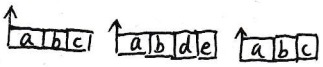
Cyclic Executives

Process	Period, T	Computation Time, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Advantages of being fully deterministic

```
loop
  wait_for_interrupt;
  produce_for_a; produce_for_b; produce_for_c;
  wait_for_interrupt;
  produce_for_a; produce_for_b; produce_for_d;
  produce_for_e;
  wait_for_interrupt;
  produce_for_a; produce_for_b; produce_for_c;
  wait_for_interrupt;
  produce_for_a; produce_for_b; produce_for_d;
end loop
```



All process periods must be a multiple of the minor cycle time.

Issues

The difficulty of incorporating processes with long periods

Sporadic activities are difficult to incorporate

Fixed-Priority Scheduling (FPS)

Each process has a fixed, static, priority which is computed pre-run-time

The runnable processes are excuted in the order determined by their priority.

Earliest Deadline First (EDF) scheduling

The runnable processes are executed in the order determined by the absolute deadlines of the processes.

The next process to run is the one with the shortest (nearest) deadline.

Preemption and Non-preemption

With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one.

In a preemptive scheme, there will be an immediate switch to the higher-priority process

With non-preemption, the lower-priority process will be allowed to complete before the other executes.

Preemption Preemptive schemes enable higher-priority processes to be more reactive, and hence they are prefered.

Alternative strategies allow a lower priority process to continue to execute for a bounded time

These schemes are known as deferred preemption or cooperative dispatching.

FPS and Rate Monotonic Priority Assignment

Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority.

$T_i < T_j \Rightarrow P_i > P_j$

Utilization-based Analysis

$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1)$ { Sufficient / Not necessary

Task families

↳ A set of tasks whose periods are all integer multiples of the shortest period in the family

Hyperbolic Bound

$\prod_{i=1}^{N} \left(\frac{C_i}{T_i} + 1\right) \leq 2$ { Sufficient / Not necessary

Utilization-based Test for EDF

$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$

EDF is dynamic and requires a more complex run-time system which will have higher overhead

Response-Time Analysis for FPS

$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil C_j$

hp(i) is the set of tasks with priority higher than task i.

Process	Period T	Computation Time C	Priority P
a	7	3	3
b	12	3	2
c	20	5	1

$R_a = 3$

$w_b^0 = 3$

$w_b^1 = 3 + \lceil \frac{3}{7} \rceil \cdot 3 = 6$

$w_b^2 = 3 + \lceil \frac{6}{7} \rceil \cdot 3 = 6$

$R_b = 6$

$w_c^0 = 5$

$w_c^1 = 5 + \lceil \frac{5}{7} \rceil \cdot 3 + \lceil \frac{5}{12} \rceil \cdot 3 = 11$

$w_c^2 = 5 + \lceil \frac{11}{7} \rceil \cdot 3 + \lceil \frac{11}{12} \rceil \cdot 3 = 14$

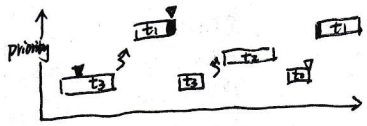
$w_c^3 = 5 + \lceil \frac{14}{7} \rceil \cdot 3 + \lceil \frac{14}{12} \rceil \cdot 3 = 17$

$w_c^4 = 5 + \lceil \frac{17}{7} \rceil \cdot 3 + \lceil \frac{17}{12} \rceil 3 = 20$

$w_c^5 = 5 + \lceil \frac{20}{7} \rceil \cdot 3 + \lceil \frac{20}{7} \rceil \cdot 3 = 20$

$R_c = 20$

Priority Inversion



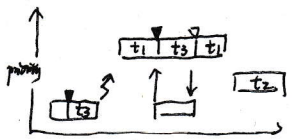
Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for a lower-priority task to complete

When t1 preempts t3 and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that t1 would be blocked no longer than the time it normally takes t3 to finish with the resource, the would be no problem because the resource cannot be preempted.

