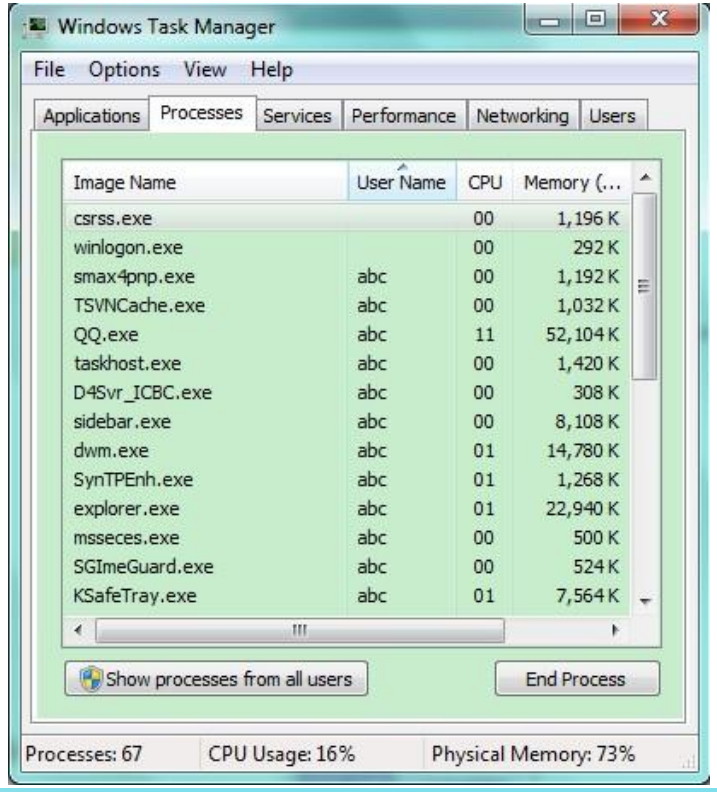
Critical region

- Critical region
- Semaphore
- Locking

- Critical region

With context switch

- different processes can coexist



IPC

- Rather than running separately, processes may cooperate with each other.
  - Xunlei, chrome
- Inter-Process Communication
  - extension: network, database transaction
- Some resources may be shared
  - memory, files

New problem

- summing from 1 to 100

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

temp	???

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100

sum	0

temp	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100

sum	0

temp	1

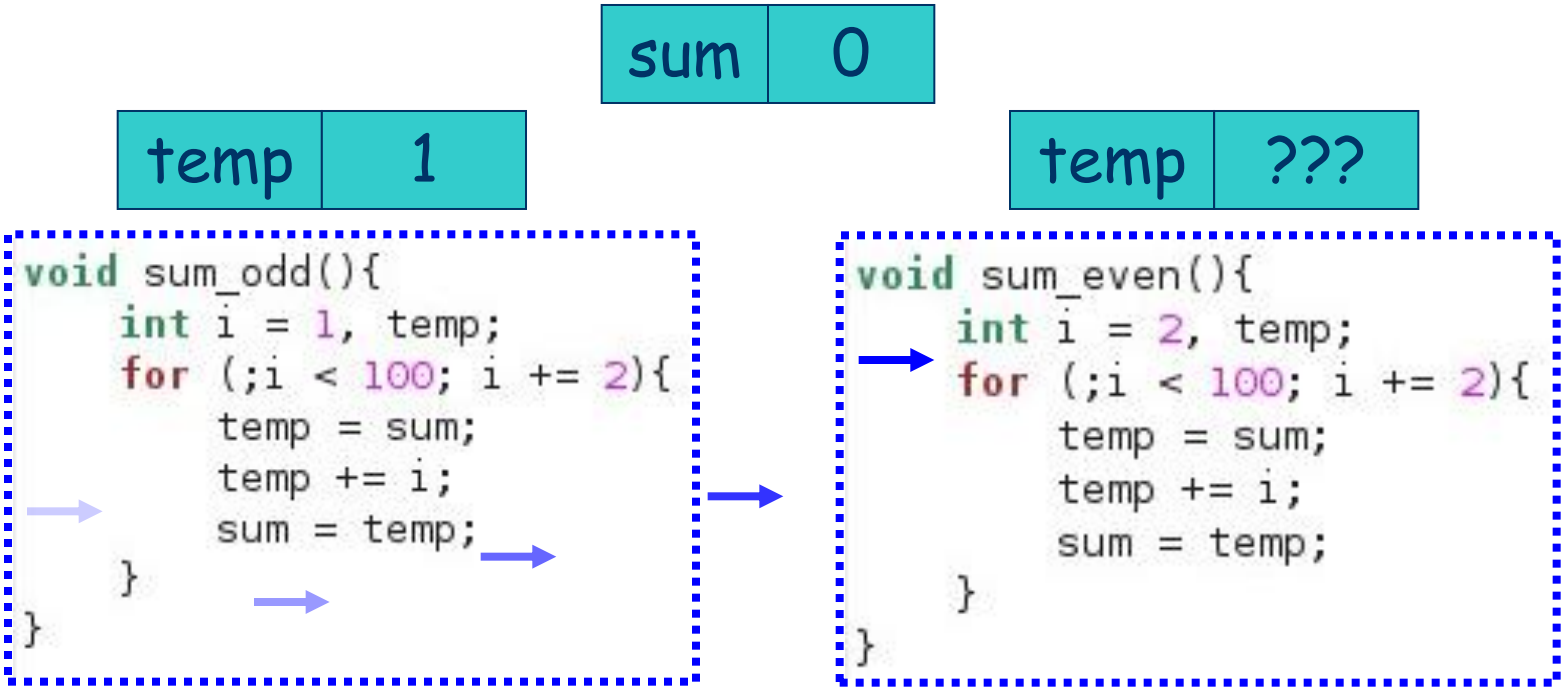
temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100



