The picture can't be displayed.

Chapter 6: Process Synchronization

Cooperating Processes

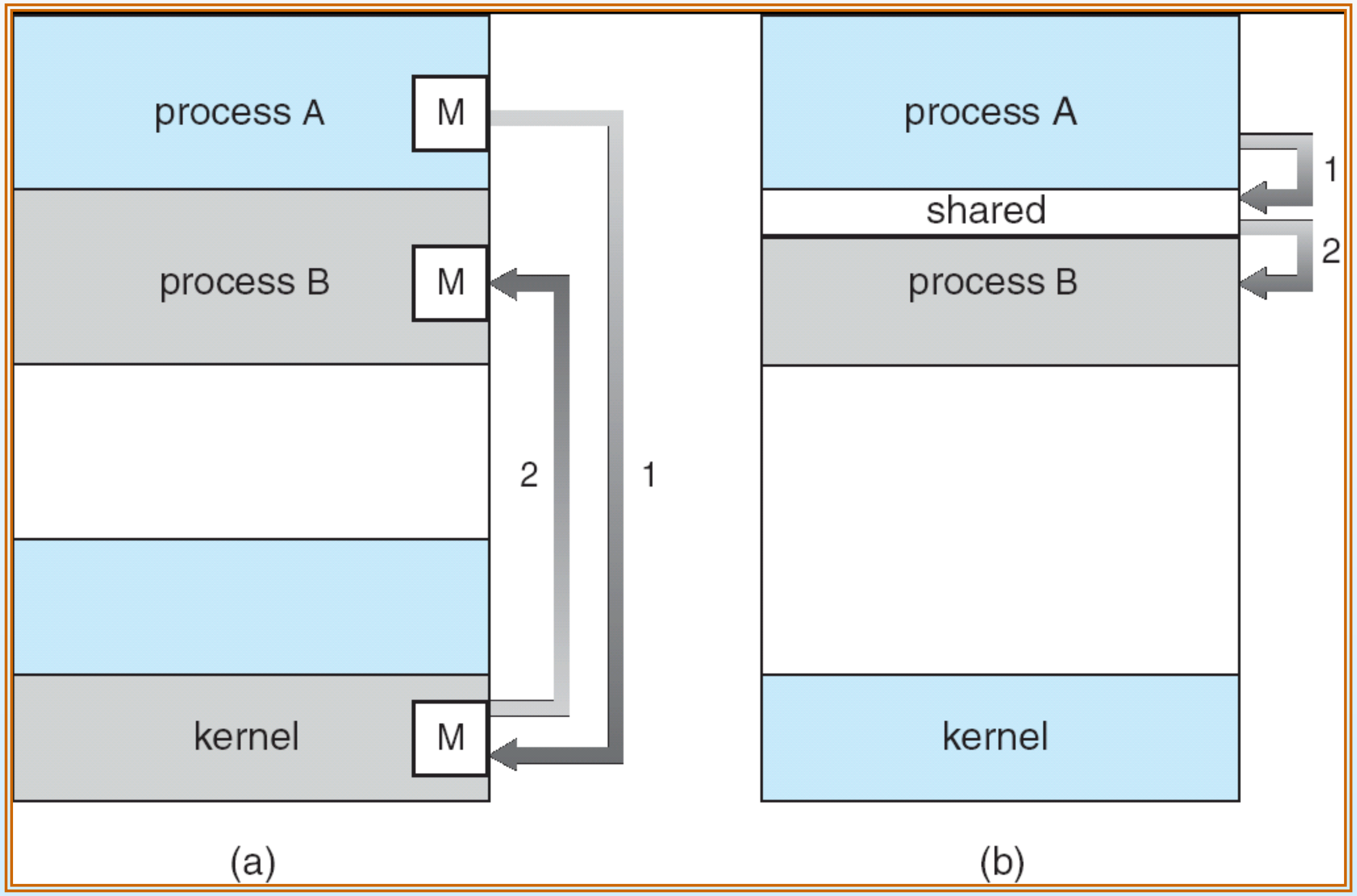
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing (shared file)
  - Computation speed-up(parallel execution of subtasks)
  - Modularity (constructing the system in a modular fashion)
  - Convenience (each user may work on many tasks at the same time; printing, editing, compiling in parallel)

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions is called IPC
- Two fundamental models – shared memory and message passing
- Shared memory: The memory region to be shared is established. Process exchange information by reading and writing to the shared region.
- Message passing: Communication takes place by exchanging messages between processes.
- Shared Memory vs Message passing:
  -Shared memory is faster since message passing uses system calls which requires kernel invention making it slower.

-For intercomputer communication, message passing is preferred

Communications Models



Shared Memory

- Processes establish a region of shared memory.
- They can exchange information by reading and writing into the shared memory region.
- The form of the data and the location are determined by the processes and are not under the OS' control
- Processes are responsible for ensuring that they are not writing to the same location simultaneously.

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Synchronisation

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Example

- Code:
  NewBalance= Balance + Amount;

  Balance= NewBalance;

Saw initially Balance is 1000 and both Jack and Jill are depositing 200 each that should have resulted I 1400

Jack first adds so line 1 executed

NewBalance = 1000+200

Then Jill adds so line 1 executed

NewBalance= 1000 +200 (Note balance had not got updated

Then Jack executes line 2

Balance = 1200

Then Jill executes line 2

Balance=1200

Producer Consumer

```
while (1)
  {
 /* produce an item and put in
    nextProduced


    while (count ==
    BUFFER_SIZE)
     ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    }
```

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

/* consume the item in nextConsumed
  }
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

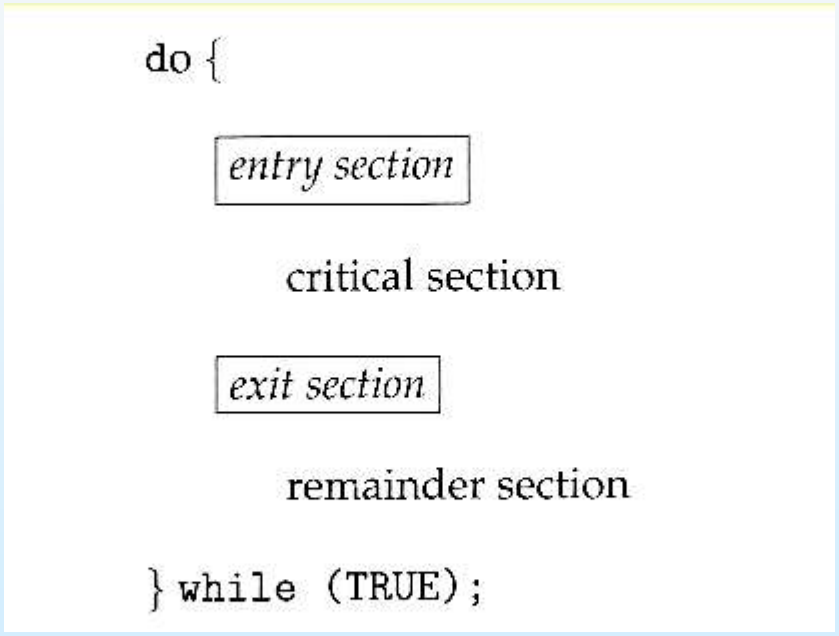
- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

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

Critical Section Problem



```
do {
        $\boxed{\textit{entry section}}$

            critical section

        $\boxed{\textit{exit section}}$

            remainder section

} while (TRUE);
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Hypothetical case -1

- Two processes P1 and p2.

Initially say turn =1.

-

  Pi

  While turn !=i ; //waits for his turn

- Critical Section
- Turn=j

  Will this work?

Hypothetical case -1

- Two processes Pi and pj. Say intially turn=j

Pi