However, the low-priority task is vulnerable to preemption by medium-priority tasks (like t2), which could inhibit t3 from relinquishing (放弃) the resource. This condition could persist, blocking t1 for an indefinite period of time.

⇒ Priority Inheritance

We can reduce priority inversion with a priority-inheritance algorithm. The priority-inheritance protocol ensures that a task that owns a resource executes at the priority of the highest-priority task blocked on that resource.



Response time and utilisation

$R_2 = C_2 + \lceil \frac{R_2}{T_1} \rceil C_1$

$\begin{cases} T_1 = C_2 + C_1 \leftarrow \text{Worst case} \\ T_2 = C_2 + 2C_1 \end{cases}$

$C_1 = T_2 - T_1$

$C_2 = 2T_1 - T_2$

$\lambda = \frac{T_2}{T_1}$

$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{T_2 - T_1}{T_1} + \frac{2T_1 - T_2}{T_2} = \frac{T_2}{T_1} - 1 + \frac{2T_1}{T_2} - 1 = \lambda + \frac{2}{\lambda} - 2$

$\frac{dU}{d\lambda} = 1 - \frac{2}{\lambda^2} = 0$

$\lambda = 2^{\frac{1}{2}}$

$U \leq 2(2^{\frac{1}{2}} - 1)$

$= 2^{\frac{1}{2}} + 2^{\frac{1}{2}} - 2$

C++ 11

[Thread] & [Mutex]

```
#include <iostream>
#include <thread>
#include <time.h>
using namespace std;
char letters[11] = "ABCDEFGHIJ";

int main()
{
    thread threads[10];
    for(int i=0; i<10; i++)
        threads[i] = thread(printChar, letters[i], i+10);
    for(int i=0; i<10; i++)
        threads[i].join();
    return 0;
}

void delay(int second)
{
    clock_t start = clock();
    while(clock()-start <= second*1000);
}

void printChar(char c, int i)
{
    m.lock(); ← Mutex
    while(i--)
    {
        cout << c << flush;
        delay(1);
    }
    cout << endl;
    m.unlock(); ← Mutex
    return;
}
```

函数名 参数 参数

#include <mutex> 无则报错

mutex m;

thread th;
th = thread(func, para ...)
th.join();

[Lock_guards]

```
mutex m;
...
void printChar(char c, int i)
{
    lock_guard<mutex> ul(m);
    ...
}
```

lock_guard 是 mutex 的一种包装.
程序结尾自动释放 mutex.

[Unique_lock]

unique_lock is a version of lock_guard that should be used with condition variables.

```
#include <iostream>
#include <time.h>
#include <condition_variable>
#include <mutex>
#include <thread>
using namespace std;

condition_variable cond_var;
mutex m;

int main()
{
    thread t1(producer);
    thread t2(consumer);
    t1.join();
    t2.join();
}
```

[Mutex + Condition variable]

```
mutex m;
condition_variable cv;

One of the thread function(...){
{
    unique_lock<mutex> ul1(m);
    while(...cannot make progress...){
        cv.wait(ul1);
    }
    ...do what you have to do...
    cv.notify_all(); //wake up all waiting threads
} //exiting the block releases the mutex
```

```
void consumer()
{
    while(true)
    {
        unique_lock<mutex> ul1(m);
        while(buffer_contents == 0)
            cond_var.wait(ul1);
        buffer_contents--;
        cond_var.notify_all();
    }
}

Void producer()
{
    while(true)
    {
        unique_lock<mutex> ul2(m);
        while(buffer_contents == N)
            cond_var.wait(ul2);
        buffer_contents++;
        cond_var.notify_all();
    }
}
```

fifo out

fifo in

```
Void fifo_in(char ch)
{
    if(in_index == N)
        in_dex = 0;
    fifo[in_index++] = ch;
    return;
}

char fifo_out()
{
    char ch;
    if(out_index == N)
        out_index = 0;
    ch = fifo[out_index];
    fifo[out_index++] = ' ';
    return ch;
}
```

pthread C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

int main()
{
    pthread_t thr[5];
    int ID[5];
    int i;
    for(i=0; i<5; i++)
    {
        fork[i] = true;
        ID[i] = i;
        dinner[i] = 0;
        pthread_create(&thr[i], NULL, philosopher, &ID[i]);
        pthread_join(thr[i], NULL);
    }
    return 0;
}
```

简单的程序
利用 mutex 保护 shared variable
⇒ fork[5]

将空指针→int指针 再用*读取数值

