Chapter 6: Process Synchronization

肖 卿 俊

办公室：九龙湖校区计算机楼212室

电邮：csqjxiao@seu.edu.cn

主页： https://csqjxiao.github.io/PersonalPage

电话：025-52091022

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples

Background

- Concurrent access to shared data may result in data inconsistency.

- Let us recall the concept of race condition
  - Several processes (threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

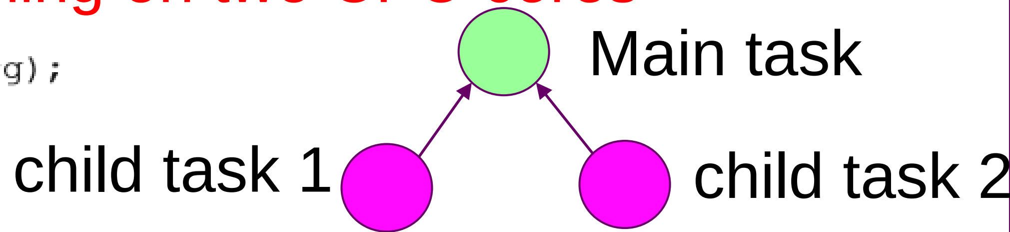
A Previously Used Example

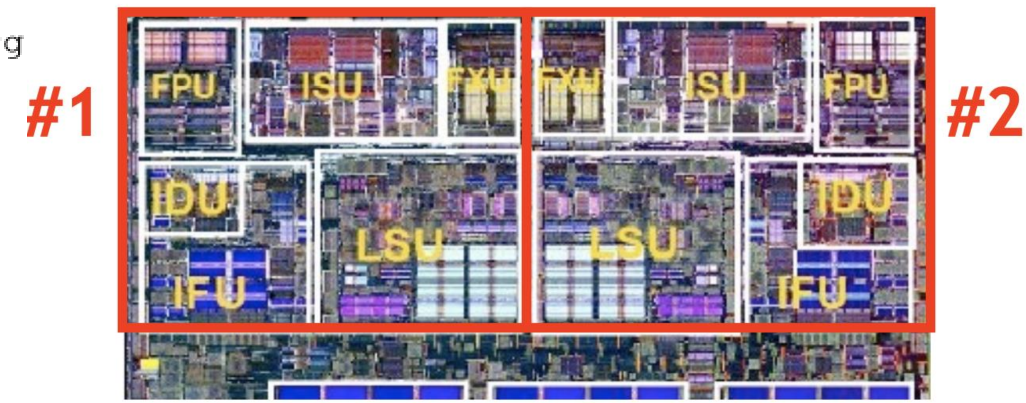
// volatile keyword forces the compiler to avoid caching the variable in CPU register. It always firstly read the data from memory.

```
volatile int counter = 0;
```

