CS2106 Live Class: Synchronization

$_/|/|U_Ch@NgRu!$

May 10, 2021

1 Goal of synchronization

Ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

2 Race condition

Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place

2.1 Example

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

The code for the consumer process can be modified as follows:

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

Although both the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5 and that the producer and consumer processes execute the statements "`counter++`" and "`counter--`" concurrently. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result, though, is `counter` == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement "`counter++`" may be implemented in machine language (on a typical machine) as

$$\begin{aligned} &register_1 = \texttt{counter} \\ &register_1 = register_1 + 1 \\ &\texttt{counter} = register_1 \end{aligned}$$

where $register_1$ is one of the local CPU registers. Similarly, the statement $register_2$"`counter--`" is implemented as follows:

$$\begin{aligned} &register_2 = \texttt{counter} \\ &register_2 = register_2 - 1 \\ &\texttt{counter} = register_2 \end{aligned}$$

where again $register_2$ is one of the local CPU registers. Even though $register_1$ and $register_2$ may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of "`counter++`" and "`counter--`" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

2.2 Example

5 threads execute the following code, where globalVar(initialized as 0) is shared

```
for (int i=0; i<50000; i++)
    globalVar++;
```

2.2.1 The smallest possible value at the end of execution

(i.e. after all threads terminated)
Answer: **2!!!!!!!!**
Here take any two thread named thread A and thread B here

1. Thread A load globalVar in to Register(0) and swapped out
2. Thread B finishes 49999 iteration, and swapped out
3. Thread A wake up and Register(0)++ and write 1 into globalVar, and swapped out
4. Thread B wake up and load blobal Var into Register(1), and swapped out
5. Thread A wake up and finishes 49999 loop in total and swapped out
6. Thread B wake up and write the 1 into globalVar and end
7. A wake up and load the globalVar into register(1) and register(1)++ and write the 2 into globalVar and end.

The final value of globalVar is 2
The other 3 thread can be just put before thread B finished 49999 iteration

2.2.2 The biggest value of globalVar at the end

250000(at most 250000 loop in total and every loop add 1)

2.3 Example

5 threads execute the following code, where globalVar(initialized as 0) and i are shared

```
for (; i<50000; i++)
    globalVar++;
```

2.3.1 the smallest for globalVar at the end of execution

1
Consider two thread A and thread B

1. thread A load globalVar into register(0) and swapped out
2. thread B finishes 50000 loop(i=50000),and end
3. thread A wake up and register(0)++, and write 1 into the globalVar and end

the other three thread can just cooperate to play the role in the thread B above

2.3.2 the largest for globalVar at the end of execution

Incredibly large consider there are two thread A and thread B,

1. assume the globalVar is not inter-leaved at all
2. thread A load i into register(0) and finish one loop and before register(0)++ and just before it write 1 into i, it is swapped out
3. thread B finishes 49999 loop(i=49999) and swapped out
4. thread A wake up the write 1 into i and swapped out
5. thread B start at i=1 again and later thread A can refresh i to 2 again

The thread C can refresh A; D refresh C; E refresh D
Therefore the final result is very very large

3 Critical section

A segment of code, in which when one process is executing, no other process is to be allowed to execute in
Each process must request permission to enter its critical section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

3.1 Key points of critical section

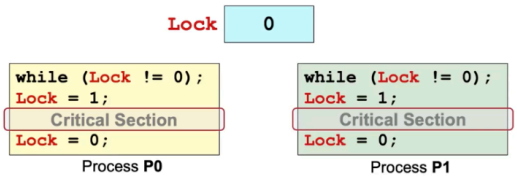
1. Mutual exclusion: only one process in the critical section at a time
2. Progress: a. If the critical section is currently free and if there are some process wish to enter their critical section; b. only those who are not executing in their remainder can be candidate; c. the selecetion cannot be dalayed/postponed indefinitely
3. Bounded wait: If the process queue up, it can eventually get the critical section
4. Independence: critical section should not be affected by code who is not in critical section

