID) {
        struct sembuf buff ;
```

```
     buff.sem_num = 0 ;  // On the 0th element
     buff.sem_op  = 1 ;
     buff.sem_flg = 0 ;
     if(semop(semID, &buff, 1) == -1) {
        cerr << "semop V operation error\n" ;
        return -1 ;
     }
     return 0 ;
}
```

```
int main() {
  int semID, chID ;

  if((semID = semget(IPC_PRIVATE, 1,  IPC_CREAT
    cerr << "semget() fails\n" ;
    exit(1) ;
  }
  semctl(semID, 0, SETVAL, 1);
  chID = fork();
  if(chID == -1){
   cerr << "fork() fails\n";
```

```
        exit(1);
    }
    if(chID > 0) { // Parent
        int status ;

        for(int i=0; i<=5; ++i) {
            //P(semID) ;
            cout << "Indian Institute of";
            fflush(stdout);
            sleep(2);
            cout << " Information Technology\n";
```

```
      //V(semID) ;
      sleep(1) ;
  }
  waitpid(chID, &status, 0) ;
  semctl(semID, 0, IPC_RMID) ;
}
else { // Child
  for(int i=0; i<=5; ++i) {
      //P(semID) ;
      cout << "Allahabad ";
      fflush(stdout);
```

```
          sleep(1);
          cout << "Bhubaneswar ";
          fflush(stdout);
          sleep(1);
          cout << "Kalyani\n";
          //V(semID) ;
          sleep(2) ;
      }
    }
    return 0 ;
}
```

POSIX Semaphore on Linux

- Both threads and processes can be synchronized using this semaphore.
- The value of a semaphore is a non-negative integer.
- The P() operation is called `sem_wait()` and the V() operation is called `sem_post()`.
- Other operations are create, initialize, get the value of and remove a semaphore.

POSIX Semaphore on Linux

- There are two forms of POSIX semaphores available on Linux. One is named and the other is unnamed.
- A named semaphore is identified by a name of the form `/xyz`.
- A named semaphore is kernel persistent i.e. once created it remains in the the system until shutdown.

POSIX Semaphore on Linux

- An unnamed or memory-based semaphore does not have a name.
- It resides in the shared memory between threads (global data) or processes (shared memory).
- This requires to be initialized explicitly before use and deallocated after use.

POSIX Named Semaphore as Mutex Lock

- Following example shows how a portion of code is made atomic using a named semaphore.
- The semaphore is used as a mutex lock. The output of two processes (parent and child) are intermingled when the semaphore is not used.

POSIX Named Semaphore as Mutex Lock

```
/*
 semaphorePOS1.c++ shows the use of POSIX
       named semaphore as mutex lock
 $ g++ -Wall semaphorePOS1.c++ -lpthread
*/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <semaphore.h>
#include <errno.h>

int main() {
    int i, cPID, status;
    sem_t *sP;
```

```
sP = sem_open("/abcd", O_CREAT, 0777, 1);
if(sP == SEM_FAILED && errno != EEXIST){
   cerr << errno << " Semaphore error\n";
   exit(1);
}
cPID = fork();
if(cPID == -1){
  cerr << "fork() fails\n";
  exit(1);
}
if(cPID != 0) { // Parent
```

```
for(i=1; i<=5; ++i) {
   // sem_wait(sP); // get lock
   cout << "Indian Institute of";
   fflush(stdout);
   sleep(2);
   cout << " Information Technology\n";
   // sem_post(sP); // release lock
   sleep(1);
}
waitpid(cPID, &status, 0);
sem_close(sP);
```

```
}
else { // Child
   for(i=1; i<=5; ++i) {
      // sem_wait(sP); // get lock
      cout << "Allahabad ";
      fflush(stdout);
      sleep(1);
      cout << "Bhubaneswar ";
      fflush(stdout);
      sleep(1);
      cout << "Kalyani\n";
```

```
            // sem_post(sP); // release lock
            sleep(2);
        }
    }
    return 0 ;
}
```

Output without Lock

```
$ a.out
Indian Institute of Allahabad Bhubaneswar  Inform
Kalyani
Indian Institute ofAllahabad  Information Technol
Bhubaneswar Indian Institute ofKalyani
 Information Technology
Allahabad Indian Institute ofBhubaneswar Kalyani
 Information Technology
Indian Institute ofAllahabad Bhubaneswar  Informa
```

`Kalyani`

`Allahabad Bhubaneswar Kalyani`

Output with Lock

```
$ a.out
Indian Institute of Information Technology
Allahabad Bhubaneswar Kalyani
Indian Institute of Information Technology
Allahabad Bhubaneswar Kalyani
Indian Institute of Information Technology
Allahabad Bhubaneswar Kalyani
Indian Institute of Information Technology
Allahabad Bhubaneswar Kalyani
```

Indian Institute of Information Technology
Allahabad Bhubaneswar Kalyani

Where is the Semaphore

A named semaphore on Linux is created in the directory under `/dev/shm`.