```
  While turn !=i ;
Critical Section
Turn=j
```

Note: While mutual exclusion is guaranteed, progress is not. Say Pi wants to run and Pj does not. Then Pi waits on a process who is not ready to enter the Critical section. Another case: Pj runs first and then Pi runs. Now Pi wants to run again but turn =j so Pi has to wait till Pj runs again.

Hypothetical case -2

- Two processes : flag [k] for pk intialised to 0 for both processes

```
  Pi
  While flag[j] do;[ looping ]
Flag[i]=true
Critical Section
Flag[i]=false
```

Will this work?

Hypothetical case -2

- Two processes : for both process flag [k] for pk intialised to 0

  Pi
  1.While flag[j] do;[looping]

  2. Flag[i]=true

  3. Critical Section

  4. Flag[i]=false

  Both processes can reach critical if sequence of execution is pi.1, pj.1, pi.2, pi.3, pj.2, pj.3,…
  Mutual exclusion is not guaranteed

Hypothetical case -3

- Two processes : flag [k] intialised to 0 for both

  Pi

  1. Flag[i]=true

  2. While flag[j] do; // looping

  3. Critical Section

  4. Flag[i]=false

Hypothetical case -3

- Two processes : flag [k] intialised to 0

  Pi
  1. Flag[i]=true
  2. While flag[j] do;[looping]
  3. Critical Section
  4. Flag[i]=false

  Both processes keep waiting if Pi.1, Pj.1, Pi.2, Pj.2 …. Bounded waiting not guaranteed

Peterson's Solution

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

Algorithm for Process P_i

```
do {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

              CRITICAL SECTION

        flag[i] = FALSE;

                REMAINDER SECTION

    } while (TRUE);
```

WHAT happens if the below?

```
do {
        flag[i] = TRUE;
        turn = i;
        while ( flag[j] && turn == j);

              CRITICAL SECTION

        flag[i] = FALSE;

                REMAINDER SECTION

    } while (TRUE);
```

WHAT happens if the below?

```
do {
        flag[i] = TRUE;
        turn = i;
        while ( flag[j] && turn == j);

              CRITICAL SECTION

        flag[i] = FALSE;

                REMAINDER SECTION

    } while (TRUE);
```

Flag[i]=0 and flag[j]=0
Turn=0 initially

p1.1, p1.2, p1.3
Then p2.1, p2.2, and p2
also enters critical section

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems since message being passed to all processors will be time consuming.
  - Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

  - Either test memory word and set value

  - Or swap contents of two memory words

Structure

- {
- acquire lock
- critical section
- release lock
- remaining code
- }

TestAndndSet Instruction

- Definition:

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

Solution using TestAndSet

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

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

        //    critical section

   lock = FALSE;

        //      remainder section

 } while ( TRUE);
```

No bounded waiting.
Some process might starve

Swap Instruction

- Definition:

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

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {
    key = TRUE;
     while ( key == TRUE)
        Swap (&lock, &key );

          //    critical section

     lock = FALSE;

        //      remainder section

   } while ( TRUE);
```

Question

- Can we just use a queue where a process who will need to enter CS at some point in time; goes into a queue for that resource. The process can start running when the process gets his turn after the resource is available.

  Issues: No concurrent running can happen even of the portions that could run concurrently. Instead the process can go into a queue at the point it is next needs the CS. There has to be some code that stops regulates this queue where it puts something into the queue if resource busy and removes an element from the queue when the resource is available. So you now have to write CS code that does if(!Test&Set)push_queue(Pid); //wait on something...

  The issue without a lock to enter CS will result in same issues as we saw earlier in the producer consumer problem with the buffer queue.

Bounded waiting mutual exclusion for multiple processes

- Initially lock is false

```
do
{ waiting[i] = true;
  key = true;
  while (waiting[i] && key)
        key = test_and_set(&lock);
  waiting[i] = false;
/* critical section */

j = (i + 1) % n;
while ((j != i) && !waiting[j])
        j = (j + 1) % n;
if (j == i) lock = false;
else waiting[j] = false;
/* remainder section */
} while (true);
```

Why not lock=false? $\searrow$

Bounded waiting mutual exclusion for multiple processes

- Initially lock is false

```
do
{ waiting[i] = true;
  key = true;
  while (waiting[i] && key)
        key = test_and_set(&lock);
  waiting[i] = false;
/* critical section */

j = (i + 1) % n;
while ((j != i) && !waiting[j])
        j = (j + 1) % n;
if (j == i) lock = false;
else waiting[j] = false;
/* remainder section */
} while (true);
```

Why not lock=false?

Semaphore

- Synchronization tool that does not require busy waiting if implemented along with an associated queue where process is queue is woken up when resoure is available.
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

  -
    ```
    wait (S) {
         while S <= 0
              ; // no-op
           S--;
     }
    ```

  -
    ```
    signal (S) {
       S++;
    }
    ```

Advantage of Semaphore

- It's simple and always have a non-negative Integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores. Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain. Example: indicate number of commodities in market.
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

Mutual Exclusion

```
Semaphore S=1;
```

```
{  wait (S);
   Critical Section
   signal (S);
}
```

```
wait (S) {
     while S <= 0
          ; // no-op
     S--;
  }


signal (S) {
       S++;
     }
```

Producer Consumer naïve

```
PRODUCER
    send(char c)
    {
      buf[in]=c
          in=(in+1)%N
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            c=buf[out];
            out=(out+1)%N;
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
```

Need to add below constraints

-Consumer consumes only if atleast one element in buffer

-Producer produces if buffer not full

consume when buffer not empty

consume when buffer not empty

```
PRODUCER
    send(char c)
    {
      buf[in]=c
          in=(in+1)%N
          signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {      char c;
           wait(chars);
           c=buf[out];

           out=(out+1)%N
    ;
           return c;
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0;
```

Produce when buffer not full

Produce when buffer not full

```
PRODUCER
    send(char c)
    {
      wait(space);
      buf[in]=c
      in=(in+1)%N
      signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];

            out=(out+1)%N
    ;
            signal(space);
            return c;
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N;
```

consumer produces a space!

Multiple Producers and one consumer

Multiple Producers and one consumer

Will two producers find single space left in line 1 and both write into the single space left?