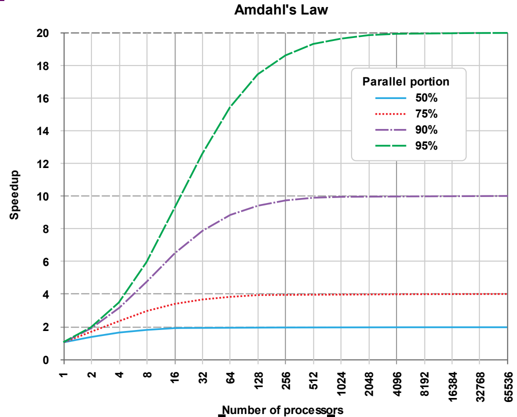
We split the for-loop across two threads running on two CPU cores

```
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg
    return NULL;
}


int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

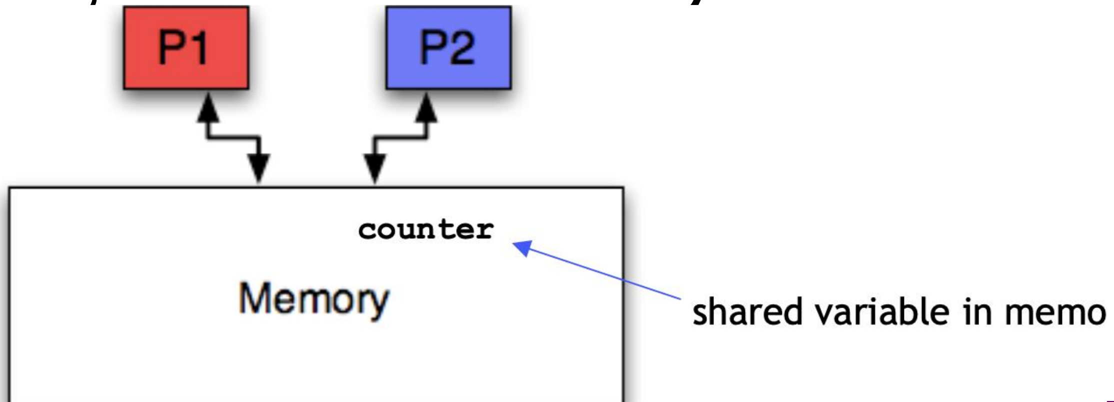




How much faster?

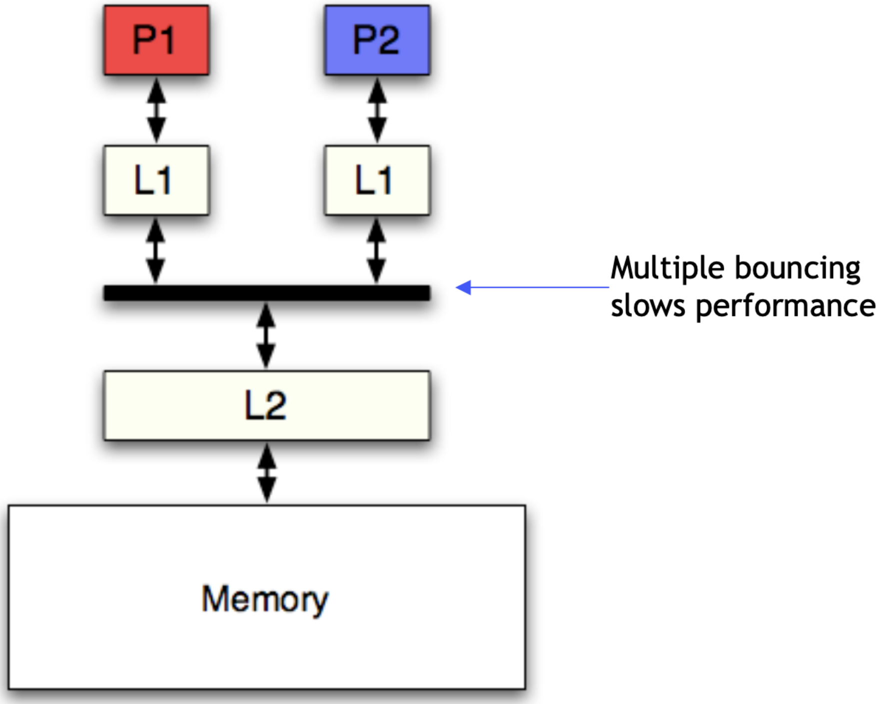


- We’re expecting a speedup of 2
- OK, perhaps a little less because of Amdahl’s Law (to predict the theoretical speedup when using multiple processors)
  - overhead for forking and joining multiple threads
- But it is actually slower!! Why??
- Here’s the mental picture that we have – two processors, shared memory



This mental picture is wrong!

- We've forgotten about caches!
  - The memory may be shared, but each processor has its own L1 cache
  - As each processor updates `counter`, it bounces between L1 caches



The code is not only slow, it's WRONG!

- Since the variable *counter* is shared, we can get a data race
- Increment operation: *counter*++ MIPS equivalent:

```
lw $t0, counter
addi $t0, $t0, 1
sw $t0, counter
```

- A data race occurs when data is accessed and manipulated by multiple processors, and the outcome depends on the sequence or timing of these events.

Sequence 1

Processor 1	Processor 2
`lw $t0, counter`	
`addi $t0, $t0, 1`	
`sw $t0, counter`	
	`lw $t0, counter`
	`addi $t0, $t0, 1`
	`sw $t0, counter`

`counter` increases by 2

Sequence 2

Processor 1	Processor 2
`lw $t0, counter`	
	`lw $t0, counter`
`addi $t0, $t0, 1`	
	`addi $t0, $t0, 1`
`sw $t0, counter`	
	`sw $t0, counter`

`counter` increases by 1 !!

Another Example: Revisit the Producer Consumer Problem

- Shared-memory solution to bounded-buffer problem (Chapter 3)
  - The code can only use N-1 items in the buffer

Producer:

```
while (1) {
  while (((in+1) % BUF_SIZE) == out) ;
  ……
  in = (in+1) % BUF_SIZE;
}
```

Consumer:

```
while (1) {
  while (in == out) ;
  ……
  out = (out+1) % BUF_SIZE;
}
```

We modify the above code by adding a variable *counter*, such that all items in the buffer can be used

Bounded-Buffer Solution

- Shared data

```
#define BUF_SIZE 10
class Item {
   . . .
 Item & operator=(const Item & ) { … }
}
Item buffer[BUF_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer Solution

- Producer process

```
Item nextProduced;

while (1) {
    while (counter == BUF_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUF_SIZE;
    counter++;
}
```

- Consumer process

```
Item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUF_SIZE;
    counter--;
}
```

Critical Shared Data

- Counter is a piece of critical shared data
- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Difficult to Implement the Atomic Guarantee

- However, the statement "count++" may be implemented in machine language as:
  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement "**count--**" may be implemented in machine language as:
  register2 = counter
  register2 = register2 – 1
  counter = register2

Potential Data Inconsistency

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Potential Data Inconsistency

- Assume **counter** is initially 5. One interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Producer

register1 = counter

register1 = register1 + 1

counter = register1

Consumer

register2 = counter

register2 = register2 – 1

counter = register2

Concept of Race Condition

- **Race condition** occurs, if:
  - **two or more processes/threads access and manipulate the same data concurrently, and**
  - **the outcome of the execution depends on the particular order in which the access takes place.**

- To prevent race conditions, concurrent processes must be **synchronized**.