New problem

- summing from 1 to 100

sum	0

temp	1

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100

sum	0

temp	1

temp	0

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100

sum	0

temp	1

temp	2

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

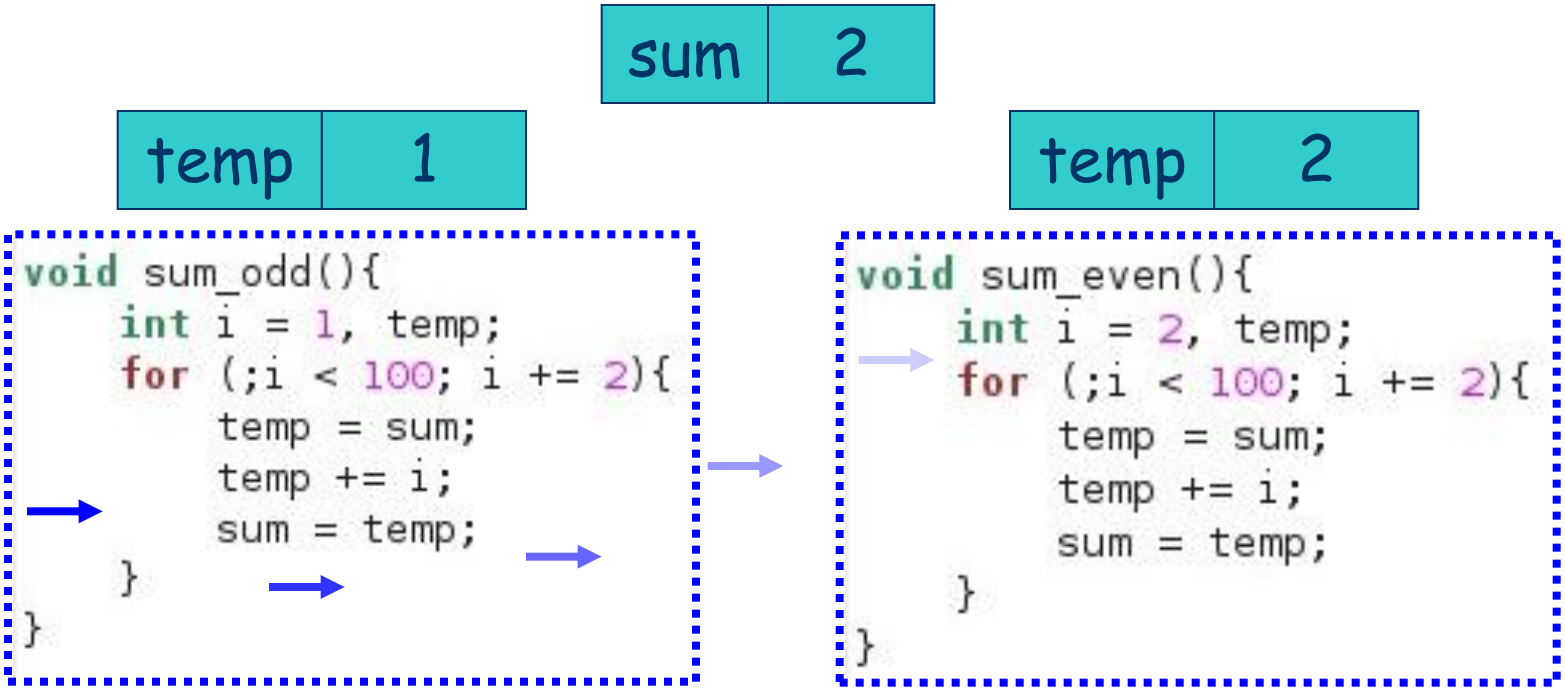
- summing from 1 to 100


```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

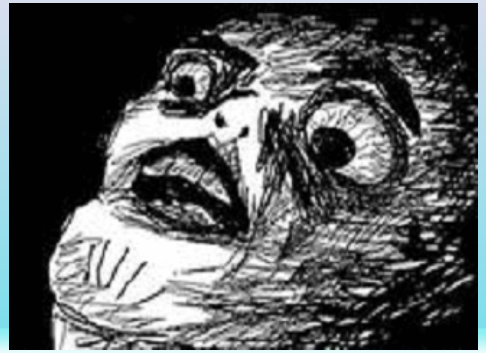
```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