```
PRODUCER
    send(char c)
    {
      wait(space);
     wait(lock);
      buf[in]=c
      in=(in+1)%N
      signal(lock);
      signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];

            out=(out+1)%N
    ;
            signal(space);
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple Producers and one consumer

Will two producers
find single space
left in line 1
and both write
into the single
space left?
No. First producer
Will decrement space
To 0 so second will
wait

```
PRODUCER
    send(char c)
    {
      wait(space);
     wait(lock);
      buf[in]=c
      in=(in+1)%N
      signal(lock);
      signal(chars)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            c=buf[out];
            out=(out+1)%N
    ;
            signal(space);
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple producers and consumers

```
PRODUCER
     send(char c)
     {
        wait(space);
      wait(lock);
        buf[in]=c
        in=(in+1)%N
        signal(lock);
        signal(chars)
    }
```

```
CONSUMER
     char rcv()
     {      char c;
            wait(chars);
         wait(lock)
            c=buf[out];

            out=(out+1)%N
     ;
         signal(lock)
            signal(space);
            return c;
     }
```

WORKS ?

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple producers and consumers

```
PRODUCER
     send(char c)
     {
        wait(space);
      wait(lock);
        buf[in]=c
        in=(in+1)%N
        signal(lock);
        signal(chars)
    }
```

```
CONSUMER
     char rcv()
     {       char c;
             wait(chars);
          wait(lock)
             c=buf[out];

             out=(out+1)%N
     ;
          signal(lock)
             signal(space);
                 rn c;
```

ISSUE ??
Does not permit Producer and consumer to simultaneously function

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple producers and consumers

```
PRODUCER
    send(char c)
    {
      wait(space);
     wait(lock);
      buf[in]=c
      in=(in+1)%N
      signal(lock);
      signal(chars)
   }
```

```
CONSUMER
    char rcv()
    {      char c;
           wait(chars);
         wait(lock)
           c=buf[out];

           out=(out+1)%N
    ;
         signal(lock)
           signal(space);
             rn c;
```

If simultaneously Function not required then would this work?

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

Multiple producers and consumers

```
PRODUCER
     send(char c)
     {
        wait(sem space);
      wait(lock);
        buf[in]=c
        in=(in+1)%N
        signal(lock);
        signal(chars)
    }
```

```
CONSUMER
     char rcv()
     {       char c;
             wait(chars);
          wait(lock)
             c=buf[out];

             out=(out+1)%N
     ;
          signal(lock)
             signal(space);
                rn c;
```

If simultaneously Function not required then would this work? yes

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

If simultaneously
Function not
required then would
this work?

```
PRODUCER
    send(char c)
    {
      wait(lock);
     wait(space);
      buf[in]=c
      in=(in+1)%N
      signal(chars);
      signal(lock)
    }
```

```
CONSUMER
    char rcv()
    {      char c;
           wait(chars);
         wait(lock)
           c=buf[out];

           out=(out+1)%N
    ;
         signal(lock)
           signal(space);
           return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1, ;
```

If simultaneously
Function not
required then would
this work? No

```
PRODUCER
    send(char c)
    {
      wait(lock);
     wait(space);
      buf[in]=c
      in=(in+1)%N
      signal(chars);
      signal(lock)
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
         wait(lock)
            c=buf[out];

            out=(out+1)%N
    ;
         signal(lock)
```

Producer got lock (wait lock) so proceed to step 2
Consumer finds he can consume (wait chars) so proceed to step 2
Producer waiting since buffer full (wait space) so waiting at step 2
Consumer wants to proceed to consume but cant get lock(wait lock).
So waiting at step 2.
➔ Both producer and consumer are waiting.

Multiple Producers and consumers

Modify the code to now work for simultaneous working of producer and consumer

```
PRODUCER
    send(char c)
    {
      wait(space);
      wait(lock);
      buf[in]=c

      in=(in+1)%N

      signal(lock)
      signal(chars);

    }
```

```
CONSUMER

    char rcv()

    {       char c;

            wait(chars);
            wait(lock);

            c=buf[out];

            out=(out+1)%N;

            signal(lock);
            signal(space);
            return c;

    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1;
```

Multiple Producers and consumers

```
PRODUCER
    send(char c)
    {
      wait(space);
      wait(lock_p);
      buf[in]=c
      in=(in+1)%N
      signal(lock_p)
      signal(chars);
    }
```

```
CONSUMER
    char rcv()
    {       char c;
            wait(chars);
            wait(lock_c);
            c=buf[out];
            out=(out+1)%N;
            signal(lock_c);
            signal(space);
            return c;
    }
```

```
/*the buffer*/
Char buf[N];
In =0; out =0;
Semaphore chars=0, space =N, lock=1;
```

Permits a consumer and producer
to simultaneously function!

Counting Semaphores

Can Semaphores be used to allow processes to use multipe copies of an available resource. Eg: 5 printers available to a set of processes.

- Say you have a limited number of resources..eg:5 printers. How would you design a users code using semaphores.
- If all 5 in use then user cant print. If at least one available then user can use it.

- Say you have a limited number of resources..eg:5 printers. How do design a users code using semaphores.

```
Semaphore P=5;