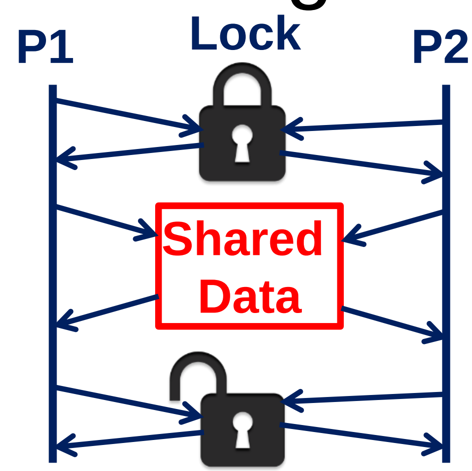
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples

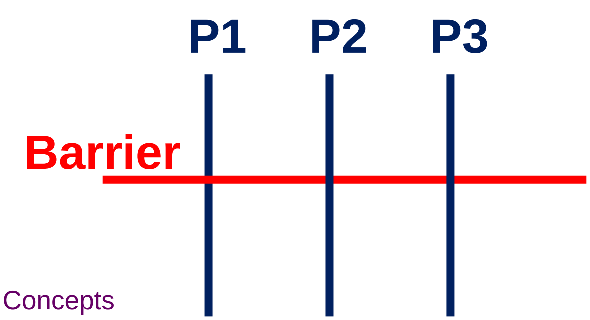
Three Typical Mechanisms of Process Synchronization

- Locks for shared memory programming
  - Exclusive Lock

  - Shared Lock:
    - ✓Readers can share a lock, but readers and writers must access the data mutually exclusively



- There are other synchronization primitives for shared memory programming, e.g., Barrier



OS Support to Implement an Exclusive Lock for Threads

- Using Mutex: Sleeping of threads Instead of spinning that wastes CPU cycles

- APIs of POSIX Thread for exclusive lock

```
int pthread_mutex_lock(pthread_mutex_t* mutex)
int pthread_mutex_unlock(pthread_mutex_t* mutex)
```

- An Example

Give a demonstration

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
counter = counter+1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The Critical-Section Problem: An Use Case of Exclusive Lock

- Multiple processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Critical Section and Mutual Exclusion

- Thus, the execution of critical sections must be *mutually exclusive* (e.g., at most one process can be in its critical section at any time).

- The *critical-section problem* is to design a protocol that processes can use to cooperate.

The Critical Section Protocol

```
do {
    entry section
    critical section
    exit section
} while (1);
```

- A critical section protocol consists of **two** parts: an ***entry section (or lock)*** and an ***exit section (or unlock)***.
- Between them is the critical section that must run in a **mutually exclusive** way.

Solution to Critical-Section Problem

- Any solution to the critical section problem must satisfy the following three conditions:
  - **Mutual Exclusion**
  - **Progress**
  - **Bounded Waiting**

- Moreover, the correctness of the solution cannot depend on relative speed of processes and scheduling policy.

Mutual Exclusion

- If a process P is executing in its critical section, then *no* other processes can be executing in their critical sections.
- The entry protocol should be capable of blocking processes that wish to enter but cannot.
- Moreover, when the process that is executing in its critical section exits, the entry protocol must be able to know this fact and allows a waiting process to enter.

Progress

- If no process is executing in its critical section and some processes wish to enter their critical sections, then
  - Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).
  - No other process can influence this decision.
  - This decision cannot be postponed indefinitely.

Bounded Waiting

- After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a *bound* on the number of times that other processes are allowed to enter.

- Hence, even though a process may be blocked by other waiting processes, it will not be waiting forever.
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the n processes

Solve the Problem without any OS Support

- Consider a simple case of only 2 processes, P_0 and P_1
- General structure of process P_i (and P_j)

do {   ***entry section***

critical section

exit section

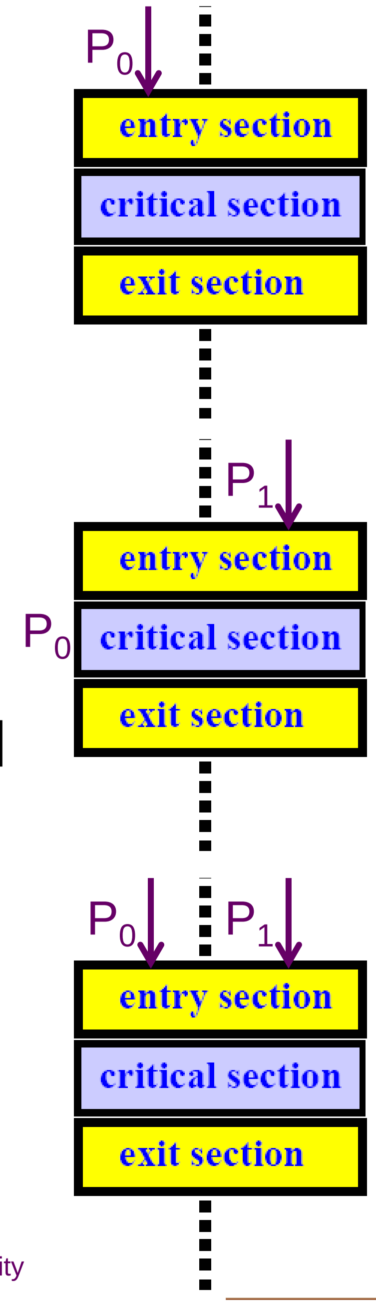
remainder section

} while (1);

- Processes may share some common variables to synchronize their actions.