```
$ ls -l /dev/shm
lrwxrwxrwx ... /dev/shm -> /run/shm
$ ls -l /run/shm
total 316
...........
-rwxrwxr-x 32 Aug 15 05:53 sem.abcd
```

Different APIs Used

```
sem_t *sem_open(const char *name,
                int oflag,
                mode_t mode,
                unsigned int value)
```

- `name` is the name of the semaphore of the form `/xyz`. It comes as `/dev/shm/sem.xyz`.
- `oflag` is the control flag (see the man page).

Different APIs Used

- `mode` is for permission bits.
- `value` is the initial value of the semaphore.
- The `mode` and `value` are ignored if the semaphore name already exists and `oflag` is set to `O_CREAT`.
- On success, the return value is the address of the semaphore.

Different APIs Used

`int sem_wait(sem_t *sem)`

- Decrements the semaphore pointed by sem if it is greater than zero (0), and proceeds.
- If it is zero (0), then the call blocks until the value of the semaphore is greater than zero (then it can be decremented).
- It returns zero (0) on success.

Different APIs Used

`int sem_post(sem_t *sem)`

- Increments (unlock) the semaphore pointed by `sem`.
- If there is a process or thread in the queue of this semaphore, blocked on a `sem_wait()` call, it will proceed to decrement (lock) the semaphore.

POSIX Unnamed Semaphore as Mutex Lock

- Unnamed semaphore for process lives in a shared memory.
- So a shared memory segment is created using `shmget()` and is attached to a pointer of type semaphore type, `sem_t`.
- It is initialized using `sem_init()`.

`sem_init()`

```
int sem_init(sem_t *sP, int shrd,
unsigned int val)
```

- If `shrd` is zero (0), the semaphore is shared between threads. In that case `sP` points to a global or heap memory location.
- If `shrd` is not zero ($\neq 0$), it is shared between processes, and `sP` points to a shared memory location.

Unnamed Semaphore Persistence

- The persistence of a semaphore defined on a global or a heap variable is the life-time of the process.
- The persistence of a shared memory semaphore is the life-time of the shared memory. It may be up to the shut-down of the system.

POSIX Unnamed Semaphore as Mutex Lock

- The use of `sem_wait()` and `sem_post()` are same.
- Finally we close and remove the semaphore by `sem_close()` and `sem_destroy()`.
- The value of a semaphore can be extracted by `int sem_getvalue(sem_t *sem, int *sval)` through sval.

Semaphore and Deadlock

Consider the situation with two binary semaphores s_1 and s_2. They are used by two processes P_1 and P_2 in the following way.

	P_1	P_2
	wait(s_1)	wait(s_2)
	$\vdots$	$\vdots$
C:	wait(s_2)	wait(s_1)
	$\vdots$	$\vdots$
	signal(s_2)	signal(s_1)
	$\vdots$	$\vdots$
	signal(s_1)	signal(s_2)

Semaphore and Deadlock

- If both the processes reach the point C, none of them can move any further.
- The situation is called a deadlock.
- It is necessary to design a system so that no deadlock can occur.
- It is also necessary to detect, if it ever occurs.

Producer-Consumer on pthread

- Following code shows an implementation of the producer-consumer problem using two threads.
- The bounded buffer (circular queue) is implemented as a global data.

Producer-Consumer on pthread

- We may use locking facility available in pthread to ensure mutual exclusion of critical sections.
- We also can use busy wait using machine instruction xchg.

Producer-Consumer on pthread

```
/*
 * prodConPth1.c++ Producer-Consumer Problem on
 *         pthread
 * $ g++ -Wall prodCon1.c++ queuePth1.o -lpthread
*/
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <pthread.h>
```

```
#include "queuePth1.h"

void *thread1(void *);
void *thread2(void *);

void producer(queue *);
void consumer(queue *);

int countP = 0, countC = 0;

int main(int count, char *vect[]) {
```

```
pthread_t thID1, thID2; // thread ID
queue *qP;

qP = new queue;
pthread_create(&thID1, NULL, thread1, (void
pthread_create(&thID2, NULL, thread2, (void
pthread_join(thID1, NULL);
pthread_join(thID2, NULL);
cout << countP << " data produced\n";
cout << countC << " data consumed\n";
return 0;
```

```
}


void *thread1(void *p){
     queue *qP = (queue *)p;
     producer(qP);
     pthread_exit(NULL);
}


void * thread2(void *p){
     queue *qP = (queue *)p;
     consumer(qP);
```

```
     pthread_exit(NULL);
}

void producer(queue *qP){
     int added = 1, i ;

     for(i=1;i<=500000;++i) {
        int data, err;

        if(added) {
           data = rand() ;
```

```
                    added = 0 ;
                }
                err = qP->addQ(data) ;
                if(err == OK) {
                    added = 1 ;
                    cout << "Produced Data "
                         << ++countP
                         << " " << data << "\n" ;
                }
            }
        }
```

```
void consumer(queue *qP) {
     int i ;

     for(i=1; i<= 500000; ++i) {
        int data, dataOK;

        dataOK = qP -> frontQ(data);
        if(dataOK == OK){
           qP -> deleteQ() ;
           cout << "\tConsumed Data "
```

```
                    << ++countC
                    << " " << data << "\n" ;
            }
        }
}
```

Producer-Consumer on pthread

```
/*
 queuePth1.c++ implementation of int
           queue
 $ g++ -Wall queuePth1.c++ -c
*/
#include "queuePth1.h"

pthread_mutex_t makeAtomic = PTHREAD_MUTEX_INITIA
............................
```