New problem

- summing from 1 to 100



New problem



- summing from 1 to 100

lost update

sum	1

temp	1

temp	2

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        temp = sum;
        temp += i;
        sum = temp;
    }
}
```

Why?

- allow concurrent access
  - shared by two or more processes
- allow update
  - allow write operation
- not atomic
  - remain inconsistent while interrupted
- without concurrency control
  - access critical regions casually

Some mechanisms are needed

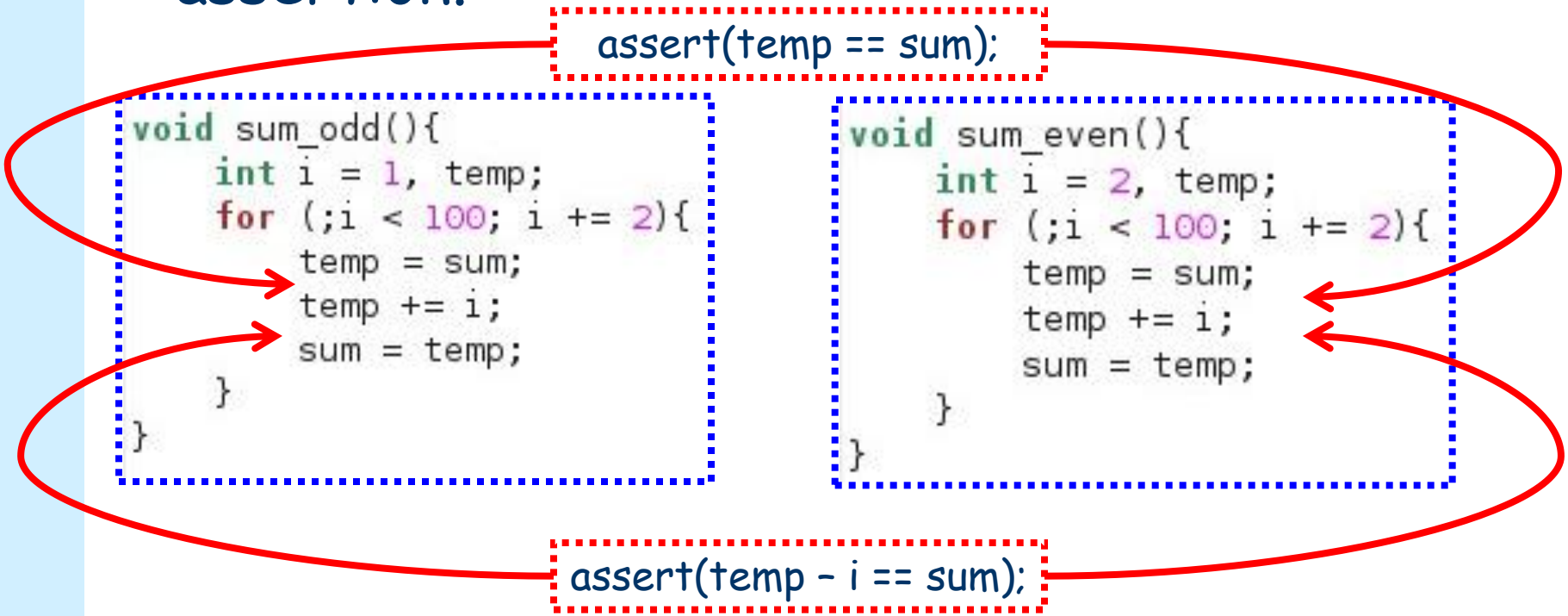
- allow concurrent access?
  - back to uni-tasking? ✗
- allow update?
  - read only? ✗
- not atomic?
  - make the process uninterruptable
- without concurrency control?
  - control the actions of accessing critical regions

Atomicity & Consistency

- atomicity
  - "all or nothing"
  - when seen by others, it performs either all operations or no operations in the critical region
- consistency
  - this is actually what correctness means
  - induction on the consistency of states
    - consistency ➔ inconsistency ➔ consistency
  - inconsistent states should not be exposed to others
- no atomicity ➔ no consistency

Consistency & Assertion

- Consistency can be used as the condition of assertion.



Another example

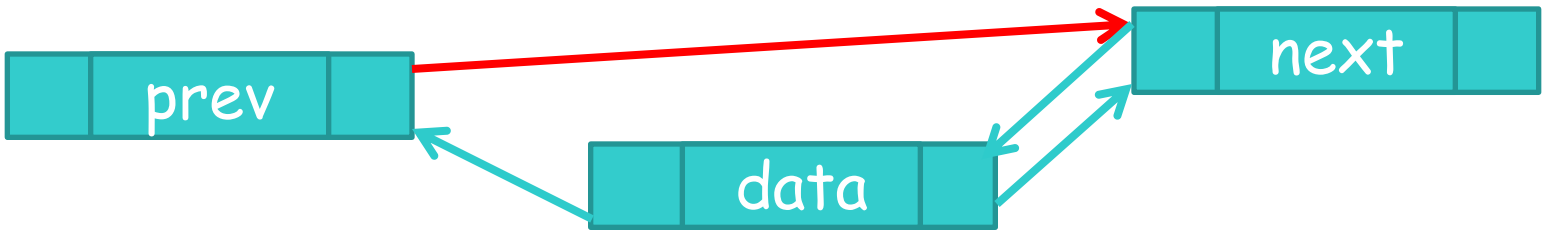
```
void list_del(ListHead *data) {
    ListHead *prev = data->prev;
    ListHead *next = data->next;
    assert(prev == NULL || prev->next == data);
    assert(next == NULL || next->prev == data);
    if (prev != NULL) prev->next = next;
    if (next != NULL) next->prev = prev;
}
```

check consistency

Another example