4 Problem of failure of synchronization

1. Incorrect output/behavior(usually due to lack of mutual exclusion
2. Deadlock: All process blocked → no progress
3. Livelock:(usually related to deadlock avoidance mechanism; processes keep changing state to avoid deadlock and make no other progree; typically process are not blocked)
4. starvation: some process repeatedly get the critical section while some other never get

5 High level language implementation

5.1 HLL: Attempt 1

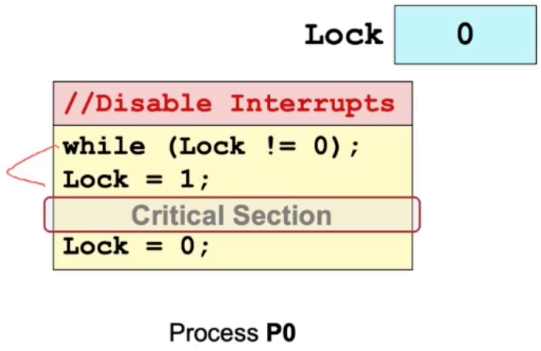


While lock is not 0, cannot enter; if it's 0, make it 1, and enter the critical section

5.1.1 Issue of Attemp1

1. What if two processed enter at the same time when the lock is 0

5.2 HLL: (Attemp 1 Fixed)

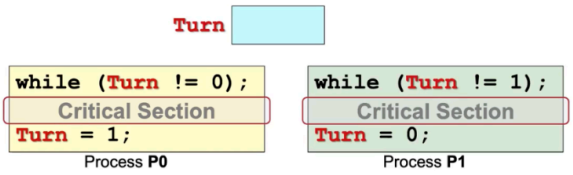


1. Between reading the lock and writing the lock Disable interrupts and scheduling(won't allow any process to come in)
2. The entire system is disabled by the timer

5.2.1 The problem of Attemp 1 fixed

1. The method only works on a single core
2. if the other process runnign the same code, and it can enter the process

5.3 HLL: Attempt 2



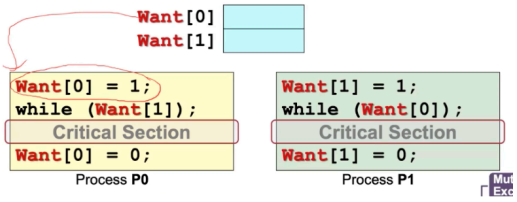
Switch by a value turn, the value can be 1 or 0; by this value to decide which process get the critical section; The process read the turn and see whether it's its own turn to get into the critical section

5.3.1 The analysis of attemp2

1. Mutual exclusion: it achieves mutual exclusion

2. Progress:not fullfill, the problem: what if it's the 0 process's turn but the process 0 enter the critical section tomorrow
3. independence: failed, in the scene above, the process 1 was prevented to get the critical section by process 0
4. Bounded wait:fail process 1 may wait for very lone time in the above example

5.4 Attempt 3

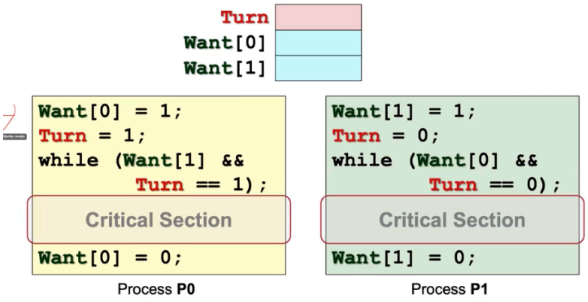


There is a list of flags, when a process wanna enter the critical section, they will set their corresponding critical section to be 1; If the other process need the critical section, it will wait; after the other process finished the job in critical section, it will let set the flag to 0 and let the waiting process in

1. Mutual exclusive: yes
2. there is a live lock, two processes wait for each other

5.5 Attempt 4: Peterson' s Solution

Combine attempt 2 and attempt 3 (Peterson' s Solution)



The process indicate the want, if they both indicate, depend on it's whose turn

5.5.1 Analysis of Attempt4

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

5.5.2 Problem of Attempt4

1. if one process uses the critical section for long, the other waiting process will burning power to do the while loop
2. the solution is complex
3. hard to generalized to multiple process(N processes)