```
int queue::addQ(int n){
    int temp, i;
    if(isFullQ()) return ERROR;
    rear = (rear + 1) % MAX;
    data[rear] = n;
//    count = count+1;
    pthread_mutex_lock(&makeAtomic);
      temp = count;
      for(i=1; i<= 100000; ++i); // Delay
      temp = temp+1;
      count = temp;
```

```
    pthread_mutex_unlock(&makeAtomic);
    return OK;
}
```

Similar modification in `deleteQ()`.

`pthread_mutex_lock`

- `int pthread_mutex_lock(pthread_mutex_t *mutexP)` locks the mutex object referenced by `mutexP` if it is not already locked.
- If the mutex object is already locked, the calling thread is blocked until the object is released.

`pthread_mutex_unlock`

- `int pthread_mutex_unlock(pthread_mutex_t *mutexP)` unlocks the mutex object referenced by `mutexP` if it is locked.
- If there are threads blocked on the mutex object, one of them may own the lock depending on the scheduling policy.

C++11 Thread and Mutex

- The language C++11 has support for threads, mutual exclusion etc.
- Following is a code for producer-consumer problem.

queueC11.h

```
........
#include <thread>
#include <mutex>
.......
```

Other things are similar.

queueC11.c++

```
/*
 $ g++ -Wall -std=c++11 queueC11.c++
                        -lpthread -c
*/
#include "queueC11.h"
mutex counterMutex;
........
int queue::addQ(int n){
    int temp, i;
```

```
     if(isFullQ()) return ERROR;
     rear = (rear + 1) % MAX;
     data[rear] = n;
//     count = count+1;
       counterMutex.lock();
       temp = count;
       for(i=1; i<= 100000; ++i); // Delay
       temp = temp+1;
       count = temp;
       counterMutex.unlock();
     return OK;
```

```
}
```

```
. . . . . . . . . . . . . .
```

Similar change in `deleteQ()`. Other things are similar.

prodConC11.c++

```
/*
 $ g++ -Wall -std=c++11 prodConC11.c++
                 queueC11.o -lpthread
*/
#include <iostream>
#include <thread>
using namespace std;
#include "queueC11.h"
```

```
void producer(queue *);
void consumer(queue *);

int countP = 0, countC = 0;

int main(int count, char *vect[]) {
    queue *qP;

    qP = new queue;
    thread th1(&producer, qP);
    thread th2(&consumer, qP);
```

```
    th1.join();
    th2.join();
    cout << countP << " data produced\n";
    cout << countC << " data consumed\n";
    return 0;
}
```

Producer and consumer codes are similar to earlier examples.

C++11 Thread and Mutex

- `thread` is a class, it represents a single thread of execution.
- `mutex` is also a class, it is used to protect shared data.

Reformulating Producer-Consumer Problem

- The producer-consumer problem can be formulated and solved using two counting and a binary semaphores.
- The binary semaphore is a mutex lock.
- But the test for queue-full and queue-empty are replaced by two counting semaphores full[a] and empty[b].

---

[a] Initialized to zero (0)

[b] Initialized to N, the size of the queue.

Initialization

```
semaphore lock = 1, empty = N, full = 0;
```

Producer

```
empty.wait();    // Reducing empty by 1
lock.wait();     // Mutex lock
// Produce item and put in queue
lock.signal()    // Mutex unlock
full.signal()    // Increasing full by 1
```

Consumer

```
full.wait();    // Reducing full by 1
lock.wait();    // Mutex lock
// delete from queue.
lock.signal()   // Mutex unlock
empty.signal()  // Increasing empty by 1
```

Code: `prodConSem.c++` `queue1f.h` `queue1f.c++`

Two Usage of Semaphore

- In this version of implementation of producer-consumer problem we see two distinct usages of semaphores.
- The `lock` is used for mutual exclusion.
- But `full` and empty are used to communicate the change of state of the queue in one process to the other process.

Two Usage of Semaphore

- `full.signal()` from producer informs the consumer that the queue is more-full.
- If the consumer is suspended on `full.wait()` (empty queue), it is ready for execution after `full.signal()`.
- Similar is the case for `empty.signal()`.

Reader-Writer Problem

- Another classic synchronization problem is the readers-writers problem.
- There are a few readers (process/thread) and a few writers.
- A reader reads data from the database but does not update it. But a writer can read and write.

Reader-Writer Problem

- The restriction is natural. When a writer is active on the data, no reader or any other writer can be active i.e. a writer has an exclusive access to data.
- But more than one reader can be active simultaneously.

Reader-Writer Problem: Solution I

- In case of simultaneous access request to the database, different versions of solutions are presented with different priorities of reader and writer.
- When no writer is not writing, any number of reader may go on reading.
- In this policy a waiting writer may starve.

Reader-Writer Problem: Solution II

- The priority may be given to the writer.
- Once a writer is ready to write, no more reader can start reading afresh.
- This can make waiting readers starve.

First Solution

Two semaphores and a shared counter are used to implement this version of solution where readers have priority.