```
void list_del(ListHead *data) {
    ListHead *prev = data->prev;
    ListHead *next = data->next;
    assert(prev == NULL || prev->next == data);
    assert(next == NULL || next->prev == data);
    if (prev != NULL) prev->next = next;
    if (next != NULL) next->prev = prev;
}
```

check consistency

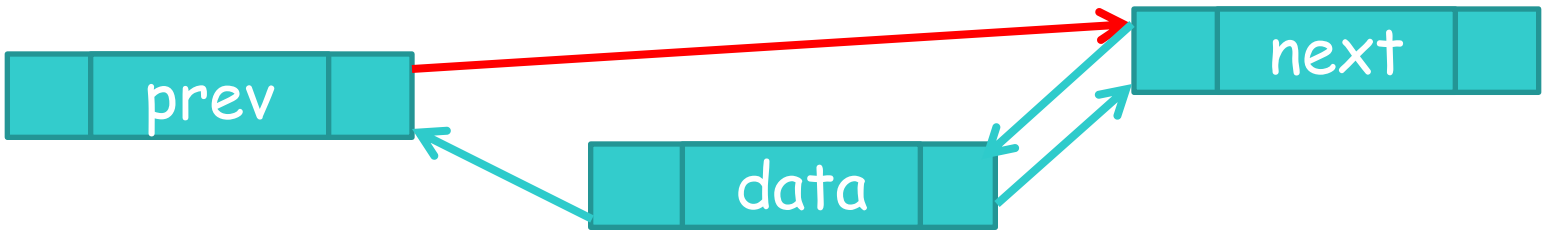


Another example

```
void list_del(ListHead *data) {
    ListHead *prev = data->prev;
    ListHead *next = data->next;
    assert(prev == NULL || prev->next == data);
    assert(next == NULL || next->prev == data);
    if (prev != NULL) prev->next = next;
    if (next != NULL) next->prev = prev;
}
```

check consistency

context switch happens

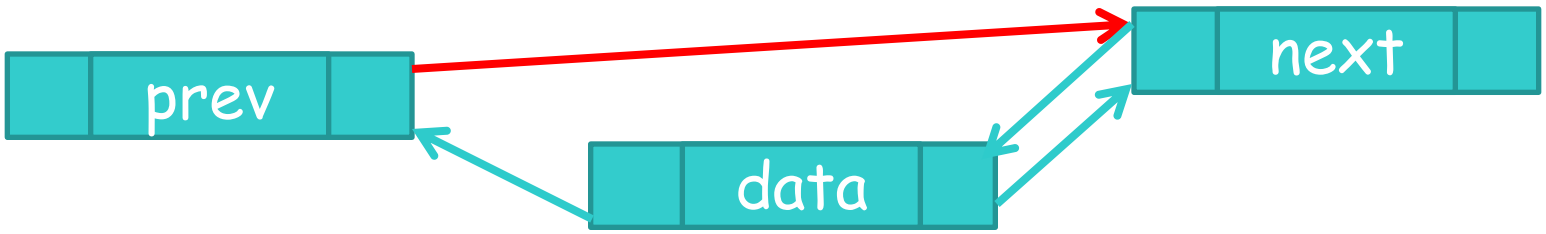


Another example