user(..)
{wait(P);
  ......
 Signal(P);
}
```

Semaphore Implementation

The big problem with semaphores as described till now is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***

An alternative approach:

-Block a process if waiting and swap it out of the CPU.

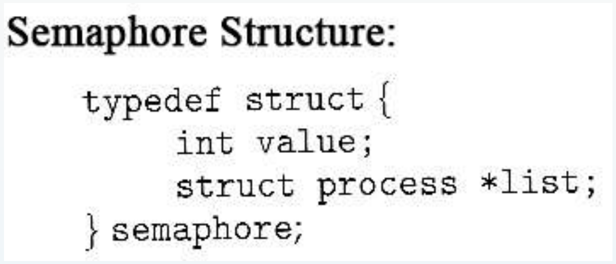
-Each semaphore needs to maintain a list of processes that are blocked waiting for it

-One of the processes is woken up and swapped back when the semaphore becomes available.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations :
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)



Semaphore Structure:

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

- Implementation of wait:

Value=1//available
Value<=0//in use so wait in queue
The negative value indicates the number of processer waiting(exclude the one running)

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

- Implementation of signal:

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

Semaphore Implementation with no Busy waiting (Cont.)

Understanding Wait:

Say you have a printer
Value=1//available
First person to use printer will
Call wait(S) and value now becomes
0. However due to condition
If(value<0) block; the first user is not
blocked and uses the printer.

Next user at the end of first line has
value=-1 and is blocked.

Hence value =-x indicates that x ppl
are waiting.
Vlue>=0 means currently no one is
waiting

Implementation of wait:

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

Implementation of signal:

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

Semaphore Implementation with no Busy waiting (Cont.)

Understanding signal

value =-x indicates that x ppl are waiting.
Value>=0 means currently no one is waiting

Signal does a value++ in step 1.
Hence in step 2:
Value –x means x+1 ppl are waiting
Value=0 means 1 person is waiting
Value>0 means noone is waiting.

There is no action required if noone is waiting
Thus at value<=0 the first process is woken up

}

Implementation of wait:

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

Implementation of signal:

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

Semaphore Implementation with no Busy waiting (Cont.)

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

- After a process is done, a resource is available. On doing value++, either
- value>0 OR value <=0. Note if value=-x, it means x+1 processes' are waiting
- If value >0, it implies that there are no processes waiting so nothing to do.
- If value<=0 then there is at least one process waiting. Take one off the wait queue and add it to the ready queue.

Semaphore Implementation w vs w/o Busy waiting

Without busy waiting

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

With busy waiting

Classical Problems of Synchronization

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

Readers-Writers Problem

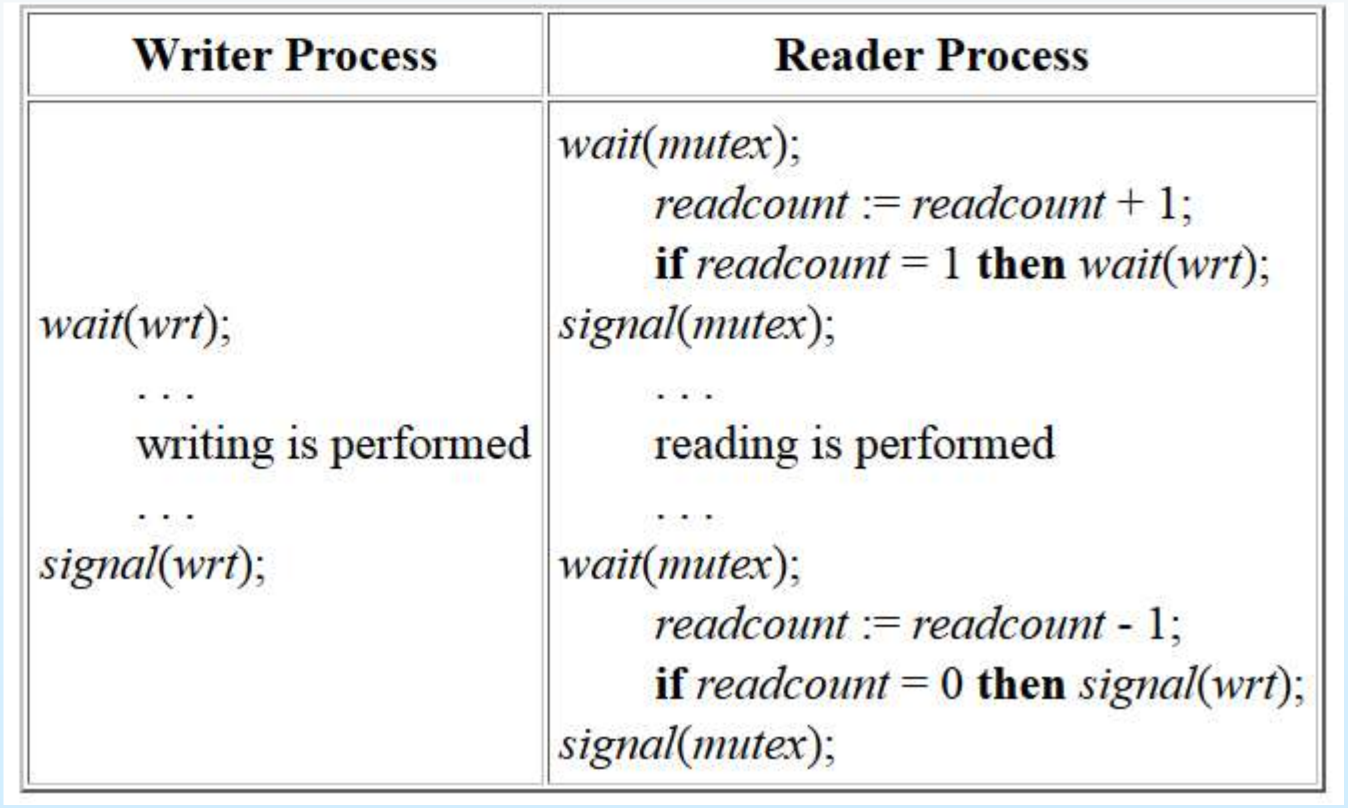
- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

http://www.cis.upenn.edu/~lee/03cse380/lectures/ln2-ipc-v2.pdf

Constraints

- Only one writer writes at a time
- If even one reader is reading then, no writing should happen
- Multiple readers can read at the same time

Readers-Writers Problem (Cont.)



Writer Process	Reader Process
	wait(*mutex*);
	readcount := *readcount* + 1;
	if *readcount* = 1 **then** *wait*(*wrt*);
wait(*wrt*);	*signal*(*mutex*);
. . .	. . .
writing is performed	reading is performed
. . .	. . .
signal(*wrt*);	*wait*(*mutex*);
	readcount := *readcount* - 1;
	if *readcount* = 0 **then** *signal*(*wrt*);
	signal(*mutex*);

Second Readers-Writers Problem

- First-readers-writers-problem : Gives preference to readers –no readers will be kept waiting unless writer has got access
- Second-readers-writers-problem: Requires writers to be given priority where writers should not be kept waiting. That is, no new reader can gain access if a writer is waiting

Second readers writers

- Make sure multiple readers can read
  -You hence cant lock the reading part as it restricts another reader reading simultaneously
- Make sure more readers don’t join the reading group if a writer expressed interest in writing
  -When writer interested lock readers from entering
- When a writer writes, no one reads/writes.
  -lock while writing
- If both reader and writer is waiting, give preference to writer
  A)writer should not release lock to reader if another writer is waiting.
  B)If first writer is writing, restrict the number of readers who can even show interest so that if a writer enters after first writer relinquishes lock then he does not have to wait for a queue of waiting readers to finish.

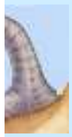
Second Readers-Writers Problem

Writer process

```
Repeat
      wait(mutex2);
      writers ← writers + 1;
      if writers = 1 then
            wait(read);
      end-if
      signal(mutex2);
      wait(write);

      // write the resource

      signal(write);
      wait(mutex2);
      writers ← writers - 1;
      if (writers = 0) then
            signal(read);
      end-if
      signal(mutex2);
until done
```

Reader process

```
repeat
      wait(mutex3);
      wait(read);
      wait(mutex1);
      readers ← readers + 1;
      if readers = 1 then
            wait(write);
      end-if
      signal(mutex1);
      signal(read);
      signal(mutex3);

      // read the resource

      wait(mutex1);
      readers ← readers - 1;
      if (readers = 0) then
            signal(write);
      end-if
      signal(mutex1);
until done
```



Mutex 1 and 2 makes sure that reader(writer) increment/decrement is not concurrently attempted by another reader(writer)

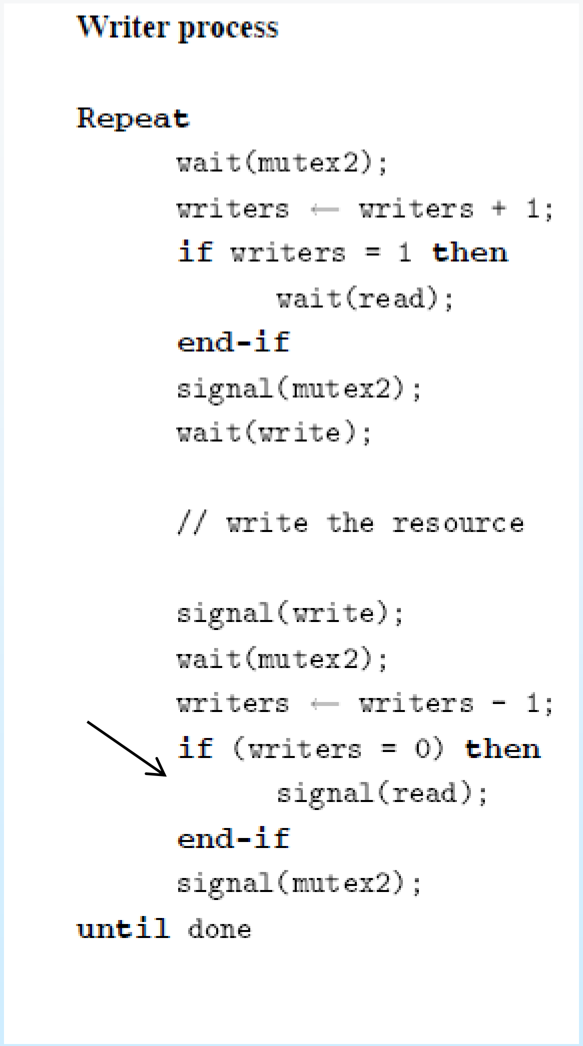
Semaphore read: makes sure that reading happens only when writer is not interested in writing. When writer interested then further readers are stopped by wait(read)

Semaphore write: makes sure only one writer writes and no writer writes when someone is still reading

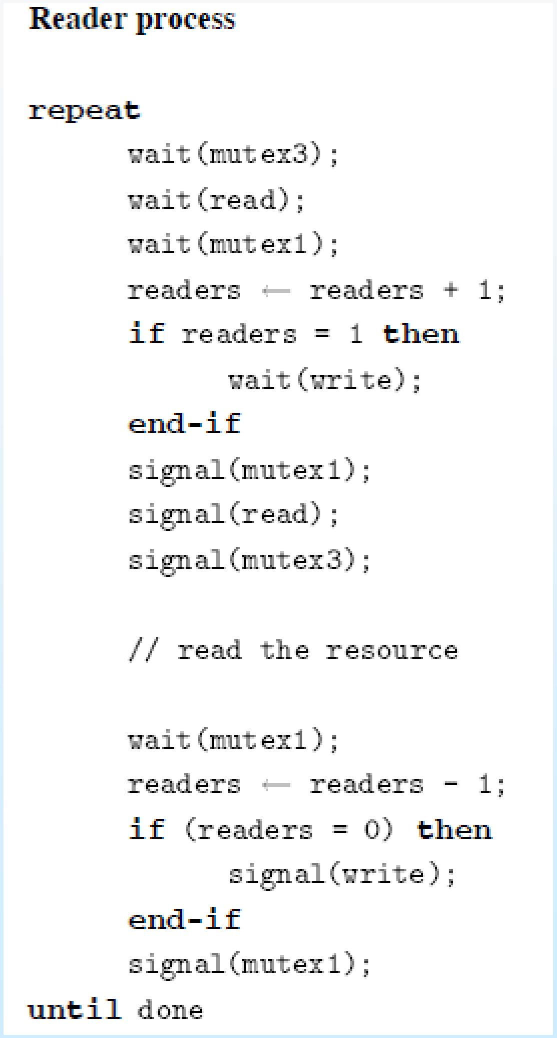
Mutex 3: makes sure that only one reader accummulates when a writer is writing. Other readers even if interested can not join the read queue though they can join mutex3 queue.

Second reader writers problem says that a writer should not be kept waiting. Say a writer W1 is writing and readers R1,R2,R3,R4 arrive. If writer W2 arrives when W1 is still writing, then all readers wait until W2 is done(semaphore read). If W2 arrives after W1 is over, then W2 waits only for single reader R1 to complete as only R1 joins the read queue(mutex3).

Preference to writer