```
semaphore mutex = 1, countsem = 1;
int readerCount = 0;
```

Both the semaphores are essentially mutex locks.

First Solution

- The semaphore `mutex` is used for mutual exclusion of entering critical sections between a reader and a writer and also between two writers.
- The shared variable `readerCount` keeps track of the number of readers.
- `countsem` makes the increment and decrement operations on `readerCount` atomic.

Writer Process

```
P(mutex) ;
    /* CS: writer writes */
V(mutex) ;
```

A writer enters its critical section if there is no other writer or reader in their critical sections.

Reader Process

```
P(countsem) ;
++readerCount ;
if(readerCount == 1) P(mutex);
V(countsem) ;
     /* CS: reader reading */
P(countsem) ;
--readerCount ;
if(readerCount == 0) V(mutex) ;
V(countsem) ;
```

Reader Process

```
P(countsem);
++readerCount;
if(readerCount == 1) P(mutex);
V(countsem);
```

- The code makes the increment of reader count atomic using the binary semaphore `countsem`.

Reader Process

- If there is a writer in its critical section, there cannot be any reader in its critical section. The `readercount` is zero (0) at that point.
- During this period, if any reader wishes to enter making `readercount == 1`, it will be suspended at `P(mutex)`.

Reader Process

- If there is no writer in its critical section, the first reader before entering its critical section will acquire the lock by `P(mutex)`.
- No writer can enter its critical section after this as long as there are readers.
- But other readers can enter their critical sections[a].

---

[a] This is the source of starvation of the writers as readers may go on entering.

Reader Process

```
P(countsem);
--readerCount;
if(readerCount == 0) V(mutex);
V(countsem);
```

- The code makes the decrement of reader count atomic using the binary semaphore `countsem`.

Reader Process

- Only when the last reader leaves, a writer can enter.
- A writer may have to wait indefinitely (starvation) if readers keep on coming.
- But a reader may enter immediately once the writer leaves.

Reader Priority

```
rederWriter1.c++ Reader priority,
                 starvation of writer
$ g++ -Wall readerWriter1.c++ -lpthread
```

Second Solution

- In this solution a writer gets the priority.
- Once a writer wishes to enter its critical section no more readers are allowed to enter.
- This may lead to some kind of starvation of readers as concurrent reading may not be possible.

Second Solution

Four semaphores and two shared counters are used to implement this version.

```
semaphore mutex = rCmutex = wCmutex = try = 1;
int rCount = wCount = 0;
```

- The role of `mutex` is same as before. Mutual exclusion between reader-writer and also between two writers.
- The counters `rCount` and wCount are used for reader and writer counts respectively.

Second Reader-Writer Problem

- The semaphores `rCmutex` and `wCmutex` are used to make increment and decrement operations atomic on `rCount` and `wCount` respectively.
- The semaphore `try` prohibits a new reader to enter after a writer is trying to do so. The reader is permitted only after the writer finishes.

Reader Process

```
P(try);     // reader tries to reg.
P(rCmutex); // no race between readers
++rCount;   // increment reader count
if(rCount == 1) P(mutex); // enter CS or wait
V(rCmutex); // another reader may reg.
V(try);     // reader or writer may try

// Readers Critical Section
```

```
P(rCmutex); // a reader wishes to leav
--rCount;   // decrement reader
if(rCount == 0) V(mutex); // waiting writer
                          // gets chance
V(rCmutex); // aother reader may reg.
```

Writer Process

```
P(wCmutex); // one writer reg.
++wCount;
if(wCount == 1) P(try);
            // enter CS or wait
V(wCmutex); // another writer may reg.

P(mutex); // only one writer
// Critical section
V(mutex);
```

```
P(wCmutex); // writer leavs
--wCount;
if(wCount == 0) V(try); // others may try
V(wCmutex); // another writer may reg.
```

Reader Process

- The entry section is completed one-reader at a time.
- If there is no request from any writer any number of reader may enter their critical sections.

Reader Process

- But even if there is a reader in the critical section, and another reader is permitted to enter, it may be blocked at `P(try)`, by a writer who has already attempted to enter its critical section.
- The action `V(try)` by the writer is performed only after it comes out of the critical section. Only then any following reader may enter.

Writer Process

- `P(try)` stops any further entry of reader until the writer finishes.
- The `P(mutex)` and `V(mutex)` pair prohibits any other reader or writer to be in their critical sections.

Producer-Consumer Revisited

Consider the following simplified code of producer-consumer problem.

```
/*
 * slowProdCon.c++ Producer-Consumer Problem
 *         one is slower than the other
 * $ g++ -Wall slowProdCon.c++  -lpthread
*/
#include <iostream>
using namespace std;
```

```
#include <stdlib.h>
#include <pthread.h>
#include "queuePth1.h"
#define DELAY 10000

int data;
int dataCount=0;



void *producer(void *);
void *consumer(void *);
```

```
pthread_mutex_t makeAtomic = PTHREAD_MUTEX_INITIA

int main(int count, char *vect[]) {
     pthread_t thID1, thID2; // thread ID
     int num;

     cout << "Enter number of data: ";
     cin >> num ;

     pthread_create(&thID1, NULL, producer, (void
```

```
    pthread_create(&thID2, NULL, consumer, (void
    pthread_join(thID1, NULL);
    pthread_join(thID2, NULL);
    return 0;
}

void *producer(void *par){
    int num = *(int *)par;

    for(int i=1; i<=num; ) {
        pthread_mutex_lock(&makeAtomic);
```

```
 cout << "P->  ";
 if(dataCount==0){
            dataCount = 1;
            data=i;
    ++i;
 }
         pthread_mutex_unlock(&makeAtomic);
         for(int j=0; j<DELAY; ++j); // Delay
     }
     pthread_exit(NULL);
}
```