Consider Three Test Cases

- Case 1: One process repeatedly attempts to enter the critical section (CS)
  - Progress Test: Whether P_0's repeated entering of CS is independent of P_1's attempt
- Case 2: One process is already in the critical section, and meanwhile the other process attempts to enter
  - Mutual Exclusive Test: P_0 safely block P_1 out
  - Progress Test: When P0 exists, P1 is notified
- Case 3: Two processes try to enter the critical section simultaneously
  - Progress Test: Whether it is possible for the two processes to block each other's entry
  - Mutual Exclusive Test: Whether it is possible for them to both enter the section



Our First Attempt: Algorithm 1

- Shared variables:
  - **boolean lock**; *// initially **lock = false***
  - **lock = true** ⇒ the critical section has been locked
- Process P_i :

  do { **while (lock)** ; *// if locked then wait*

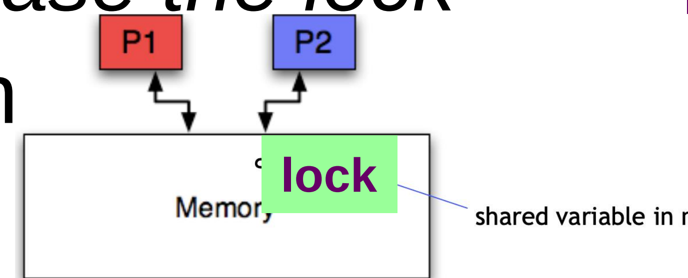
  lock = true; *// acquire the lock*

  critical section

  lock = false; *// release the lock*

  remainder section

  } **while (1)**;

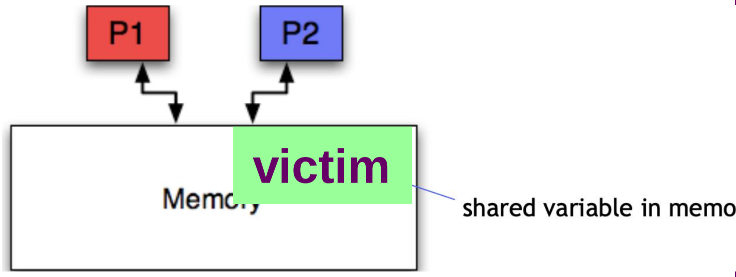


- Does not satisfy mutual exclusion. Why?

Our Second Attempt: Algorithm 2

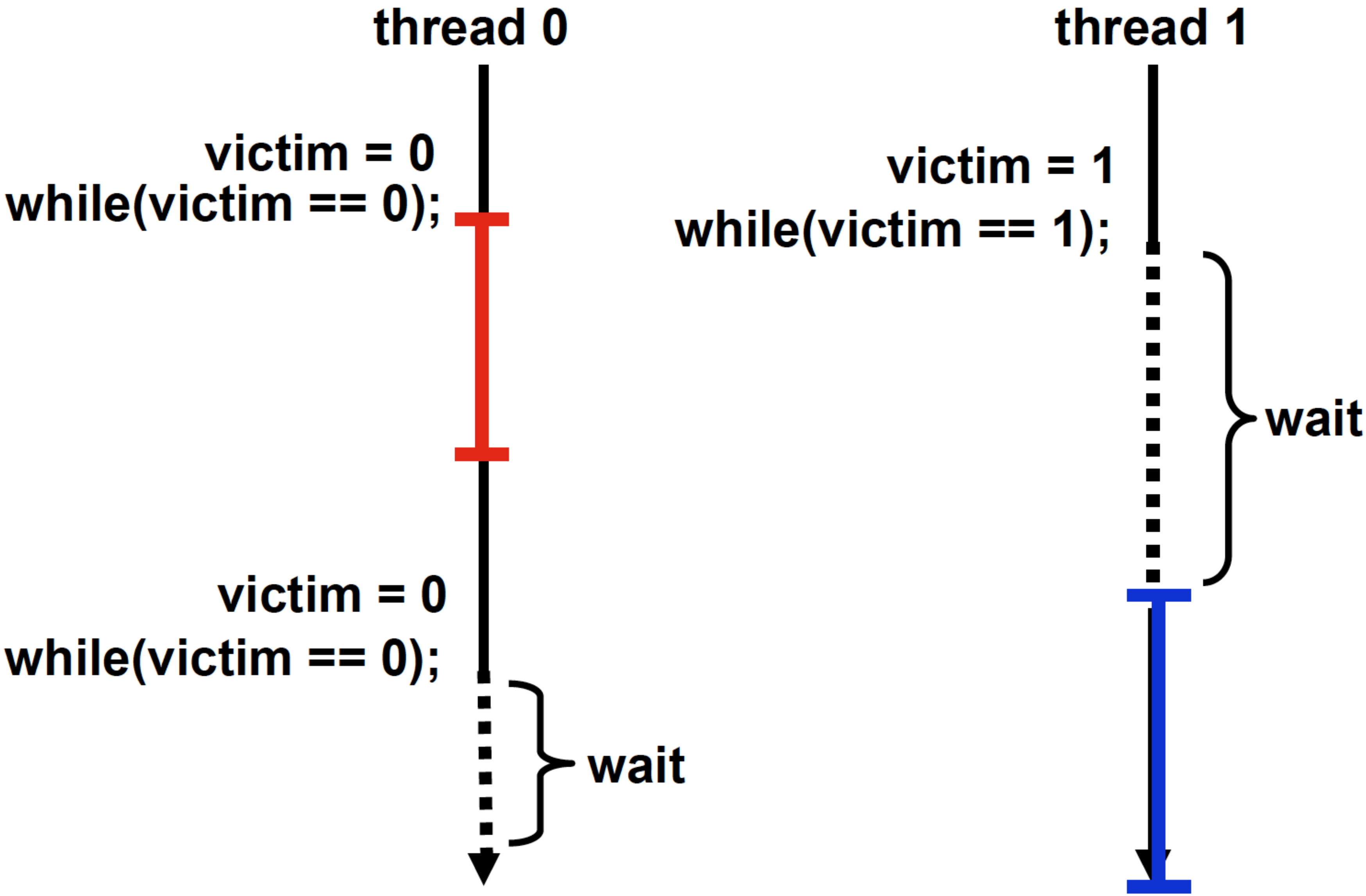
- Shared variables:
  - **int victim**; initially **victim = 0 (or victim = 1)**
- Process P_i :

```
do {victim = i;          // determine who is the victim
    while (victim == i) ; // if I am victim, then wait
    critical section  // assume empty
    // do nothing for CS exit
    remainder section
} while (1);
```