```
void list_del(ListHead *data) {
    ListHead *prev = data->prev;
    ListHead *next = data->next;
    assert(prev == NULL || prev->next == data);
    assert(next == NULL || next->prev == data);
    if (prev != NULL) prev->next = next;
    if (next != NULL) next->prev = prev;
}
```

check consistency

context switch happens



Protect critial regions

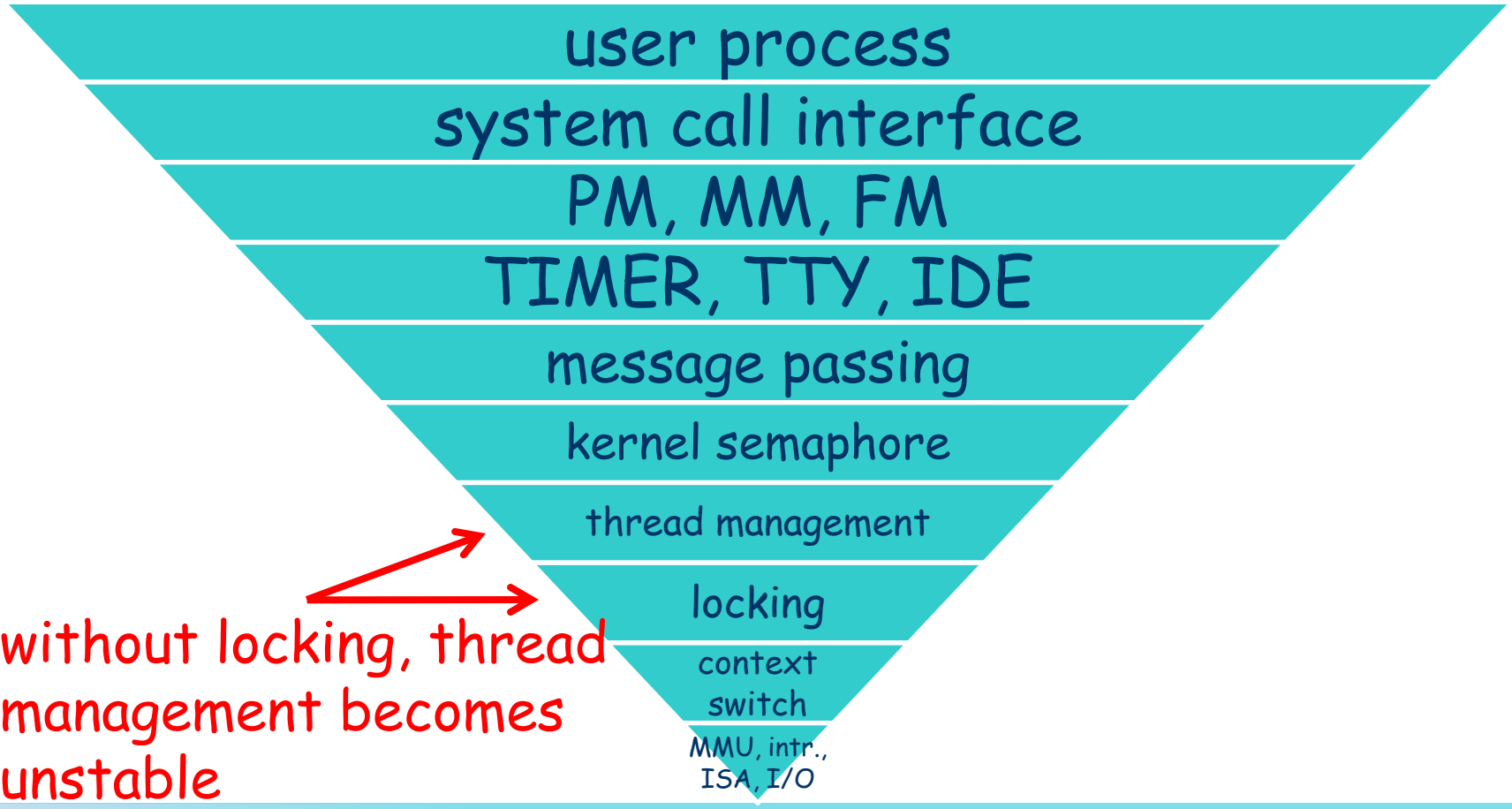
- concurrency control
  - goal: guarantee atomicity & consistency
- Without protection, your code will get stuck in mysterious behaviors.
  - nothing is impossible
  - your program will become an awful mess!
  - VERY HARD to backtrack, as well as reproduce

An awful mess

```
Hello, OS World!
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdab
cdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdab
cdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdab
cdabcdabcdabcdabcdabcdabcdabccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
Hello, OS World!
abcdaHello, OS World!
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdab
cdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdab
cdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

The architecture of Nanos



linux发行版不支持win8快速启动功能会导致数据丢失

- the “most” largest critical region

Windows 8的新功能Fast Startup能让双系统启动时更快，然而其负面作用是，当用户加载第二款系统（通常是Linux）时本地存储的数据处于风险之中。原因很简单，**在双启动配置下，Fast Startup选项并不会完全关闭Windows 8，只是强制它进入休眠模式**，从而能在用户退出另一个系统重新启动Windows 8时获得更快的加载速度。

由于是在休眠状态下，**Windows 8会继续存储Windows会话信息，包括FAT和NTFS分区文件系统数据**。当用户重新切换回Windows 8时，一些文件已经被删去或重写了。

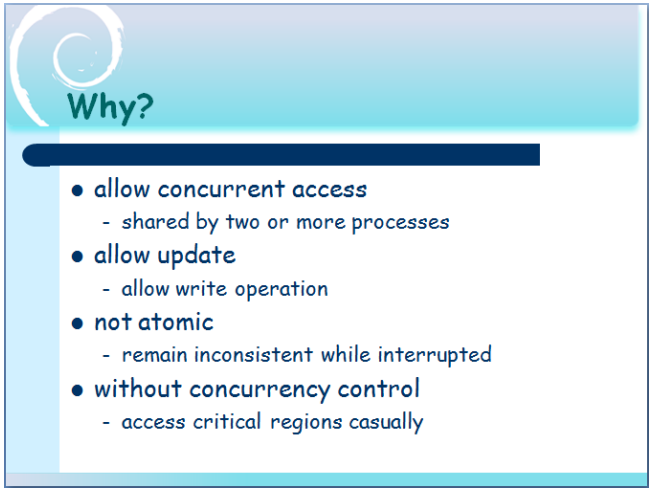
ntfs-3g FUSE文件系统驱动开发人员发现，Linux会试图在Windows分区写入数据，有时会重写存储在menmory image中的系统文件，**当切换回Windows 8时，Fast Startup只重新加载了系统，却不能重新找回丢失的数据**。

为了解决这个问题，ntfs-3g FUSE开发人员不得不开发了新版，让Linux系统将NTFS分区设定为只读。

但是，大多数Linux发行版，包括Ubuntu、Debian和openSUSE等目前都没有解决这个问题，所以**强制阻止操作系统重写Fast Startup数据是当前最好的解决办法**。

How to discover critial regions?

- (1) the same memory location may access by more than one process
  - usually global variables, or fixed pointers
- (2) one of them may be write operation
- (1) + (2) = data race



More data races

- process ➔ execution flow
  - process1 v.s. process2
  - process v.s. interrupt
  - process v.s. signal
  - interrupt1 v.s. interrupt2
  - interrupt1 v.s. itself
    - re-enterable & unre-enterable
- more tricky in SMP
  - "real" simultaneousness

Simplication in Nanos

- All hardware interrupts are unre-enterable.
- We do not need to perform extra protections for interrupt procedures.
  - Do NOT enable interrupt during the interrupt procedures.
  - Or your system may get stuck in mysterious behaviors.