5.5.3 Proof of how Peterson's solution works

```
//Initial Value
int turn=0;
int want[2];
//The lock
void lock(int self, int other)
{
   want[self]=1;
   turn=other;
   while(want[other] && turn==other);
}
//The release
void leave(int self)
{
   want[self]=0
}
```

No dead block:

1. Assume 1: for every process i can and only itself can change value of want[i]
2. Assume 2: variable "turn" can be changed by both processes, which means there may be interleaving
3. Assume 3: no processes i can change the variable "turn" to i

4. Assume 4: no matter how interleaving, the variable "turn" must be either 1 or 0(not both) at certain moment

5. if two process both try to modify the want in interleaved manner(which means they **both update their want to 1**), the variable "turn" must be either 0 or 1, when the first "while" judgement between the two process is running on process i

   (a) turn==(i+1)%2

      i. process (i+1)%2 still does not write i to variable"turn": the process (i) cannot change the turn thereby
      the process (i+1)%2 also still does not execute while loop, because it still does not write the "turn", so the while loop of process i just burns
      There must be a time when the process (i+1)%2 writes "turn" to i, and thereafter no one can change the "turn" anymore without the calling of release function in process i, in this condition, because "turn" is i and both process make their own want as 1, the process i will pass the while burn;

      ii. process (i+1)%2 wrote i to variable "turn" but the process i wrote (i+1)%2 later: In this case, the state of "turn" will thereafter be fixed without the release function in process i, later, when the process (i+1)%2 execute "while" loop, it must be able to pass(because turn!=i);

   (b) turn==i process i must have written to "turn" beforehand(this can be proved by contradiction with the fact that process i is doing "while" check), process (i+1)%2 must have written to "turn" otherwise the "turn" cannot be i; hence the "turn" will thereafter be fixed without the release function in process i; the process will pass the "while" loop

6. Once one process passed the while(get-lock process), the variable "turn" is fixed without calling release() function in the get-lock process and there is no chance for the other process to pass the while loop unless the calling of release function in the get-lock process because the other process cannot change the "turn" to itself according to Assume 3 and it can not also change the want of the get-lock process according to Assume 1

7. Therefore, the mutual exclusive achieves

6 Hardware support:lock

That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section

There is one hardware instruction that can read the lock and write the new lock to it

6.1 Machine instruction to aid synchronization

TestAndSet Register, MemoryLocation

6.1.1 Behavior

1. Load the current content at MemoryLocation into Register
2. Stores a 1 into MemoryLocation

IMPORTANT: The above is performed as a single **atomic machine operation, that is as one uninterruptible unit/simultaneously**
The reason that simultaneous rather than just interrupt is that in multi-core machine, the interrupter can only interrupt process on one core to change the status of the lock; if all core are interrupted and wait, it's very source-wasting

6.1.2 How the instruction is used in practice

```
void EnterCS(int * Lock)
{
   while( TestAndSet(Lock) == 1);
   //if the set is 1, modify it 1 to 1(does not change anything);
   //but if it's 0, this will write it to 1
}

void ExitCS(int * Lock)
{
   *Lock = 0;//set the Lock to 0
}
```

6.1.3 Analysis

This is a good implementation, the problem is that when one process is in critical section, the other process will burn doing the while loop
This is called spin lock(employs busy waiting; while waiting, do spining in the while loop)

6.2 Some alternative instruction

1. Compare and Exchange(The prof's favorite, so must have a look...)

2. Atomic Swap
3. Load Link/Store Conditional

7 Semaphore

7.1 Definition

An generalized synchronization mechanism(Because the hardware-based solutions to the critical-section problem presented are complecated)
A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). (as shown in 7.2

(ps:Only behaviors are specified → can have different implementation(Somewhat likes protocol)
Created by Edgar W. Dijkstra(who also created the famous Dijkstra Algo))
(ps:The wait command P(S) decrements the semaphore value by 1.; The V(S) i.e. signals operation increments the semaphore value by 1; The letters come from the Dutch words Probeer (try) and Verhoog (increment). V stands for signal and P stands for wait, because Dijkstra is Dutch)
Provides