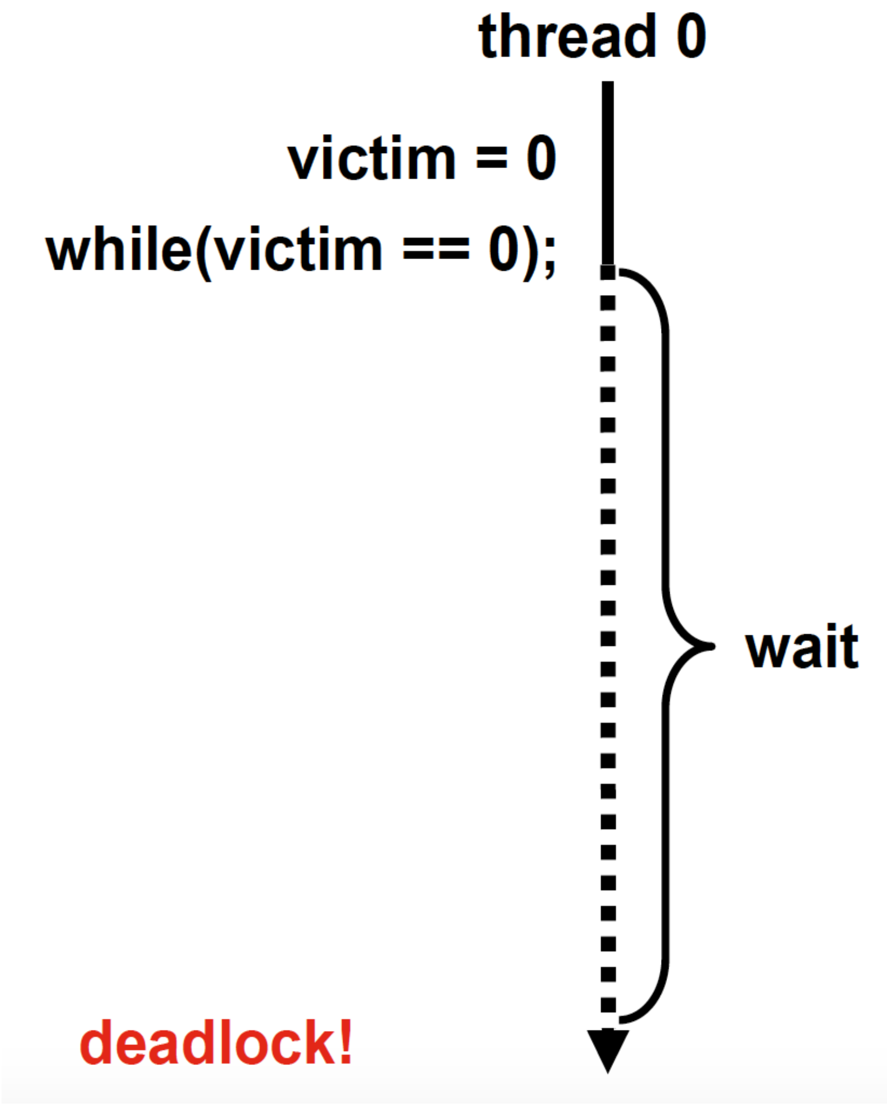


- Processes are forced to run in an alternating way
- Satisfies mutual exclusion, but not progress