Protect critical regions (cont.)

- Do not let two processes (execution flows) enter the same critical region.

- based on sleeping
  - semaphore
- based on waiting
  - locking

- Semaphore

Semaphore

- busy-then-sleep

Semaphore (cont.)

- maintain the number of resource available
  - but not the specific resource
- token > 0
  - *token* units of resource are available
- token == 0
  - no resource remains
  - processes must queue for the resource

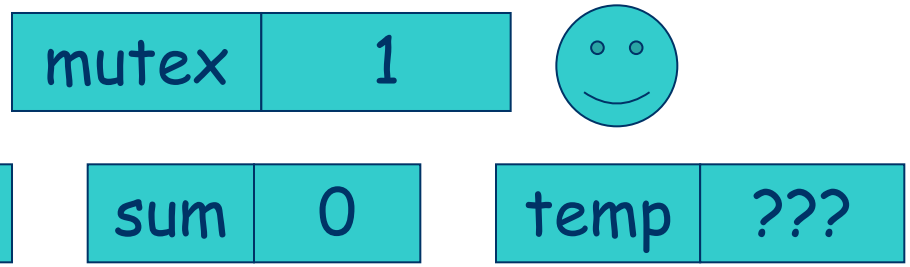
P-V operation

- manage the semaphore
- P operation
  - request for one unit of resource
  - block if no resource is available
- V operation
  - release one unit of resource
  - wake up a blocking process queuing for the same resource, if any

Usage

- counting semaphore
- binary semaphore
  - mutex lock
  - only one process can get the resource
    - when resource = critical region, the atomcity can be guaranteed
  - initialize with token = 1

Example - summing




```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	???

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i +=
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

get a token

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	0

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```


```
m_even(){
int i = 2, temp;
for (;i < 100; i += 2){
    P(mutex);
    temp = sum;
    temp += i;
    sum = temp;
    V(mutex);
}
}
```

Example - summing

mutex	0

temp	1

sum	0

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

sleeping

Example - summing

mutex	0

temp	1

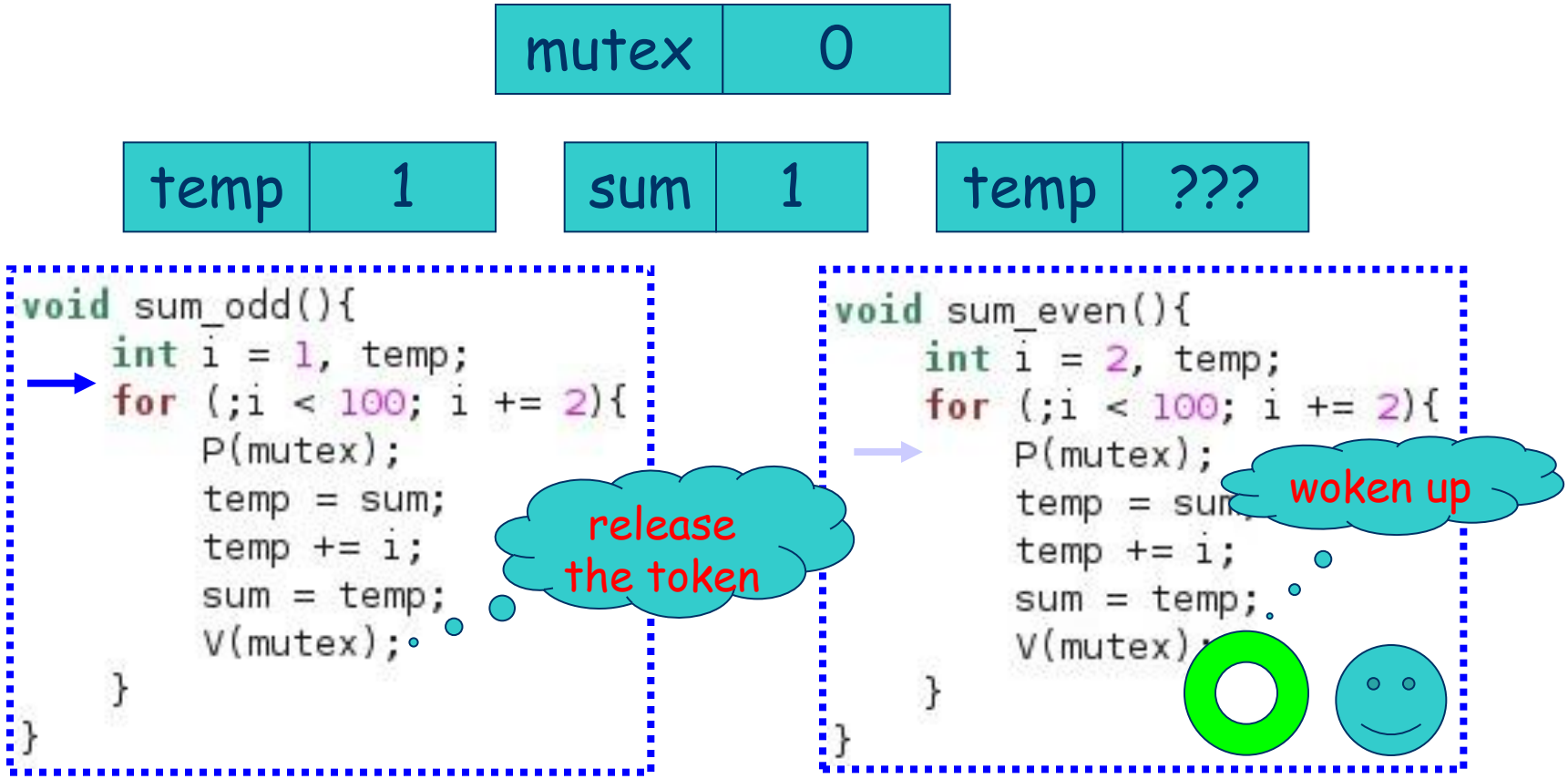
sum	1

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing



Example - summing

mutex	0

temp	1

sum	1

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	1

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	1

temp	???

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	1

temp	1

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	1

temp	3

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Example - summing

mutex	0

temp	1

sum	3

temp	3