1. A way to block a number of process(known as sleeping process)
2. a way to unblock/wake uo one or more sleeping process

7.2 Semaphore: Wait() and Signal(), implementation 1.0

A semaphore S contains integer value(initialized to any non-negative value)
The S means S number of processes can enter the critical section

```
Wait(S)
//If s<=0, means there is no body else allowed to enter the critical section, blocks (go to
    sleep)
//If s>=0 Decrement S
//Also known as P() or Down()
\\defination
wait(S) {
   while S <= 0
      ; // no-op
      S--;
}
```

```
Signal(S)
//Increments S
//Wait up one waiting/sleeping process if any
// Also known as V() or Up()
//defination
signal(S) {
    S++;
}
```

IMPORTANT: the increment and decrement are done within the operating system

7.3 Semaphore: Wait() and Signal(), implementation 2.0

7.3.1 Problem of 1.0

busy waiting: While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code
This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

7.3.2 Solution

Rather than engaging in busy waiting, the process can block itself.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

```
//This part is pretty optional
typedef struct{
    int value;
    struct process *list;
} semaphore;

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

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

7.4 Semaphores: properties

Given: $S_{Initial} \geq 0$
$S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$

7.5 Two types of semaphore

1. **General semaphore/counting semaphores S:** $S \geq (S = 0, 1, 2, 3, ...)$ also called ; control access to a given resource consisting of a finite number of instances
2. **Binary semaphore/mutex locks S**: S=0 or 1; locks that provide mutual exclusion; deal with the critical-section problem for multiple processes

General semaphore is provided for convenience, while binary semaphore is sufficient
General semaphore can be mimicked by binary semaphores

7.6 Analysis

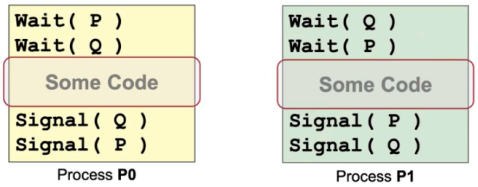
Mutual exlusion: Achieved
$N_{CS} = Numberofprocessincriticalsection = Processthatcompletedwait()butnotsignal()$
$N_{CS} = \#Wait(S) - \#Signal(S)$

1. $S_{Initial} = 1$
2. $S_{current} = 1 + \#Signal(S) - \#Wait(S)$

$S_{current} + N_{CS} = 1$
Deadlock:
Deadlock means all process stuck at wait (S); but it will not happens here because $S_{current}+N_{CS} = 1$, if there is no process in critical section, $S_{current}$ will be larger than 0
Starvation: depends
Suppose P1 is blocked at wait(S)

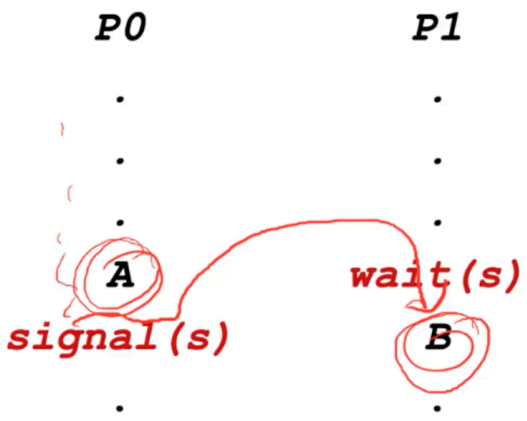
P2 is in CS, exits CS with signal(S), there is no specification to pick which waiting process, so starvation is possible

7.7 Incorrect use of semaphore



The P0 takes P and wait for Q; The P1 takes Q and wait for P and hence blocked

7.8 semaphore can be used as a general synchronization tool



1. Execute B in P1 only after A executed in P0
2. Use semaphore s initialized to 0 and let B wait(S) only when A executed and signaled the B can execute

7.9 Summary of use of semaphores

7.9.1 Critical section

```
BinarySemaphore mutex=1;
void use_restroom()
```

```
{
    wait(mutex);
    critical_section(); // this process is critical section
    signal(mutex);
}
```

7.9.2 Sage-distancing problem/high level synchronization