```
Writer process

Repeat
        wait(mutex2);
        writers ← writers + 1;
        if writers = 1 then
                wait(read);
        end-if
        signal(mutex2);
        wait(write);

        // write the resource

        signal(write);
        wait(mutex2);
        writers ← writers - 1;
        if (writers = 0) then
                signal(read);
        end-if
        signal(mutex2);
until done
```



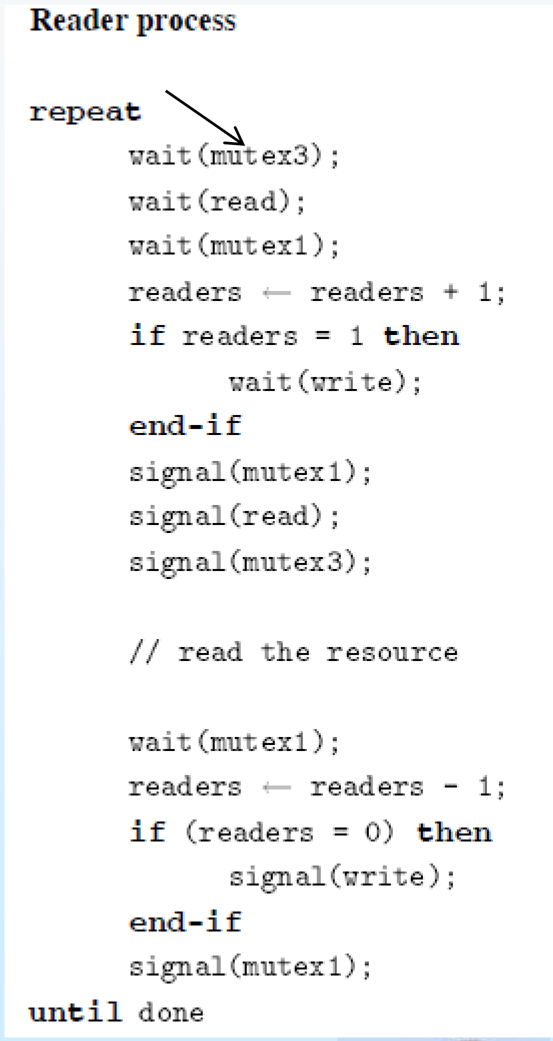
```
Reader process

repeat
        wait(mutex3);
        wait(read);
        wait(mutex1);
        readers ← readers + 1;
        if readers = 1 then
                wait(write);
        end-if
        signal(mutex1);
        signal(read);
        signal(mutex3);

        // read the resource

        wait(mutex1);
        readers ← readers - 1;
        if (readers = 0) then
                signal(write);
        end-if
        signal(mutex1);
until done
```

If more writers join when one writer is writing then reader cant progress as signal(read) wont happen unless writer=0. Meanwhile new writers will reach uptil wait(write) even if reader arrived earlier than the new writers

Preference to writer(2)

Mutex3 limits number of
Waiting readers. Many readers
could otherwise accummulate
making ‘semaphore read’ a
Large negative. If a new writer
joins then he gets stuck in the
Queue until all readers exit
‘Sempahore read’s queue.
However, now atmost one reader
Can accummulate while a writer is
Writing.

Writer starvation could happen in
Case of the busy waiting implementation
Of semaphore.
A reader might always grab the
lock when both a writer and many
Readers are waiting



Reader process

```
repeat
        wait(mutex3);
        wait(read);
        wait(mutex1);
        readers ⟵ readers + 1;
        if readers = 1 then
                wait(write);
        end-if
        signal(mutex1);
        signal(read);
        signal(mutex3);

        // read the resource

        wait(mutex1);
        readers ⟵ readers - 1;
        if (readers = 0) then
                signal(write);
        end-if
        signal(mutex1);
until done
```

test

- Ok to remove mutex1? Since it is already inside mutex3. Hence two processes wont be contesting for mutex1 anyways.
- Can i remove wait(read)? An interesting writer waits at wait(writer)
- For the third readers writers problem: Consider that writers code is: “wait(read), wait(write), write, signal(write), signal(read)”.
  and readers code is same as 2nd readers writers problem code except that we remove wait and signal mutex3.
  Will this work ? If yes are there any inefficiencies. If not, why not?

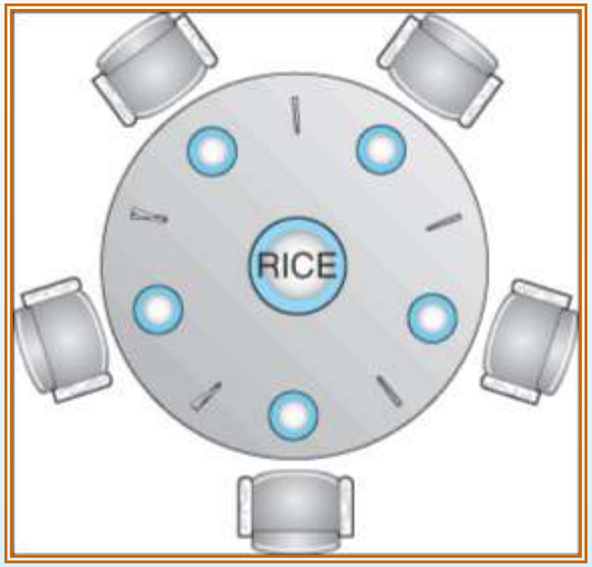
homework

- How would you solve the third readers writers problem.
  -Multiple readers can read and only either of read or write can happen. However, order of arrival is given preference.

homework

- How would you solve the third readers writers problem. -Multiple readers can read and only either of read or write can happen. However, order of arrival is given preference.
- https://rfc1149.net/blog/2011/01/07/the-third-readers-writers-problem/
- http://cs.nyu.edu/~lerner/spring10/MCP-S10-Read04-ReadersWriters.pdf
- If writers is "wait(read), wait(write), write, signal(write), signal(read)"
  and readers is same except remove wait and signal mutex3
  Will this work or not work?

Dining-Philosophers Problem



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

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

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

            //  eat

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

           //  think

} while (true) ;
```