```
void sum_odd(){
    int i = 1, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

```
void sum_even(){
    int i = 2, temp;
    for (;i < 100; i += 2){
        P(mutex);
        temp = sum;
        temp += i;
        sum = temp;
        V(mutex);
    }
}
```

Implementation

- data structure
  - token
  - queue
- operation
  - P() operation
    - if blocked, put the current process in the queue
  - V() operation
    - if some processes are blocking, wake up one of them
- It is easy to implement semphore with sleep() and wakeup().

Recursion?

- Different processes may access the same token by P-V operations concurrently.
  - P-V operations are also critical regions!
  - Atomicity & consistency must be guaranteed.
- How to solve this "recursion"?
  - locking!

Summary about semphore

- easy to understand
  - get: if no resource, then sleep
  - release: wake up one waiting process
- easy to use
  - P(), V()
- easy to implement
  - counter + thread management
- but the correstness of semphore depends on the correstness of locking

- Locking

Disable interrupt

- the easiest way to implement locking
- goal: disable context switch
  - context switch = interrupt driven stack switch
  - no context switch = no concurrent access
  - guarantee the atomicity of the execution in critical regions

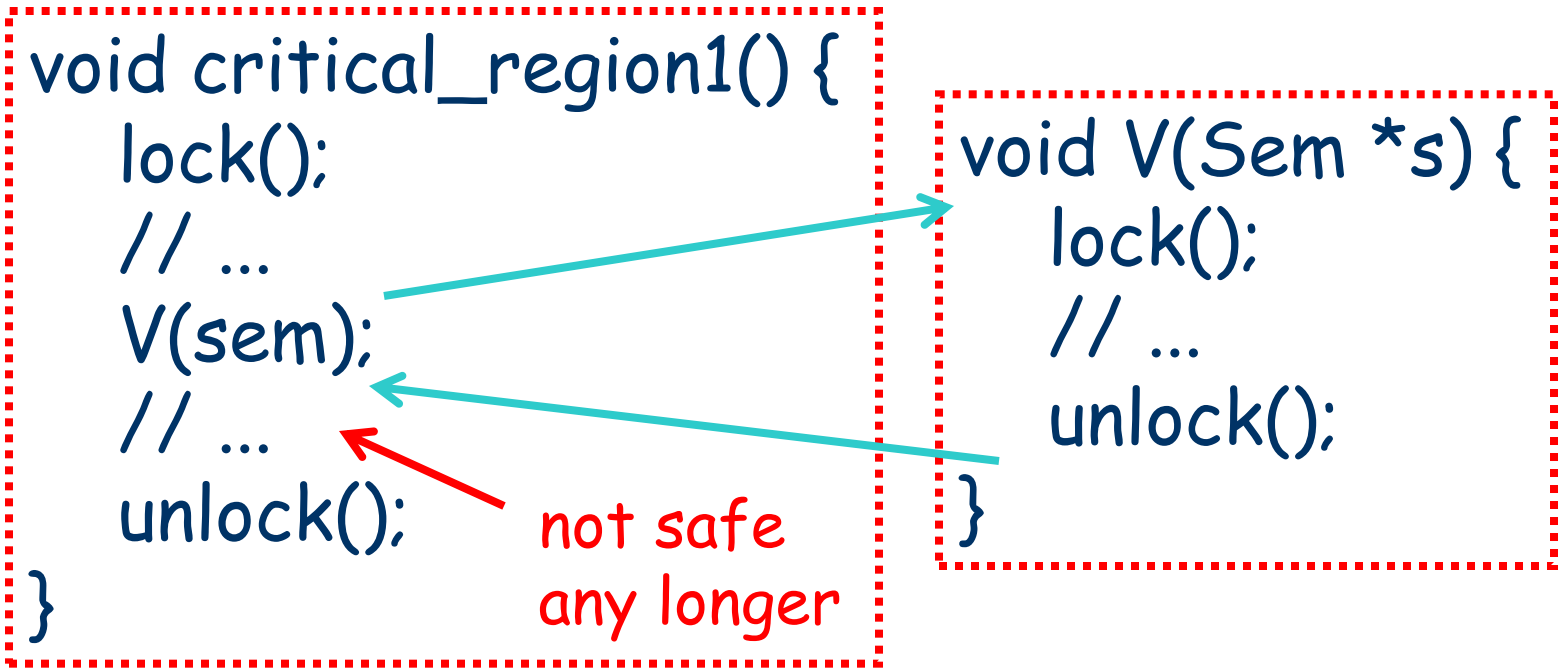
Shortcoming

- not suitable for long critical regions
  - reduce concurrency of the whole system
  - interrupts can not obtain response in time
- malicious user programs
  - do not enable interrupt any longer
- not work for SMP
  - every CPU has its own interrupt controller
    - local APIC
  - other CPUs can still enter the same region
  - atomicity & consistency are no guaranteed