```
void *philosopher(void *ID)
{
    int n = *((int*)ID);
    while(true)
    {
        printf("Philosopher %d is thinking\n", n);
        if(n==0)
        {
            while(fork[(n+1)%5] != true);
            fork[(n+1)%5] = false;
            printf("Takes the fork");
            while(fork[n] != false);
            fork[n] = false;
            ...
        }
        else
        {
            while(fork[n] != true);
            fork[n] = false;
            while(fork[(n+1)%5] != true);
            fork[(n+1)%5] = false;
        }
    }
```

[pthread Mutex]

线程变量：pthread_t th
线程创建：pthread_create(&th, NULL, function_name, &data);
mutex变量：pthread_mutex_t lock
mutex初始化：pthread_mutex_init(&lock, NULL);
线程加入：pthread_join(th, NULL)

```
{ pthread_mutex_lock(&lock);
  pthread_mutex_unlock(&lock);
```

足够保护哲学家

```
⇒ pthread_mutex_lock(&mutex[(ID+1)%5]);
  while(fork[(ID+1)%5] != 1);
  fork[(ID+1)%5] = 0;
⇒ pthread_mutex_unlock(&mutex[(ID+1)%5]);
```

PS: PPT上的 Java 哲学家未使用 mutex 保护在验证 fork 是否 available 后，如果加入一个长时间的 delay，那么所有哲学家都会拿起叉子，即5个哲学家0.4秒会同时拥有同一把叉子，应该然是错的

加入 delay 后，大大增加 race condition 的可能
while cload 后）被切出线程
没来得及把数置0，即两个线程同时进入

Synchronization

[2011-2012]

You are offered the following implementation of a thread-safe Counter in Java:

```
public final class Counter {
    private long value = 0;
    public long getValue() {
        return value;
    }
    public long increment() {
        return ++value;
    }
}
```

(i) Correct the obvious mistakes in the implementation, without restructuring the code.

(ii) Write down an alternative implementation of this class using Java 1.5's AtomicInteger

(iii) How might AtomicInteger be efficiently implemented on an Intel 486 architecture?

(iv) What is the ABA problem?

[2012-2013]

You are asked to implement a Reader-Writers application using a shared buffer in multi-threaded C++ 11. You should provide two functions, read_int() and write_int(i), which will remove and insert a single integer into a 16 deep FIFO buffer but will lock if the buffer is empty or full respectively. Many threads may be using each of the two functions at one time.

(a) Write correct C++ 11 code for the read_int() and write_int() functions, and for the shared state.

(b) How will your code perform as the number of threads becomes large?

(c) Why is Peterson's algorithm unlikely to be useful in a modern implementation?

[2013-2014]

You are asked to implement a Reader-Writers application using a shared buffer in multi-threaded C11. You may use either the pthread or the C11 threading libraries. You should provide two functions, read_int() and write_int(i) which will remove and insert a single integer into a 16 deep FIFO buffer but will block if the buffer is empty or full respectly. Many threads may be using each of the two functions at one time.

(a) Write correct C11 code for the read_int() and write_int(i) functions, and for the shared state.

(b) How will your code perform as the number of threads becomes large?

(c) Why is Peterson's algorithm unlikely to be useful in a modern implementation?

[2014-2015]

You may answer this question using either the Java programming language, or using pthreads or cthreads with the C programming language.

(a) Write down a simple pattern using mutexes and condition variables that will suffice to protect race conditions in multi-thread software.

(b) How might your pattern be inappropriate for use in a real-time system?

(c) Using the dining philosophers example, explain the advantages of an alternative approach using semaphores. In what way does the structure of this code differ from that using condition variables? Ensure your system does not deadlock.

[ABA] In multithread computing, the ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two threads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates the assumption.

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave.

① Process P1 reads value A from shared memory

② P1 is preempted allowing P2 to run.

③ P2 modifies the shared memory value A → B and back to A before preemption

④ P1 begins execution again, see that the shared memory has not changed and continues.

[Why is Peterson's algorithm unlikely to be useful in modern implementation]

In modern architecture, a CPU cache could screw up the mutual exclusion requirement.

The problem is called cache-coherence, and it is possible that the cache used by Process 0 on CPU 0 sets flag[0] equal to true, while Process 1 on CPU 1 still thinks flag[0] is false. In this case, both critical sections would enter and mutual exclusion fails.

The use of a buffer to link two tasks known as # Producer - Consumer.

[Mutual Exclusion with Busy Waiting]