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

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

            //  eat

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

           //  think

} while (true) ;
```

Deadlock!- no progress
 Each picks
Left fork at same time
And all wait forever

Conductor solution

- do
```
{wait(mutex)
  wait(i)
  wait((i+1)%5)
 signal(mutex)
//eat
 signal(i)
 signal((i+1)%5)
 think
}while(true)
```

Conductor solution

- do

```
{wait(mutex)
  wait(i)
  wait((i+1)%5)
 signal(mutex)
//eat
 signal(i)
 signal((i+1)%5)
 think
}while(true)
```

Lets say P1 got both his forks
and is eating.
P2 reaches step 3 after getting
one fork and is waiting for the other
which is being used by P1.
➔P2 is still inside the mutex lock
which does not permit P4 to request forks
though both his forks are idle.

Concurrent execution not
Possible due to mutex lock

Move with precaution

Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

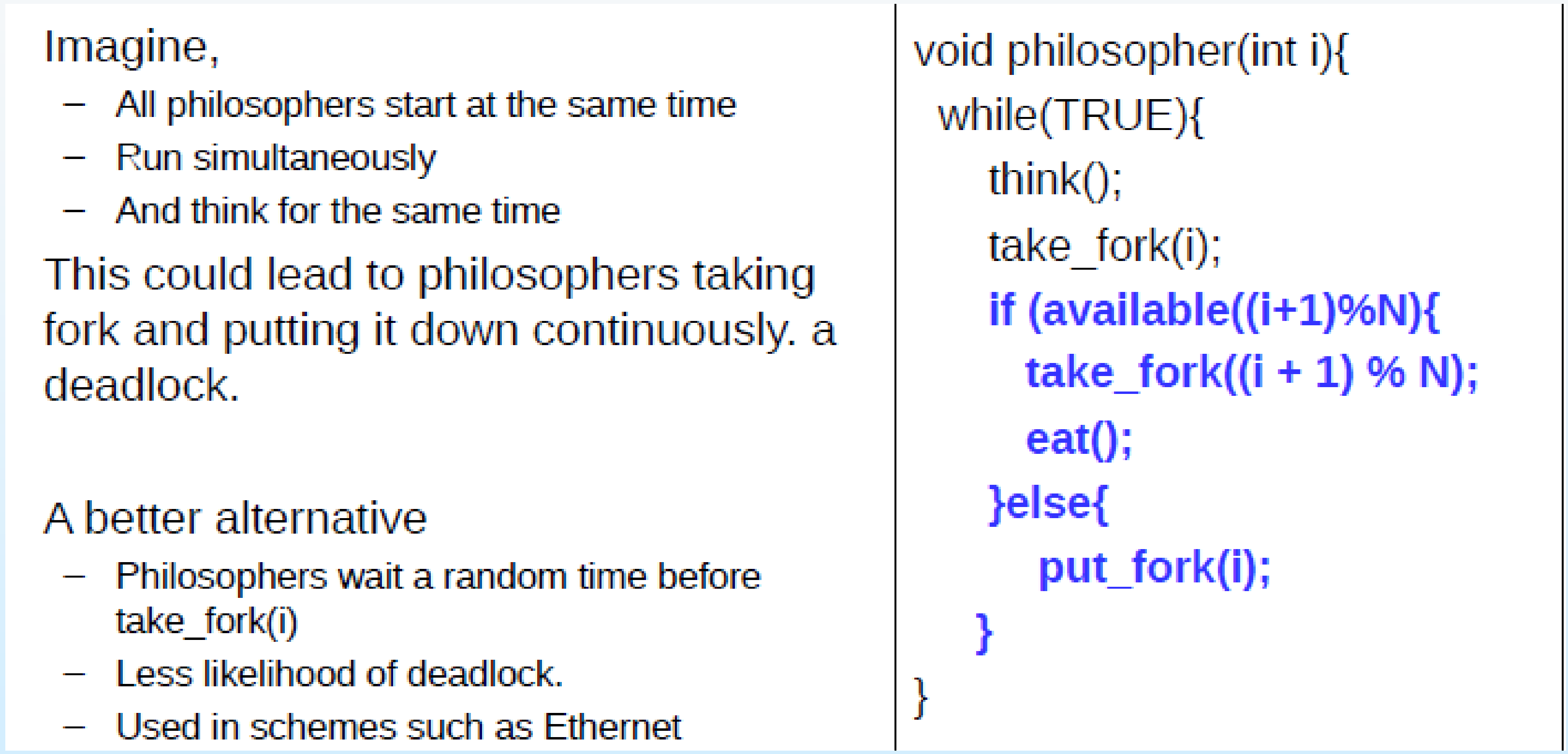
This could lead to philosophers taking fork and putting it down continuously. a deadlock.

A better alternative

- Philosophers wait a random time before take_fork(i)
- Less likelihood of deadlock.
- Used in schemes such as Ethernet