```
Semaphore sem=N; //the semaphore is set to N
void cient(){
    wait(sem);
    safe_eating(); //this process is no critical section
    \\but some section need high level synchronization
    signal(sem);
}
```

7.9.3 General synchronization

```
Semaphore sem=0;
```

```
// Process P1:
produce(X)
signal(sem)
```

```
// process P2:
wait(sem)
consum(X)
```

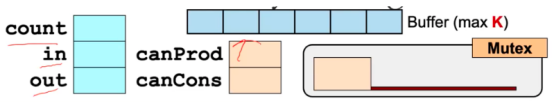
In the case above, the process P2 will wait until the P1 finish the job

8 Producer Consumer Problem:Bounded-Buffer

8.1 Busy waiting version

Processes share a bounded buffer of size K

1. Producers produce items to insert in buffer(if <K)
2. Consumers remove items from buffer only when buffer is not empty(if >0)



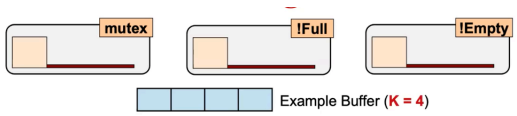

```
//Producer Process
while (TRUE) {
Produce Item;
while(!canProduce);
wait( mutex ); // to prevent data race
if (count < K) {
buffer[in] = item;
in = (in+1) % K;
count++;
canConsume = TRUE; }
else
canProduce = FALSE;
signal( mutex );
}
```

```
//Consumer Process
while (TRUE) {
while (!canConsume);
wait( mutex ); // to prevent data race
if (count > 0) {
item = buffer[out];
out = (out+1) % K; count--;
canProduce = TRUE;
}
else canConsume = FALSE;
signal( mutex );
Consume Item;
}
//Initial Values:
//count=in=out=0
//mutex = S(1) //semaphore with initial value 1
//canProduce = TRUE and canConsume = FALSE;
```

8.1.1 Analysis

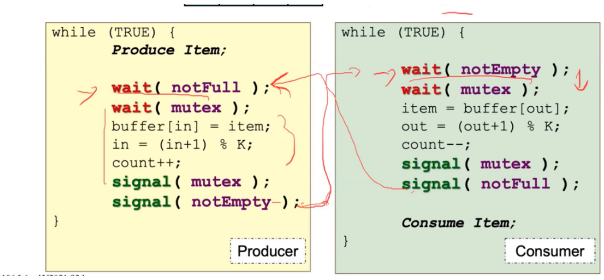
1. data race/mutual exclusive
The wait(mutex)/signal(mutex) prevents the two processes both use the critical section(within the wait and signal) 2. It's a little complecated
multi-core
3. Ten producer and ten consumer, and the buffer is full, the ten core will wait for the critical section, also burning while loop

8.2 Blocking version




```
// Producer Process
while (TRUE) {
Produce Item;
wait( notFull );
wait( mutex );
buffer[in] = item;
in = (in+1) % K; count++;
signal( mutex );
signal( notEmpty );
}
```

```
//Consumer process
while (TRUE) {
 wait( notEmpty ); wait( mutex );
item = buffer[out]; out = (out+1) % K; count--;
signal( mutex ); signal( notFull );
Consume Item;
}
```



The !Full and !Empty works as Sage-distancing problem so the initial value>1
The mutex works as critial section semaphore The producer wait for notFull cemaphore
The consumer waits for notEmpty cemaphore
They produce the signal for each other
logic is the same, but no need to count
The function of wait(mutex): to achieve the mutual exclusive
the function of wait(notFull) and wait(notEmpty): to count

8.2.1 Example of producer consumer: Massage passing

```
MessageQueue mQueue =new MessageQueue(K);
//producer
while(TRUE)
{
    Produce Item;
    mQueue.send(Item);
}