```
while(true) {
    while (free==0);
    free=0;
    critical_region();
    free=1;
    noncritical_region();
}
```

Fatal Race Condition

```
while(true) {
    while (turn!=0);
    critical_region();
    turn=1;
    noncritical_region();
}
```

Strict Alternation

[Peterson's Algorithm]

```
int turn;
int interest[2];
void enter_region(int thread)
{
    int other;
    other = 1 - thread;
    interested[thread] = true;
    turn = other;
    while (turn==other && interested[other]);
}
void leave_region(int thread)
{
    interested[thread] = false;
}
```

```
#Thread 0
while(true) {
    enter_region(0);
    critical_region();
    leve_region(0);
    noncritical_region();
}
```

Waste valuable Process time

[Non-busy synchronization: Sleep and Wakeup]

```
#define N 100
int count=0;
void producer(void)
{
    int item;
    while(true) {
        item = produce_item();
        if (count==N) sleep();
        insert_item(item);
        count = count + 1;
        if(count==1)
            wakeup(consumer);
    }
}
```

```
void consumer(void)
{
    int item;
    while(true) {
        if (count==0) sleep();
        item=remove_item();
        count = count -1;
        if(count== N-1) {
            wakeup(producer);
        consumer_item(item);
    }
}
```

The race condition can occur because access to count is unconstrained.
Eg. The buffer is empty and the consumer has just read count to see if it is 0. At this instant, the scheduler decides to stop running the consumer temporily and start running the producer. The producer inserts an item in the buffer, increment count and notice that it is now 1. The producer wakes consumer up.
⇒Unfortunely, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of item it previously read, find it to be 0 and go to sleep.
⇒Later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

※A wakeup sent to a proces that is not (yet) sleeping is lost.

[Java's Synchronized wait() and notify()]

```
public final class ProducerConsumer {
    final int N=100;
    private int count=0;
    private Object lock = new Object();

    void producer() {
        Object item;
        while(true) {
            item = Producer.produce_item();
            synchronized(lock) {
                while(count==N)
                    lock.wait();
                Buffer.insert_item(item);
                count = count+1;
                lock.notifyAll();
            }
        }
```

```
void consumer() {
    Object item;
    while(true) {
        synchronized(lock) {
            while (count==0)
                lock.wait();
            item=Buffer.remove_item();
            count=count-1;
            lock.notifyAll();
        }
        Consumer.consume_item(item);
    }
```

※Use while()

※当两个线程访问 Object 的一个 synchronized(this) 同步代码时，一个时间内只有一个线程得到执行

[Template] synchronized (o) {
    while (cannot make process)
        o.wait();
    do something;
    o.notifyAll();
}

[Semaphores] down(): if (S>0) S--; else wait;
up(): if (thread_wait) wake_one; else S++;

※ wait() is atomic, one task will complete execution of this statement before the other begins

```
#include <semaphore.h>
sem_t mutex, empty, full;
void initialisation(void){
    sem_init(&mutex,0,1);
    sem_init(&empty,0,N);
    sem_init(&full,0,0);}
```

```
void producer(void) {
    Item* pitem;
    while(true){
        pitem = produce_item();
        sem_wait(&empty);  ←初始值 N 说明有N个空位
        sem_wait(&mutex);
        insert_item(pitem);
        sem_post(&mutex);
        sem_post(&full);}  ←初值一个位 full
```