```
void philosopher(int i){
  while(TRUE){
      think();
      take_fork(i);
      if (available((i+1)%N){
        take_fork((i + 1) % N);
        eat();
      }else{
         put_fork(i);
     }
}
```

Move with precaution



Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

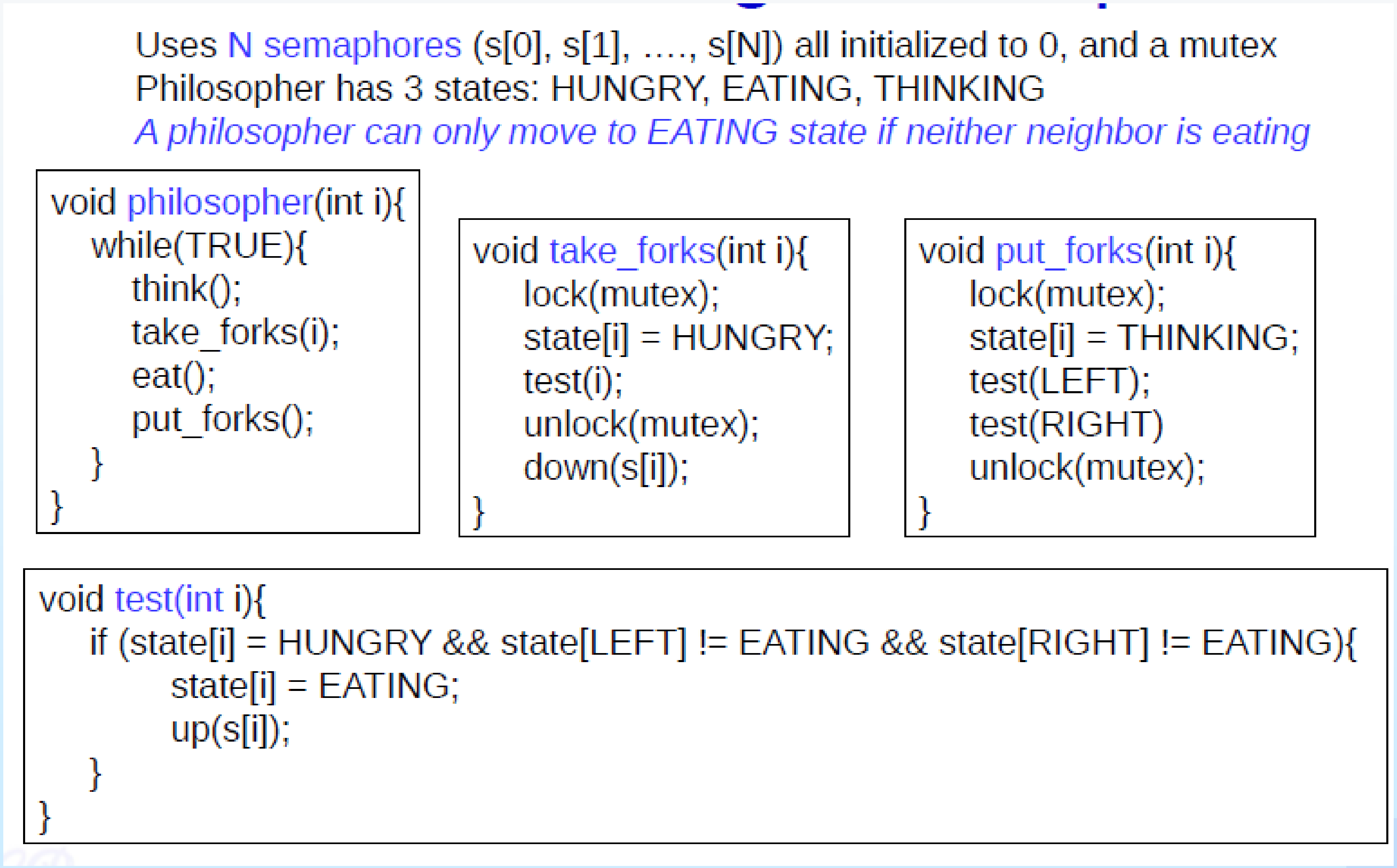
A better alternative

- Philosophers wait a random time before take_fork(i)
- Less likelihood of deadlock.
- Used in schemes such as Ethernet

```
void philosopher(int i){
  while(TRUE){
      think();
      take_fork(i);
      if (available((i+1)%N){
         take_fork((i + 1) % N);
         eat();
      }else{
          put_fork(i);
      }
}
```

All pick fork together to realise other fork is not available. So all put back fork together. This goees on.

Tannenbaum solution



Uses N semaphores (s[0], s[1], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
     lock(mutex);
     state[i] = HUNGRY;
     test(i);
     unlock(mutex);
     down(s[i]);
}
```

```
void put_forks(int i){
     lock(mutex);
     state[i] = THINKING;
     test(LEFT);
     test(RIGHT)
     unlock(mutex);
}
```

```
void test(int i){
     if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
          state[i] = EATING;
          up(s[i]);
     }
}
```

- Done from Robeiro IIT chennais slides (Synchronisation.pdf)
- Note on tannebaum solution
- Up(fork) is like signal(fork)
- Down(fork) is like wait(fork)
  wait(fork){ fork --; while (fork <0);}

At take_forks, the i philosopher might be waiting on s[i] if test[i] came out without incrementing s[i]. S[i] will be incremented when i's neighbor finishes eating and checks if his neighbor i have been waiting.
He then calls test i. At this point test [i] sets S[i] at 1. So the earlier instance that was waiting now succeeds and i enters eating phase.

So philosopher i eats either if his neighbors are not eating or if his neighbor who was eating finished eating and calls test i.

Tannenbaum solution

- A philosopher can eat under two conditions

1. He finds his neighbors not eating so he sets S[i]=1 and this allows him to not block at down(s[i])(same as wait(S[i]). So he goes on to eat
2. He finds a neighbor eating so exits test[i] with S[i]=0 and thus is kept waiting at wait[S[i]]. For process i to pass the wait statement the value of S[i] should get set as 1. This is done by one of its eating neighbors. Once the neighbor(say i+1) finishes eating, he does a test[i]. If both of i's neighbors are not eating then S[i] is set to true in the neighbor i+1's code.
   Hence in process i's code, the process now moves past the wait step and goes on to eat.

Tannenbaum solution-starvation

- Imagine that we are trying to starve Philosopher 0. Initially, 2 and 4 are at the table and 1 and 3 are hungry. Imagine that 2 gets up and 1 sits down; then 4 gets up and 3 sits down. Now we are in the mirror image of the starting position.
- If 3 gets up and 4 sits down, and then 1 gets up and 2 sits down, we are back where we started. We could repeat the cycle indefinitely and Philosopher 0 would starve.
- So, Tanenbaum's solution doesn't satisfy all the requirements.

Chandy Misra solution

- For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (*n* for agent P_n). Each fork can either be *dirty* or *clean.* Initially, all forks are dirty.
- When a philosopher wants to use a set of resources (*i.e.* eat), he must obtain the forks from his contending neighbors. For all such forks he does not have, he sends a request message.
- When a philosopher with a fork receives a request message, he keeps the fork if it is clean, but gives it up when it is dirty. If he sends the fork over, he cleans the fork before doing so.
- After a philosopher is done eating, all his forks become dirty. If another philosopher had previously requested one of the forks, he cleans the fork and sends it.
- https://ycpcs.github.io/cs420-fall2017/lectures/lecture12_classic_synchronization_problems.pdf

Rules

- Initially forks dirty
- After eating forks dirty
- If someone had requested for your fork when you were eating, then clean and give it after you finish eating
- If you have both forks start eating
- I need to give a fork on request only if its dirty. I clean it and give it. If the fork is clean it means I have requested and got it and have not eaten yet. In such a case I do not give it up.
- Initially the one with lower id gets preference. So if P1 requests forks initially, then he will get both forks. Deadlock situation wont arise.
- Starvation wont arise since a philosopher who has eaten has to wait if his neighbors have requested for fork.

Solution discussion

- Initially P1 has both forks, P5 has none and rest have one each.
- If P2 requests, then P1 has to give it as his forks are dirty. He cleans and gives it to P2. However, before P2 starts eating if P3 requests, then P2 need not give his left fork which is clean but has to give his right fork which is dirty. He cleans and gives it to P3.
- Thus a philospher whose neighbors are not wanting to eat gets the fork. If a neighbor wants to eat, he can request a fork that is not got by requesting. A clean fork(fork obtained via request and not used for eating yet) that the neighbor holds can be got only after neighbor has finished his meal which will turn the fork dirty.