```
void *consumer(void *par){
     int num = *(int *)par;

     for(int i=1; i<=num;){
         pthread_mutex_lock(&makeAtomic);
 cout << "C<- ";
 if(dataCount==1){
            cout << "Data read: " << data << endl
            dataCount=0;
    ++i;
```

```
 }
        pthread_mutex_unlock(&makeAtomic);
        // for(int j=0; j<DELAY; ++j); // Delay
    }
    pthread_exit(NULL);
}
```

Slow Producer: Output

```
$ a.out
Enter number of data: 3
P->  C<- Data read: 1
C<- C<- .......  C<- C<- P->  C<- Data read: 2
C<- C<- ........ C<- C<- P->  C<- Data read: 3
```

Producer-Consumer Revisited

- Even when the producer is not producing any new data, the consumer wastes CPU time by executing its code. This can also be other way round.
- Similarly even when the bounded buffer is full, the producer is in a loop to check for emptiness of the buffer.

Producer-Consumer Revisited

- It is good to have a mechanism so that the consumer is suspended as long as there is no new data, and it will be awakened when a new data is produced by the producer.
- Similarly in case of a bounded buffer, the producer is suspended as long as the buffer is full.

Condition Variable

- A general solution is proposed by introducing the notion of condition variable[a].
- A condition variable (cv) is used by a thread (t_i) to suspend itself when a condition is not satisfied (not a desired state).

---

[a] The name is due to Hoare in connection to his monitors and is similar to private semaphore of Dijkstra.

Condition Variable

- Each condition variable (cv) has its queue of suspended threads waiting for the condition to be true (change of state).
- Before going to suspension t_i may need to release the lock it is holding.
- Another thread (t_j) changes the state as required by the condition variable (cv).

Condition Variable

- The thread (t_j) can awaken one or more threads waiting in the queue of cv.
- If more than one threads wake up and ready, it is essential that each of them checks the condition variable again before start running.

Pthread Condition Variable

```
/*
 * slowProdConCond.c++ Producer-Consumer Problem
 *          one is slower than the other
 * $ g++ -Wall slowProdConCond.c++  -lpthread
*/
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <pthread.h>
```

```
#include "queuePth1.h"
#define DELAY 10000

int data;
int dataCount=0;


void *producer(void *);
void *consumer(void *);

pthread_mutex_t makeAtomic = PTHREAD_MUTEX_INITIA
```

```
pthread_cond_t cvF = PTHREAD_COND_INITIALIZER;  /
pthread_cond_t cvE = PTHREAD_COND_INITIALIZER;  /

int main(int count, char *vect[]) {
     pthread_t thID1, thID2; // thread ID
     int num;

     cout << "Enter number of data: ";
     cin >> num ;

     pthread_create(&thID1, NULL, producer, (void
```

```
    pthread_create(&thID2, NULL, consumer, (void
    pthread_join(thID1, NULL);
    pthread_join(thID2, NULL);
    return 0;
}

void *producer(void *par){
    int num = *(int *)par;

    for(int i=1; i<=num; ) {
        pthread_mutex_lock(&makeAtomic);
```

```
cout << "P->  ";
while(dataCount == 1) // added
             pthread_cond_wait(&cvE, &makeAtomic
// if(dataCount==0){
           dataCount = 1;
           data=i;
   ++i;
// }
pthread_cond_signal(&cvF);  // added
        pthread_mutex_unlock(&makeAtomic);
        // for(int j=0; j<DELAY; ++j); // Delay
```

```
     }
     pthread_exit(NULL);
}


void *consumer(void *par){
     int num = *(int *)par;

     for(int i=1; i<=num;){
         pthread_mutex_lock(&makeAtomic);
 cout << "C<- ";
         while(dataCount == 0)   // added
```

```
                    pthread_cond_wait(&cvF, &makeAtomi
 // if(dataCount==1){
                cout << "Data read: " << data << endl
                dataCount=0;
     ++i;
 // }
 pthread_cond_signal(&cvE);  // added
            pthread_mutex_unlock(&makeAtomic);
            for(int j=0; j<DELAY; ++j); // Delay
      }
      pthread_exit(NULL);
```