```
void consumer(void) {
    Item* pitem;
    while(true){
        sem_wait(&full);  ←判断是否为空 buffer.
        sem_wait(&mutex);
        pitem = remove_item();
        sem_post(&mutex);
        sem_post(&empty);
        consume_pitem(pitem);}}
```

Java 1.5

```
public class Semphore {
    private int value;
    public Semphore(int initial){
        value = initial;}
    synchronized public void up(){
        ++value;
        notify();}
    synchronized public void down(){
        while(value==0)
            wait();
        --value;}
```

One task will execute a wait(mutex) with mutex=1, which will allow another the task to process into its critical section and set mutex to 0, and the delayed. Once the first task has exited its critical section, it will signal(mutex). This will case the semaphore to become 1 again and allow the second task to enter its critical section (and set mutex to 0 again)
With a wait/signal bracket around a section of code, the initial value of the semaphore will restrict the maximum amount of concurrent execution of the code. If the initial value 0, no tak will ever evere enter. if it is 1 then a single task may enter (that is, mutual exclusion). For values greater than one, the given number of concurrent executions of the code is allowed.

[The Reader and Writers Problem]

Many readers can access the database at the same time, but only one writer, which must have exclusive access.

```
#include <semphore.h>
sem_t mutex, db;
int rc=0;
void initialisation(void){
    sem_init(&mutex,0,1);
    sem_init(&db,0,1);
    rc=0;}

void writer(void){
    while(true){
        invent_data();
        sem_wait(&db);
        write_database();
        sem_post(&db);}}
```

```
void reader(void){
    while(true){
△       sem_wait(&mutex); ←保护变量 rc
        rc=rc+1;
        if(rc==1) ←有一个人在读，就无法写入
○           sem_wait(&db);
△       sem_post(&mutex);
        read_database();
△       sem_wait(&mutex);
        rc=rc-1;
        if(rc==0) ←无人读取，可以写入
○           sem_post(&db);
△       sem_post(&mutex);
        use_data();}}
```

Give Readers priority
A writer might never gain access

maps: ++i → i.incrementAndGet();
i++ → i.getAndIncrement()
--i → i.decrementAndGet()
i-- → i.getAndDecrement()
i=x → i.set(x)
x=i → x=i.get(x).

[Atomic Integer]

#Old style

```
public final class Counter {
    private long value=0;
    public synchronized long getValue(){
        return value;
    }
    public synchronized long increment(){
        return ++value;
    }
}
```

#New style

```
import java.util.concurrent
public final class NonblockingCounter {
    private AtomicLong value;
    public long getValue(){
        return value.get();
    }
    public long increment(){
        int V;
        do {
            V=value.get();
        }while(!value.compareAndSet(V,V+1));
        return V+1;
    }
}
```

如果设置没成功（当前值无效，即被其它线程修改）

Model Checking

A sequential program has a single thread of control

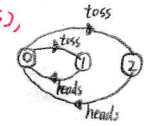
A concurrent program has multiple threads of control allowing it perform multiple computations in parallel and to control multiple external activities which occur at the same time.

Models and Model Checking

A model is an abstract, simplified representation of the real world.

Models are described using state machines, known as Labeled Transition Systems (LTS). These are textually described in a Process Algebra as finite state process (FSP) and displayed and analysed by the LTSA model checking analysis tool.

COIN = (toss → HEADS | toss → TAILS),
HEADS = (heads → COIN)
TAILS = (tails → COIN)



Process and threads

[Modelling process]

- action begin with lowercase letters
- Process begin with uppercase letters

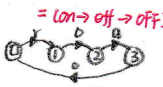
A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of a atomic actions

※ If x is an action and P a process then (x→P) describes a process that initially engages in the actim x and then behaves exactly as described by P.

ONESHOT = (once → STOP)

△Repetitive behaviour uses recursion

SWITCH = OFF ⇒ SWITCH = OFF = (on → (off → OFF))
OFF = (on → ON)
ON = (off → OFF)



△FSP model of a traffic light

TRAFFICLIGHT = (red → orange → green → orange → TRAFFICLIGHT)

※ If x and y are actions then (x→P | y→Q) describes a process which initially engages in either of the actions x or y. After the first action has occured, the subsequent behaviour is described by P if the first action was x and Q. if the first action is y.

△FSP model of a drinks model

DRINKS = (red → coffee → DRINKS | blue → tea → DRINKS).

※ Process (x→P | x→Q) describes a process which engages in x and then behaves as either P or Q)

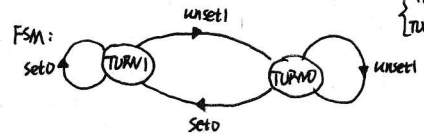
[TURN variable]