1 hungry

3 hungry

5 hungry

3 finishes

2 hungry

3 hungry

What will be the sequence of forks changing hands?

homework

- More solutions
  --if a philosopher is hungry then check that both neighbors are not eating. Get the forks and eat. Before dropping the forks check if a neighbor is hungry and pass on fork.
  --allow only 4 philosphers to eat at once.
  -- resource hierarchy
  --chandy misra solution

MONITORS

Problems with Semaphores

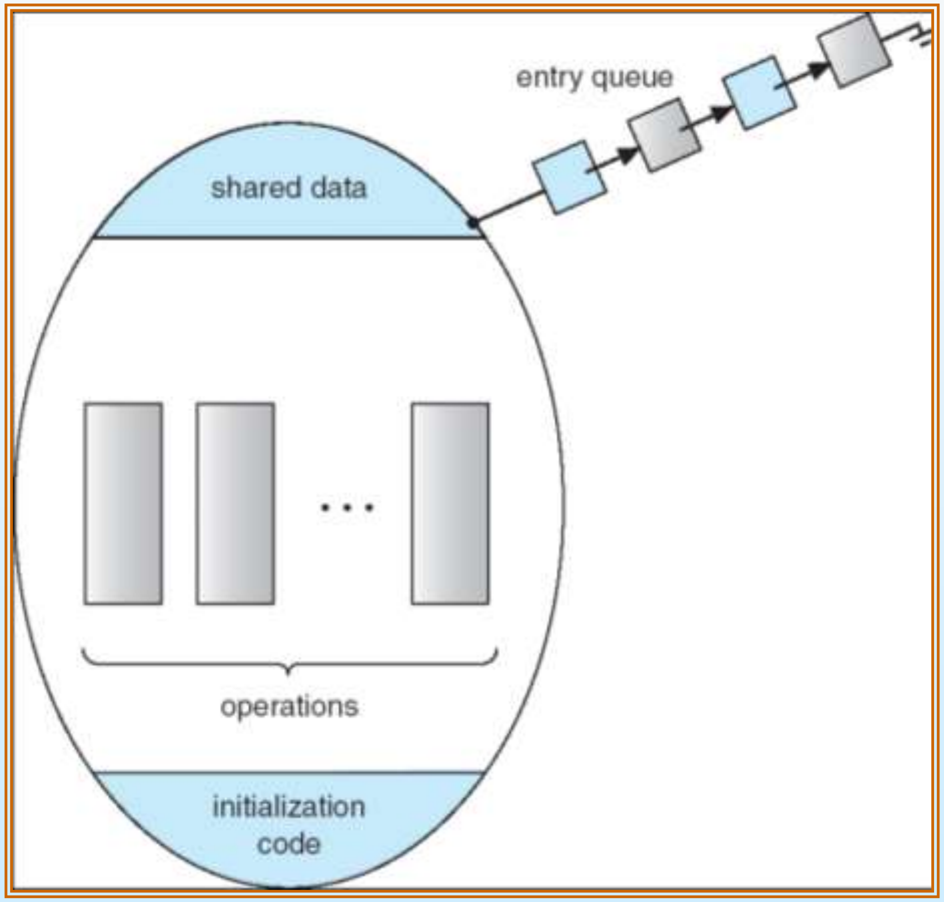
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

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

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

    procedure Pn (…) {……}

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

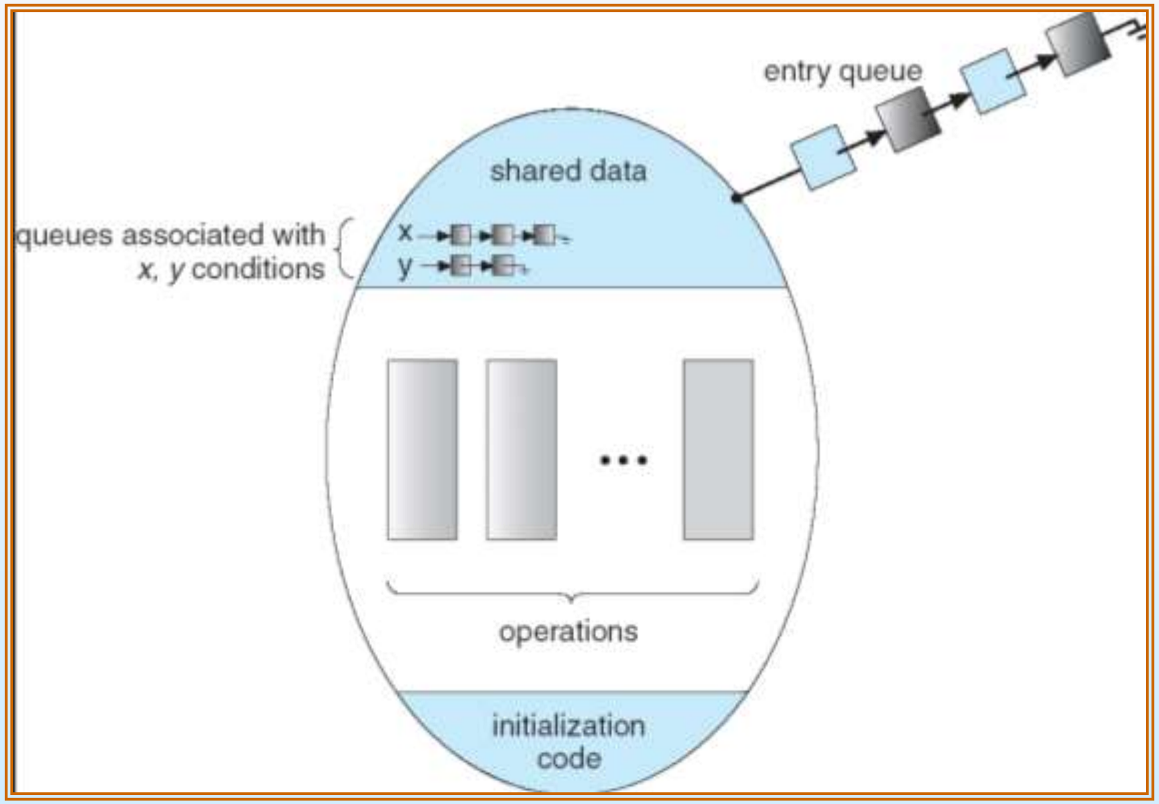
Schematic view of a Monitor



Condition Variables

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

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
   {
      enum { THINKING; HUNGRY, EATING) state [5] ;
      condition self [5];

      void pickup (int i) {
             state[i] = HUNGRY;
             test(i);
             if (state[i] != EATING) self [i].wait;
      }

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

Solution to Dining Philosophers (cont)

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

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

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                      P1
   wait (S);               wait (Q);
    wait (Q);              wait (S);
       .                       .
       .                       .
       .                       .
  signal  (S);           signal (Q);
  signal (Q);            signal (S);
```

- Starvation – indefinite blocking. A process may never

be removed from the semaphore queue in which it is suspended.

- Dining philosophers https://www.youtube.com/watch?v=rCiQc3ife90
- http://c2.com/cgi/wiki?DiningPhilosophers

Clarity needed

- Why not remove wait mutex1 and signal mutex1 in readers writers problem 2 on the top part(before reading) and keep it only in the bottom part(after reading). since wait read and signal read will make sure that only one reader is working on that code at one point in time— ???
- Can i remove mutex 3 and replace as wait(mutex1), wait(read) so that only one reader enters when a writer is writing – wont work since mutex1 is used at the end part of reader code as well so you cant have new readers and readers who are done reading waiting on the same queue
- In tannenbaum solution can I remove “if(state[i]= hungry) condition that is inside test[i] ? No as test[i] can be called by process i+1 or process i-1 in which case state[i] may or may not be hungry.
- http://gateoverflow.in/3498/gate2007-it-56
- http://gateoverflow.in/150943/process-synchronization

Petersons not guaranteed to work!

Modern CPUs pre-fetch memory operations and post writes to internal buffers. So there is no guarantee that a write on one CPU will become visible on another CPU immediately, nor that writes that occur in a particular order will be seen in that order on another CPU.

That may cause early entering into the critical section (while it's still in use by another thread) and early exiting from the critical section (when the work with the shared resource hasn't been finished yet).

For this reason one needs to ensure that the order of events occurring inside the CPU is seen the same outside. Memory barriers can ensure this serialization.