Alternating and Atomic Execution of Algorithm 2



Deadlock of Algorithm 2



Another Failed Attempt: Algorithm 3

- Shared variables:
  - **boolean flag[2]**; *// initially **flag[0] = flag[1] = false***
  - **flag[i] = true** $\Rightarrow$ P_i *wants to enter its critical section*
- Process P_i

do {flag[i] = true; *// I want to enter*

    while (flag[1-i]) ; *// If you also want, then I wait*

    critical section

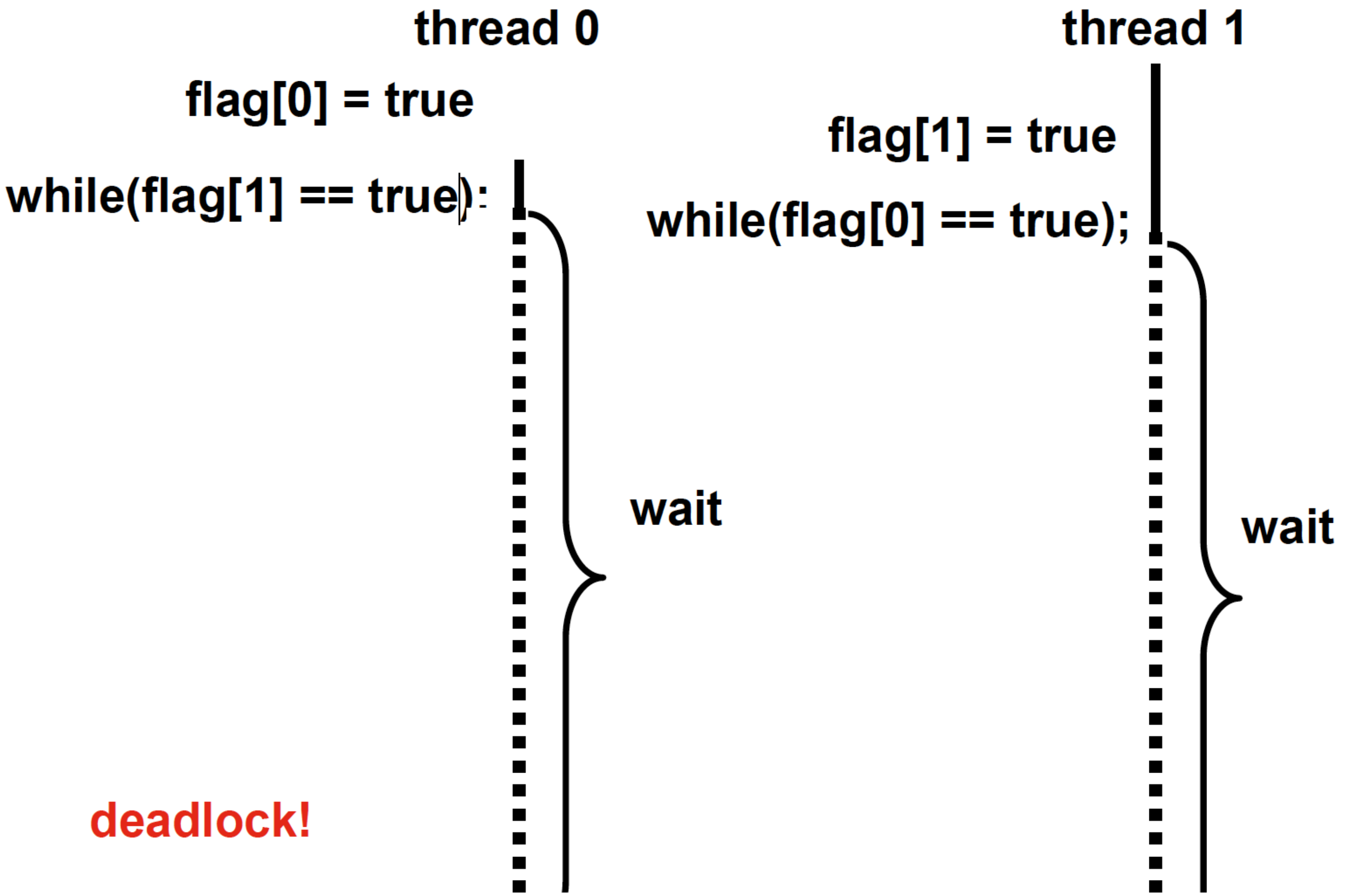
    flag[i] = false; *// I leave*

    remainder section

} while (1);

- Can satisfy mutual exclusion, but not progress requirement. Why?

Deadlock Problem of Algorithm 3



Is the Following Algorithm Correct?

- What if we change the location of the statement: **flag[i] = true**?
- Process P_i :

```
do {      while (flag[1-i]) ;
          flag[i] = true;
           critical section
          flag[i] = false;
            remainder section
 } while (1);
```

- Does not satisfy mutual exclusion

Comparison of Algorithms 1, 2, 3

Critical Section Algorithms	Test Case 1: P_0 serialized enter	Test Case 2: P_0, P_1 serialized enter	Test Case 3: P_0, P_1 concurrent enter
Algorithm 1 with a shared ***lock*** variable	√	√	× (ME)
Algorithm 2 with a shared ***victim*** variable	× (Progress)	√	√
Algorithm 3 with two shared ***flag[2]*** variables	√	√	× (Progress)
Peterson's Algorithm, with a shared ***victim*** variable and two shared ***flag[2]*** variables	√	√	√