```
int turn = 1
while (turn != me);
turn = 1 - me;
```

me: thread 0 / thread 1

P0: turn = 1-0 = 1 → set0
P1: turn = 1-1 = 0 → unset1

FSM:



TURN1 = (set0 → TURN1 | unset1 → TURN0)
TURN0 = (set0 → TURN1 | unset1 → TURN0)

P0	isunset0
turn==1	turn!=0 → ~~isset~~
turn==0	turn!=1 → isset1

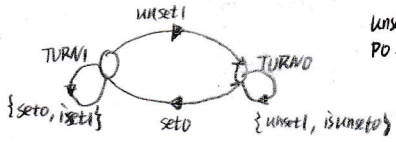
[Testing the Variable]

```
P0: while (turn != 0);
    turn = 1;
P1: while (turn != 1);
    turn = 0;
```

turn=0 / turn=1

※[FSM]

TURN1 = (set0 → TURN1 | unset1 → TURN0 | isset1 → TURN1),
TURN0 = (set0 → TURN1 | unset1 → TURN0 | isunset0 → TURN0).



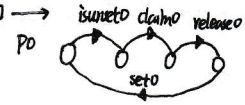
unset1은 3개로 isunset0
P0 참고

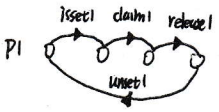
[Modelling one of the threads]

Model the wait loop simply with a transition on the appropriate is... event.
Add claim and release events so we can follow what is happening to the critical section:

P0 = (isunset0 → claim0 → release0 → set0 → P0).

※[LTS] →

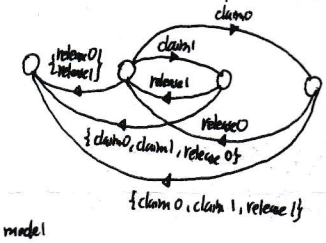




[Safety Property]

Ensure that only one thread holds the mutex at a time.
The permitted sequence of events is: MUTEX = (claim0 → release0 → MUTEX | claim1 → release1 → MUTEX)

→ FSP provides a shorthand for these processes with added error (-1) state.



{claim0, claim1, release1}

Full safety model:

TURN1 = (set0 → TURN1 | unset1 → TURN0 | isset1 → TURN1),
TURN0 = (set0 → TURN1 | unset1 → TURN0 | isunset0 → TURN0).
P0 = (isunset0 → claim0 → release0 → set0 → P0).
P1 = (isset1 → claim1 → release1 → unset1 → P1).

Property MUTEX = (claim0 → release0 → MUTEX | claim1 → release1 → MUTEX).

||SYS = (TURN1 || P0 || P1 || MUTEX). (No deadlocks/errors) So far so good.

[Liveness]

We have to check liveness. Our mutex should be willing to accept threads trying to claim it in any order. Thus it should avoid deadlock in a configuration where the external environment chooses the order in which threads acquire the mutex.

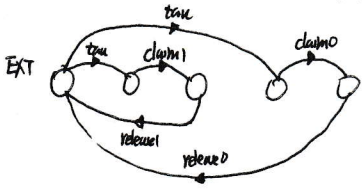
In FSP, this is achieved by hiding an environment which makes a hidden internal choice into a state which offers claim0 or a state which offers claim1.

△Liveness Test

EXT = (t → claim0 → release0 → EXT | t → claim1 → release1 → EXT) \{t}.

This introduces a new notation. The international choice arises because there are two different initial transitions on event t. The t event is then hidden with the \ operator.

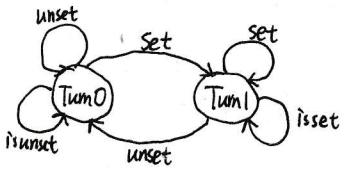
The overall effect is that the process makes an internal choice as to whether to engage in claim0 or claim1



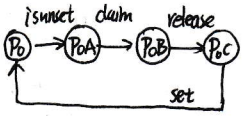
Trace to deadlock
tau
isset1

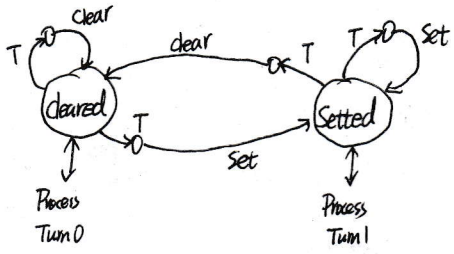
~~[illegible]~~

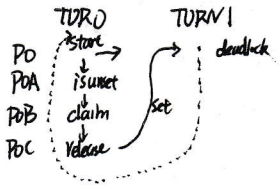
tau → claim0
isset1 → claim1



while (! TURN==0)) ;
// wait until turn equals 0







We first have to guess an invariant. In our case, where we are working our way "up" a sum, a good choice is likely to come from replacing the "n" in the final assertion with "i", so our new program, with variant, would be