```
}
```

Slow Producer with Cond Var: Output

```
$ a.out
Enter number of data: 5
P->  P->  C<- Data read: 1
P->  C<- Data read: 2
P->  C<- Data read: 3
P->  C<- Data read: 4
C<- Data read: 5
```

Condition Variable

- If the buffer (`data`) is full, the producer thread waits on the condition variable `cvE`.
- It also releases the mutex lock.
- Similarly, if the buffer is empty, the consumer thread waits on the condition variable `cvF`.

Condition Variable

- After every item produced, the producer calls `pthread_cond_signal()` to signal the condition variable `cvE`.
- The consumer thread blocked on the condition variable (`cvE`) is awakened[a].

---

[a] In general one of the threads blocked on the condition variable is awakened. There is also a call `pthread_cond_broadcast()` that awakens all blocked threads.

Condition Variable

- Similarly after every item consumed, the consumer calls `pthread_cond_signal()` to signal the condition variable `cvF`.
- The producer thread blocked on the condition variable (`cvF`) is awakened.

Condition Variable

- The consumer thread locks the mutex (`makeAtomic`).
- Then it checks the `dataCount`. If it is zero (0), then the thread will go to sleep but it should also release the mutex (`makeAtomic`) lock so that the producer can enter its critical section[a].

---

[a] This is the purpose of two parameters of `pthread_cond_wait()`.

Condition Variable

- When the consumer is awakened by the signal, it must acquire the mutex (`makeAtomic`) lock before it access the shared data (`dataCount`).
- The call to `pthread_cond_wait()` is in a loop. The reason is, even after awakening, which takes time, the thread may find that the shared variable in a 'wrong' state[a].

---

[a] May be due to other consumer.

High-Level Construct of Synchronization

- People[a] realized that semaphore is a low-level synchronization primitive.
- An incorrect use of it may lead to errors that are difficult to detect.
- So there was a thought about supporting synchronization as a high-level programming language construct.

---

[a] C A R Hoare (Tony Hoare, Sir Charles Antony Richard Hoare) and Per Brinch Hansen and others.

Monitor

- Brinch Hansen[a] and Tony Hoare introduced the concept of monitor.
- A monitor may be viewed as an abstract data type with shared data that can be accessed by different processes.
- The data is private and can be access only by public operations (problem specific) defined within it.

---

[a] First implemented in the Concurrent Pascal.

Monitor

This high-level synchronization construct has the following features:

- Mutual exclusion of using the monitor methods by threads.
- Condition variable: a thread can wait in the queue of a condition variable when certain condition is not satisfied. It relinquishes the exclusive access before suspension.

Monitor

- **Signal**: a signal is sent to the suspended thread(s) when the condition is satisfied. The signal restarts thread(s).
- A monitor has mutex locks for atomic methods and condition variables.

Monitor

- Mutual exclusion is guaranteed (by the compiler) on the operations i.e. only one thread/procedure at a time can execute a monitor operation.
- If a thread t_j tries to execute a monitor operation while another thread t_i is already in the middle of execution of a monitor operation, the thread t_j will be blocked on the monitor.

Monitor

- A condition variable has three (3) atomic operations, `wait()`, signal(), and `signalAll()` (also called `broadcast()`) defined on them.
- There is a queue of suspended threads for every condition variable.

Monitor

- If `x` is a condition variable and a thread t_i in the middle of a monitor operation executes `x.wait()`[a], it is suspended and put in the queue of `x`.
- The thread t_i must either signal a suspended thread within the monitor or release the monitor lock so that another thread t_j can enter the monitor.

---

[a] Some condition is not satisfied e.g. no space in the buffer (full), no element in the buffer (empty)

Monitor

- If the thread t_j while executing a monitor operation performs `x.signal()`, a thread e.g. t_i suspended on the condition variable `x` comes out of suspension[a].
- But then both t_i and t_j cannot be in the ready or running state within the monitor.

---

[a] `x.signalAll()` will release all threads suspended on `x`.

Monitor

- Two possible solutions are suggested.
- Signal and wait: t_j suspends itself until t_i leaves the monitor or gets suspended on another condition variable (Hoare Style.
- Signal and continue: t_j continues and t_i waits for t_j either to leave the monitor or to get suspended on a condition variable (Mesa style).

Monitor for Producer-Consumer

```
monitor ProCon {
        int data[100], front, rear, count ;
        condVar full, empty;

        initProdCon(){
            front = rear = 0;
            count = 0;
        }
```

```
void addQ(int n){
    while(count == 100) full.wait();
    rear = (rear +1)%100;
    data[rear] = n;
    count = count+1;
    empty.signal(); // empty.signalAll()
}
void deleteQ(int *dP){
    while(count == 0) empty.wait();
    *dP = data[(front+1)%100];
    front = (front+1)%100;
```

```
                count = count - 1;
                full.signal();
        }
}
```

Implementing Monitor

- We need a semaphore `mutex` for each monitor class, initialized to one (1).
- Any process must execute `mutex.wait()` before running the code of any procedure (`F()`) on shared data.
- Once the process finishes and there is no suspended process in the monitor, it executes `mutex.signal()`.

Implementing Monitor

- If we adopt the signal and wait semantics for condition variables, the signaling process will wait until the resumed process either finishes or again blocked on condition variable.
- For that we need another semaphore `next` and a counter `nextCount` both initialized to zero (0).

Implementing Monitor

- A signaling process can block itself on `next` by executing `next.wait()` after incrementing `nextCount`.
- After finishing the code of `F()` the process checks whether `nextCount` > 0 and takes the following action.

Invocation of Procedure `F()`

```
mutex.wait();
// body of F()
if(nextCount > 0) next.signal();
else mutex.signal();
```

Note

- If `nextCount` $>$ `0`, there is already some process waiting in the middle of some procedure. They should be restarted.
- Otherwise a new process may enter the monitor.

Condition Variable

- For each condition variable `cV` there is a semaphore `cVsem` and a counter `cVcount` both initialized to zero (0).
- The code for `cV.wait()` and cV.signal() are as follows:

`cV.wait()`

The thread is going to be blocked on `cVsem`.