Combine the advantages of Algorithms 2 and 3

Peterson's Algorithm Give a demonstration

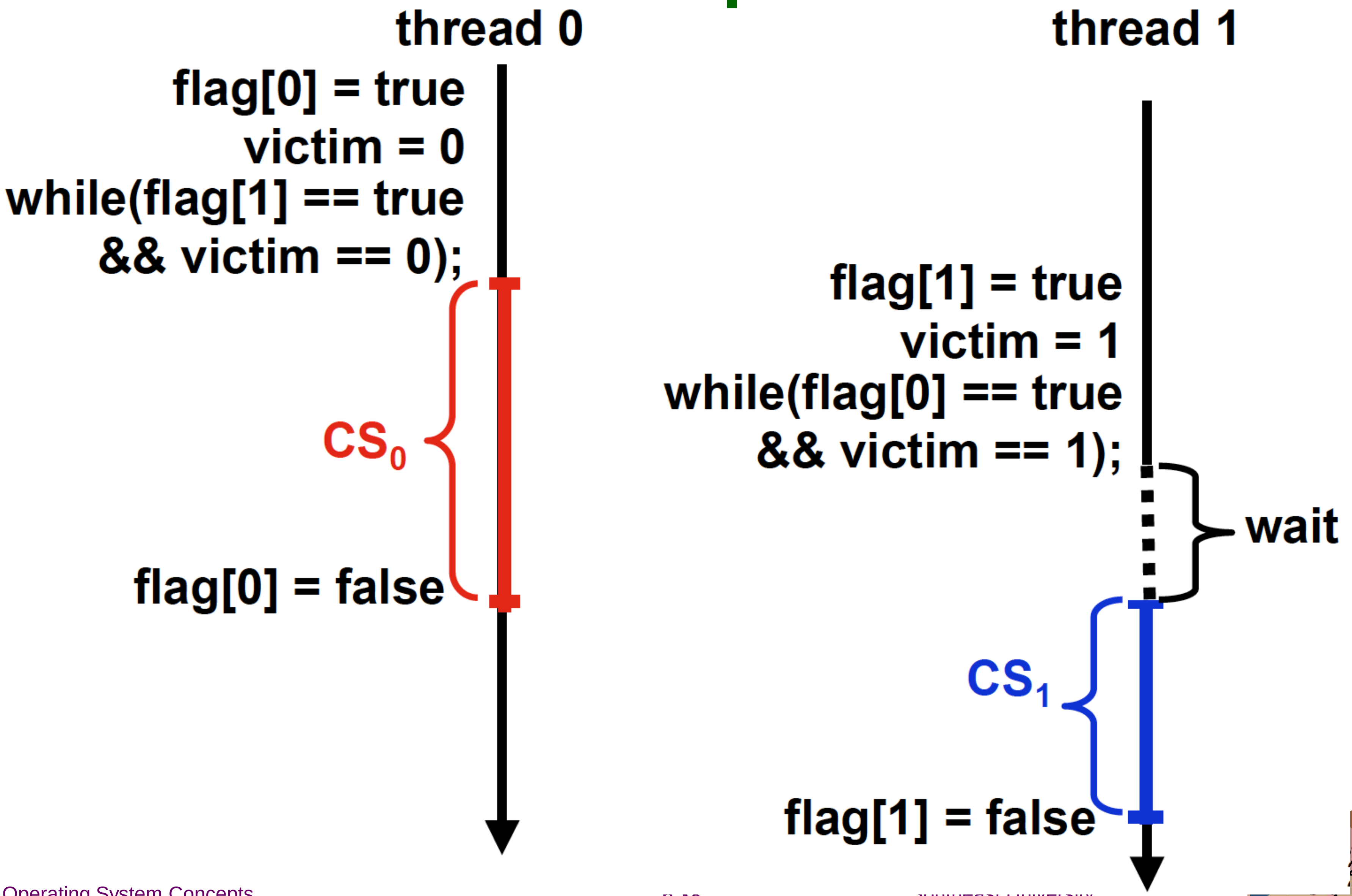
- Combined shared variables of algorithms 2, 3.
- Process P_i

```
do {      flag[i] = true;    // I'm interested
          victim = i;        // you go first
          while (flag[1-i] and victim == i) ;
             critical section
          flag[i] = false; // I'm not interested
                           // any more
             remainder section
} while (1);
```