```
while (i<n) {
    i=i+1 ;
    result = result + i*i;
    assert (result == i*(i+1)(2*i+1)/6 ); }
```

i=0

[Base case]

```
unsigned int i =0 ;
unsigned int result =0;
if (i<n ) {
    i=i+ 1 ;
    result = result +1;
    assert (result == i*(i+1)/2))
    assert (i<=n) ;}
else {
    assert (result ==n*(n+1)/2 );}
```

[Inductive step] $1 \le i < n$

```
unsigned int i;
unsigned int result;
assume (result = i*(i+1)/2);
assume (i<=n)
if (i<n ) {
    i=i+1 ;
    result = result + i[redacted];
    assert (result == i*(i+1)/2 );
    assert (i <=n ;}
```

[Termination condition]

```
unsigned int i;
unsigned int result;
assume (result == i*(i+1)/2 );
assume (i<= n);
if (! (i<n)) {
    assert (result == n*(n+1)/2 );}
```

① Write down the program with the invariants and preconditions

```
#include <limits.h>
....
unsigned int i=0 ;
unsigned int result =0;
⇒ assume (n<= LIMIT ) // precondition
where (i<n) {
    i=i+1 ;
    result = result + i*i ;
⇒   assert ((result= i*(i+2)*(2*i+1)/6) & (i<=n)); }
assert (result ==n*(n+1)*(2*n+1)/6 );
return result ; }
```

② Go through the three parts of the inductive proof by hand

1) precondition

$i=0$ ⇓ $i=i+1=1$

$1 = \frac{1*2}{2} = 1$ $\quad 1 = \frac{i*(i+1)}{2} = \frac{1\times 2}{2} = 1$

2) ~~invariant & (i<n)~~ b: invariant & (i<n) ; Loop_Body ⇒ invariant

'b' requires you to do the inductive step of the ordinary proof of the formula for sums of squares, i.e.

$$\frac{i*(i+1)*(2*i+1)}{6} + (i+1)(i+1) = \frac{(i+1)*(i+2)*(2*(i+1)+1)}{2}$$

lc' invariant & !(i<n) => postcondition.

invariant & !(i<n) = result == $i*(i+1)*(2i+1)/6$ & (i<n) & !(i<n)

$$= \text{result} == \frac{i*(i+1)*(2i+1)}{6} \;\&\; i==n$$

$$\Rightarrow \text{result} == \frac{n*(n+1)*(2n+1)}{6}$$

= postcondition