```
cVcount++;
if(nextCount > 0) next.signal();
                  // release waiting thread
else mutex.signal();  // Allow new thread to ente
cVsem.wait();
cVcount--;            // restarted
```

cV.signal()

```
if(cVcount > 0)){ // if threads waiting on cV
   nextCount++; // this thread will wait on next
   cVsem.signal();
   next.wait();
   nextCount--;    // wait on next is over
}
```

Dining Philosophers Problem

- The original problem was formulated by Edsger Dijkstra in 1965 as an examination problem, where computers are competing for tape drives.
- The current well-known formulation is due to C A R Hoare.

Dining Philosophers Problem

“Five silent philosophers sit at a round table with bowls of spaghetti. Five chopsticks are placed between each pair of adjacent philosophers.

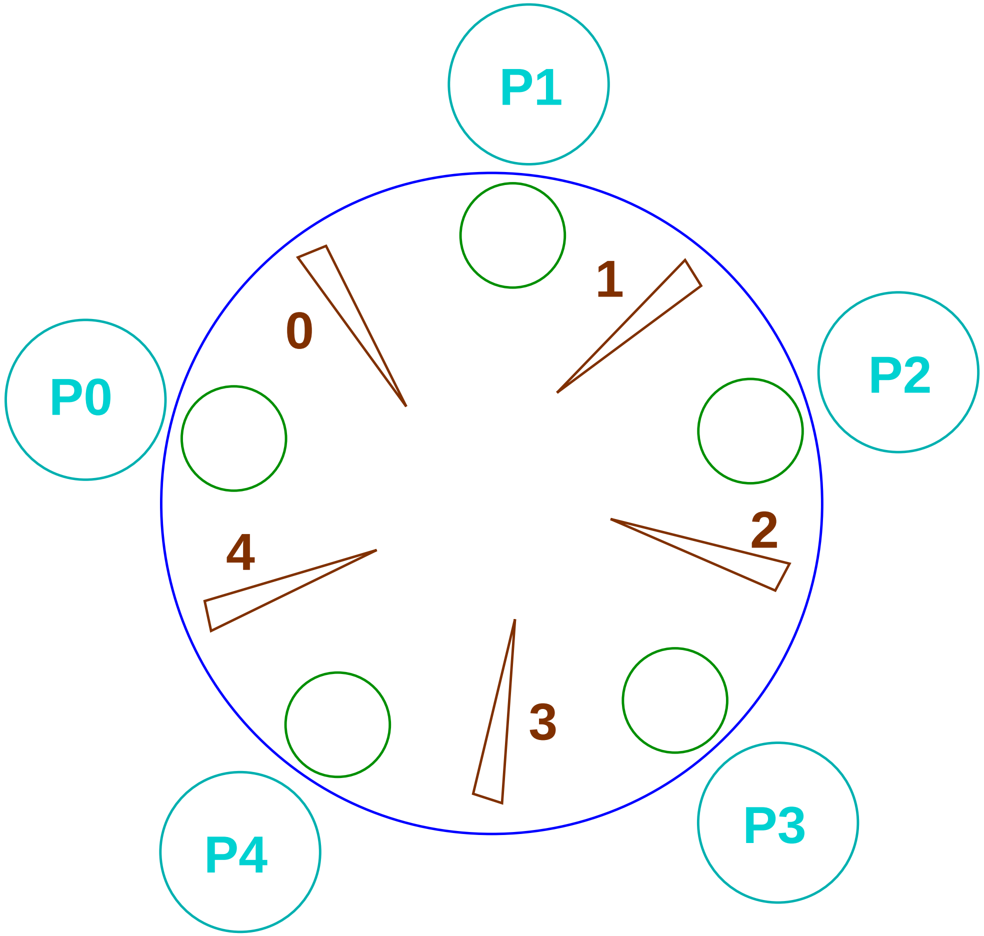
Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right chopsticks. Each i chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both chopsticks so that the chopsticks become available to others. A philosopher can take the chopstick on their right or the one on their left as they become available, but cannot start eating before getting both chopsticks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think." - Wikipedia

Note: problem may not have much practical utility!

Dining Philosophers



Dining Philosophers Problem: Requirements

- No deadlock or livelock.
- No starvation.
- Maximum parallelism.

Dining Philosophers Problem

The obvious solution is not deadlock free. Each philosopher repeats following four steps.

1. Philosopher P_i continues thinking until her left-chopstick (i) is available, and she picks it up.
2. Again continues thinking until her right-chopstick $((i + 4) \mod 5)$ is available, she picks it up too.

Dining Philosophers Problem

3. Then she eats for a finite amount of time.
4. Drops both the chopsticks.

Note: a binary semaphore can be associated with every chopstick.
Deadlock: all philosophers pick up five chopsticks from their left.

New Rules

We change the rule of the game slightly.

- After picking up the left-chopstick a philosopher waits for t_1 minutes to get the right-chopstick. If he cannot get it, puts back the left-chopstick.