Conclusions

- Disabling interrupt works well when
  - protecting short critial regions
  - in kernel space
  - in a unicore system
- Any problem?

Trap 1 – nested locking

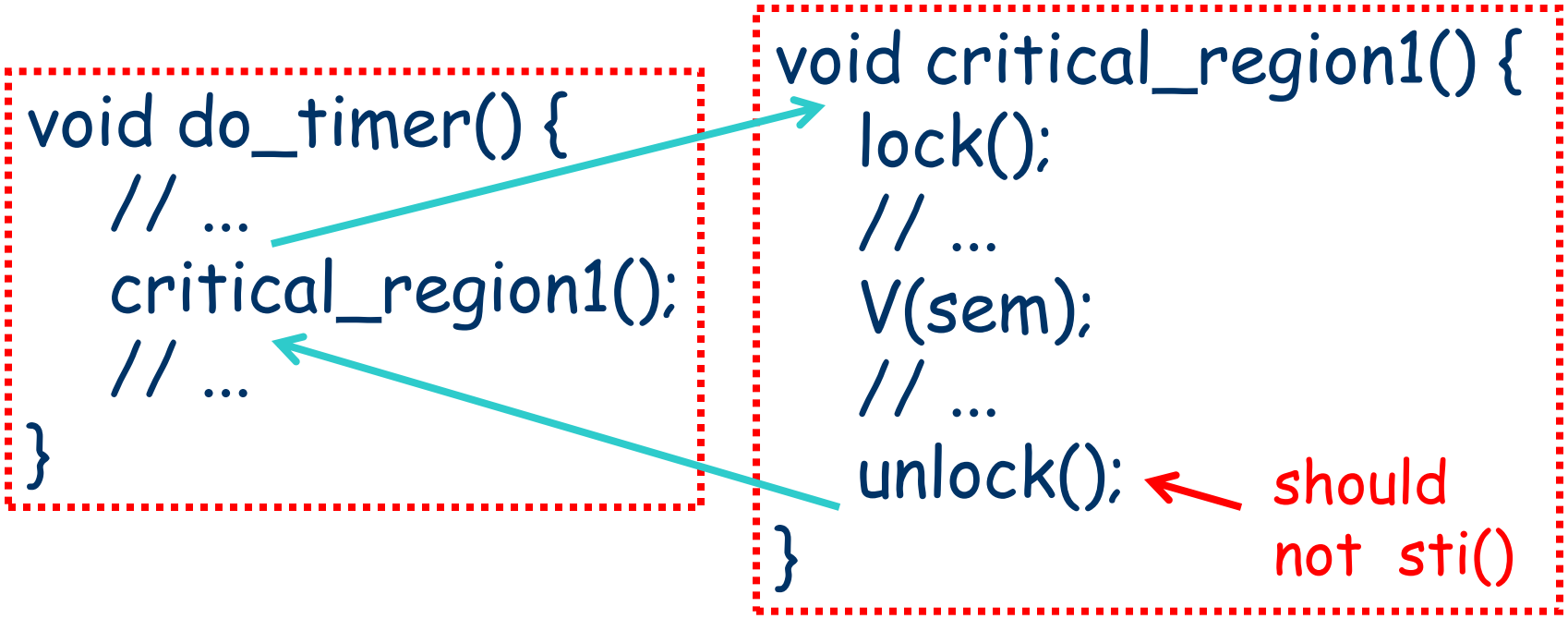


Trap 2 – sleep during locking

```
void P() {
    lock();
    // ...
    if(counter == 0)
        sleep();
    // ...
    unlock();
}
```

the consistency of locking should not be violated by other processes

Trap 3 – use locking in interrupts



Why it is hard to implement locking?

- We can not predict where we will use locking in the future.
  - make the implementation as flexible as possible
  - so does other designs
- Refactoring your code is another "solution".
  - but the cost is much higher
    - e.g. extra attentions should be payed, instead of using nested locking directly

How to verify your solution?

- try to find a common rule for locking
  - this may become a unified solution
- try to find consistency conditions
  - make them the conditions of assertions
- no assertion fail = no bad things happen
  - only bad things which may be captured by assertions you have written

How to verify your solution?

- try best to expose inconsistent states
  - use a higher timer interrupt frequency
  - more interrupts = more probability to trigger interrupt-related errors
- run test cases for a long time
  - one night is enough
  - How to detect mysterious reboot when you are not in front of your computer?

Have fun!

- Implementing locking is more "insteresting" than just listening.
- This is also a traning for software testing.
  - find assertion conditions
- Locking in SMP is more tricky.
  - We may mention this topic next week.