//receiver
while(TRUE)
{
    mQueue.receive(Item);
    Consume Item;
}
```

Have some message buffer/queue with capacity K;
The producer send the item to the queue(if the queue is not full, otherwise the OS will block);
The consumer will receive the item if the queue is not empty(the system will block if it's empty)
The synchronization of the queue is done by the system, which is called implicit synchronization.
The pipe lock in tutorial 6 is related to this

9 Example of semaphore: Reader Writer

Processes share a data structure: D:

1. Reader: Retrieves information from D
2. Writer: Modifies information in D

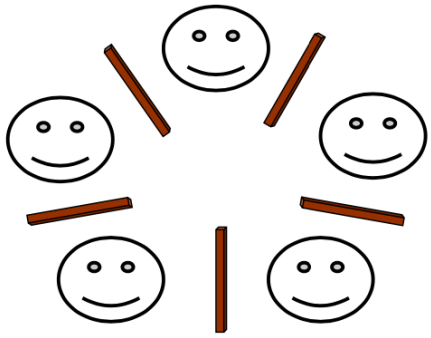
A simple version

```
// Initial Values:
roomEmpty = S(1)
mutex = S(1)
nReader = 0

// Writer Process
while (TRUE) {
    wait( roomEmpty );
    Modifies data
    signal( roomEmpty );
}

// Reader Process
while (TRUE) {
    wait( mutex ); //To mutual excluse other reader;
    //The wait() above guarantee every time there is only one reader taken into count
    nReader++;
    if (nReader == 1) wait( roomEmpty ); // only the first read should wait for the Modify
        finish(the roomEmpty semaphore)
    signal( mutex );To mutual excluse other reader

    Codes for Reading data// The read process is out of critical section

    wait( mutex );
    nReader--;
    if (nReader == 0) signal( roomEmpty ); //Finish all reading the release the critical
        section
    signal( mutex );
}
```

If one reader gets inside and read, the **nReader** will increase as the #reader increase
The problem of the simple version is that, inreality, there is possible that a plenty of reader coming at the same time; In that case, the writer may be starved

10 A "useless: problem: Dining Philosophers

10.1 specification



Five philosophers are seated around a table
There are five single chopstick placed between each pair of philosopher
When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right
Devise a deadlock-free and starve-free way to allow the philosopher to eat freely

10.2 Attempt 1

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)
//For philosopher i
while (TRUE){
    Think( );
    //hungry, need food!
    takeChpStk( LEFT );
    takeChpStk( RIGHT );
    Eat( );
    putChpStk( LEFT );
    putChpStk( RIGHT );
}
```

10.2.1 Analysis

Deadlock: If all philosophers simultaneously takes up the left chopstick, trhen none can proceed

10.3 Fix the attempt 1

The philosophers put down the left chopstick if the right chopstick cannot be taken
The fix of the attempt can avoid dead lock but it can result in live lock; each time, philosophers comity each other then block each other, on and on

10.4 Attempt 2

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)
//For philosopher i
while (TRUE){
    Think( );
    wait( mutex ); // semaphore was used
    takeChpStk( LEFT );
    takeChpStk( RIGHT );
    Eat( );
    putChpStk( LEFT );
    putChpStk( RIGHT );
    signal( mutex );
}
```

10.4.1 Analysis

It works
Each time only one philosopher can eat, so it's inefficient

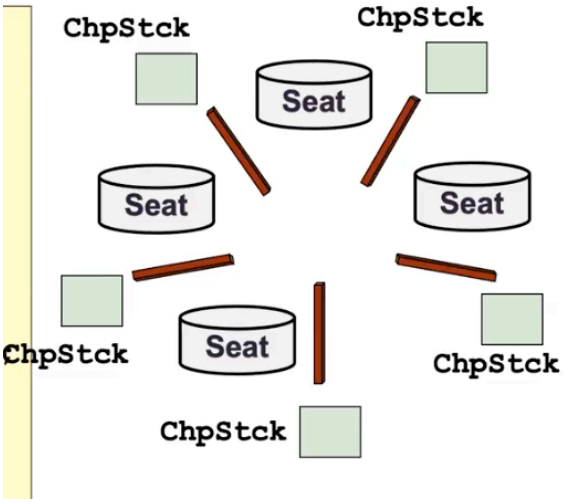
10.5 Tanenbaum Solution

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1 //indicate the desire to eat; like the variable "want" in HLL
    synchronization
#define EATING 2
int state[N]; //each philosopher has one state
Semaphore mutex = 1;
Semaphore s[N];
```

```
void takeChpStcks(i);
void safeToEat( i );
void putChpStcks( i )

void philosopher( int i )
{
   while (TRUE)
  {
    Think( );//
    takeChpStcks( i );
    Eat( );
    putChpStcks( i );
  }
}

void takeChpStcks(i)
{
   wait( mutex );
   state[i] = HUNGRY; // the state change should be protected by mutex
   safeToEat( i );
   signal( mutex );
   wait( s[i] ); // wait for local semaphore(won't affect other philosopher)
}

void safeToEat( i ) {

if( (state[i] == HUNGRY) && (state[LEFT] != EATING) && (state[RIGHT] != EATING) )
{
   state[ i ] = EATING;
   signal( s[i] ); // which means eat successfully
}
}

void putChpStcks( i )
{
   wait( mutex );
   state[i] = THINKING;
   safeToEat( LEFT );
   safeToEat( RIGHT );
   signal( mutex );
}
```