- Waits for t_2 minutes before starting the next round.

This avoids deadlock but may lead to livelock[a]

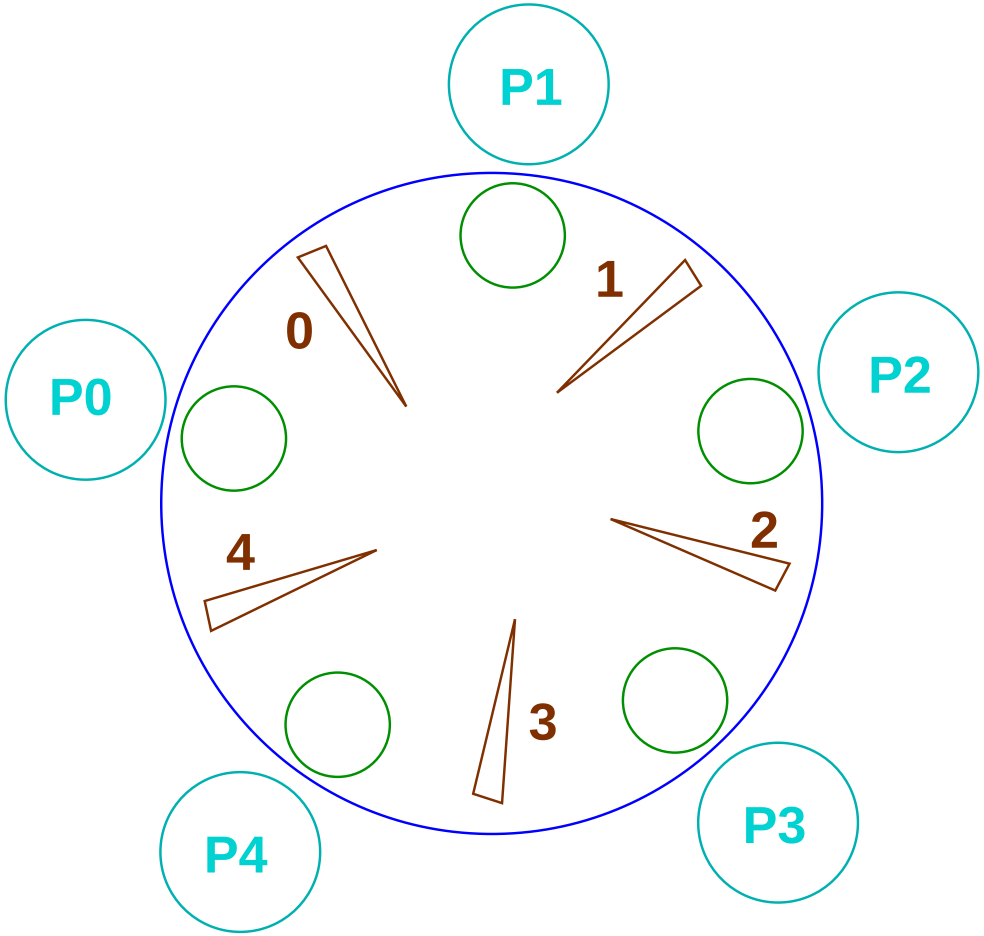
---

[a] All five of them are picking up and putting down chopsticks again and again.

First Solution

- The first solution to the problem that can make it deadlock-free was proposed by Dijkstra.
- The chopsticks (resources) are numbered from $0, 1, 2, 3, 4$.
- Restriction: a philosopher will always pick up the lower-numbered chopstick first, and then the higher-numbered chopstick.

Chopsticks are Numbered



No Deadlock

- If we now allocate chopsticks to philosophers in sequence starting from P_0 according to the protocol, we get the following assignments. $P_0 \leftarrow 0$, $P_1 \leftarrow X$, $P_2 \leftarrow 1$, $P_3 \leftarrow 2$, $P_4 \leftarrow 3$.
- Now P_0 can finish eating by picking up chopstick 4.

Solution using Arbitrator

- There is a central arbitrator (waiter, server) who keeps track of states of all philosophers (eating, thinking, hungry to eat).
- A philosopher when hungry will request the arbitrator.
- The arbitrator checks the states of two adjacent philosophers, if on of them is eating, the requesting philosopher will wait.

Solution using Arbitrator

- Otherwise the philosopher can pick-up both chopsticks and starts to eat.
- Once finished, he drops the chopsticks and starts thinking.

Bibliography

1. Operating System Concepts by Abraham Silberschatz, Peter B Galvin & Gerg Gagne, 9th ed., Wiley Pub., 2014, ISBN 978-81-265-5427-0.
2. Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau & Andre C. Arpaci-Dusseau Pub. Arpaci-Dusseau Books, LLC, 2008-19.
3. Beginning Linux Programming by Neil Mathew & Richard Stones, 3rd ed., Wiley Pub., 2004, ISBN 81-265-0484-6.
4. Understanding the Linux Kernel by Daniel P Bovet & Marco Cesati, 3rd ed., O'Reilly, ISBN 81-8404-083-0.
5. `https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem`

6. https://cseweb.ucsd.edu/classes/fa05/cse120/lectures/120-l6.pdf
7. https://people.csail.mit.edu/rinard/osnotes/h14.html
8. https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
9. https://en.wikipedia.org/wiki/Dining_philosophers_problem