Why there is no dead lock: because the wait(s[i]) just wait for themselves

11 Limited Eater



If at most 4 philosophers are allowed to sit at the table (leaving one empty seat) Deadlock is impossible!

```
//Initial value:
seats = S(4)
chpStk = S(1)[5]
void philosopher( int i )
{ while (TRUE){
    Think( );
    wait( seats );
    wait( chpStk[LEFT] );
    wait( chpStk[RIGHT] );
    Eat( );
    signal( chpStk[LEFT] );
    signal( chpStk[RIGHT] );
    signal( seats );
}
}
//So the point of the limited eater is each time, only N-1 philosophers are allowed to
    think() so that dead block would not happen
```

POSIX semaphore

Popular

12 Synchronization for multiple independent shared variable

There are cases, multiple variables are shared and modified by multiple processes, but these variable themselves are independent(i.e. the change of one would not result in the change of other variables)(e.g. A array of integers where integers are shared by multiple processes)
In this case, if only one semaphore is applied, like the code bellow

```
Semaphore mutex = 1; //binary semaphore int A[N]; //shared array

//IN: read and remove one of the N values
int IN( int idx ) {
    int result;
    wait( mutex );
    result = A[idx]; //"remove" value
    A[idx] = -1;
    signal( mutex );
    return result;
}
//OUT: write into one of the N values. Below is an attempt to use semaphore to synchronize
     the tasks in operating on the array values
void OUT( int idx, int newValue ) {   wait( mutex );
    A[idx] = newValue;
    signal( mutex );
}
```

In this case, R and W of A[0] should wait for R and W while they in fact cannot affect each other, so this is very efficient(especially when N is very large)
An alternative way is to allocate an semaphore for every variable in array "A" as shown below

```
Semaphore mutex[N] = 1; //N binary semaphore int A[N]; //shared array
    int IN( int idx ) {
    int result;
    wait( mutex[idx] );
    result = A[idx]; //"remove" value
    A[idx] = -1;
    signal( mutex[idx] );
    return result;
}

void OUT( int idx, int newValue ) {
     wait( mutex[idx] );
```

```
    A[idx] = newValue;
    signal( mutex[idx] );
}
```

13 POSIX semaphore

Popular implementation of semaphore under Unix
Header:

```
#include <semaphore.h>
```

Compilation flage: gcc something.c –**lrt**
lrt Stands for "real time library"
Basic Usage:

1. Initialize a semaphore
2. Perform wait() or signal() on semaphore

13.1 Synchronization mechanisms for pthreads

Mutex (pthread_mutex):

1. Binary semaphore (i.e. equivalent Semaphore(1)).
2. Lock: pthread_mutex_lock()
3. Unlock: pthread_mutex_unlock()

Conditional Variables(pthread_cond):

1. Wait: pthread_cond_wait()
2. Signal: pthread_cond_signal()
3. Broadcast: pthread_cond_broadcast()