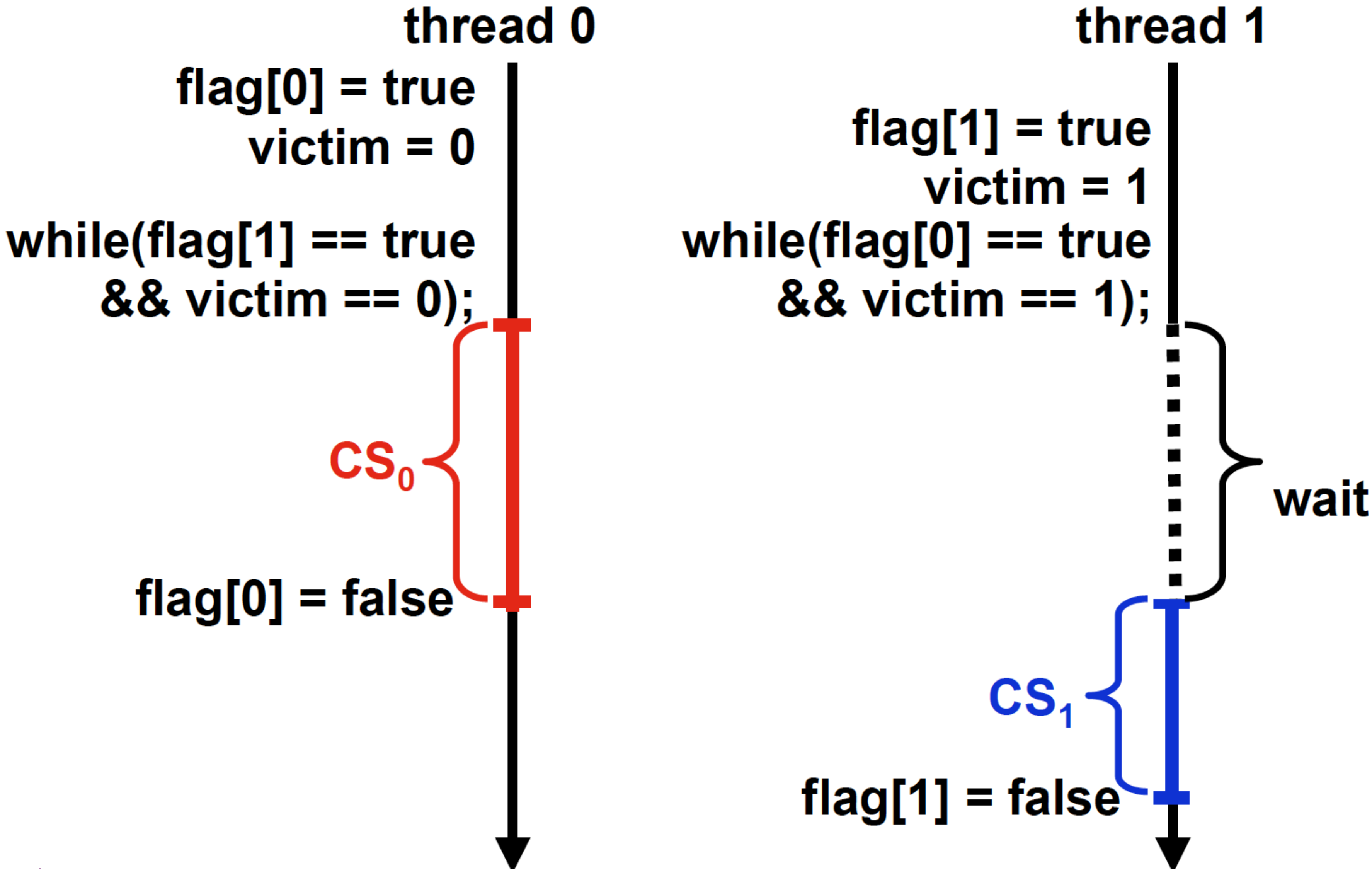
Gary Peterson. Myths about the Mutual Exclusion Problem. Information Processing Letters, 12(3):115-116, 1981.

- Meet all the three requirements; Can solve the critical-section problem for two processes.

Peterson's Lock: Serialized Acquires

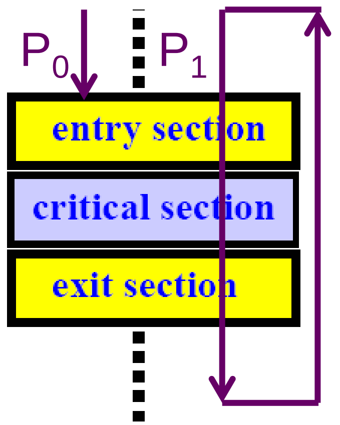


Peterson's Lock: Concurrent Acquires



Test the Bounded Waiting Property

- **Recall:** After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a *bound* on the number of times that other processes are allowed to enter.
- Test Case: Two processes attempt to enter the critical section simultaneously
  - Assume P0 is fast, while P1 is slow
  - Can P0 repeatedly grab the exclusive lock, causing P1 to starve?
    - ✓ If yes/no, the solution of critical section cannot/can satisfy bounded waiting property



Proof of Peterson's Algorithm

- The mutual exclusion requirement is assured.
- The progress requirement is assured. The victim variable is only considered when both processes are using, or trying to use, the resource.
- Deadlock is not possible. If both processes are testing the while condition, one of them must be the victim. The other process will proceed.
- Finally, bounded waiting is assured. When a process that has exited the CS reenters, it will mark itself as the victim. If the other process is already waiting, it will be the next to proceed.

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Quiz: Is the following code correct?

- What if we change **victim = i** to **victim = 1-i**?

```
do {        flag[i] = true;   // I'm interested
            victim = 1-i;        // I go first
            while (flag[1-i] and victim == i) ;
                    critical section
            flag[i] = false; // I'm not interested
                    remainder section
} while (1);
```

  - Can the code satisfy mutual exclusion?
  - Can the code satisfy progress?
  - Can the code satisfy bounded waiting?

Quiz: Is the following code correct?

- What if we change **victim = i** to **victim = 1-i**?

do { **flag[i] = true;** // I’m interested

victim = 1-i; // I go first

while (flag[1-i] and victim == i) ;

critical section

flag[i] = false; // I’m not interested

remainder section

} **while (1);**

- Can the code satisfy mutual exclusion? NO
- Can the code satisfy progress? YES
- Can the code satisfy bounded waiting? NO

Memory Fence

- Give a C-code demo of Peterson's algorithm
- A memory barrier, also known as a membar, memory fence or fence instruction, is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. https://en.wikipedia.org/wiki/Memory_barrier
- Operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

https://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Atomic-Builtins.html

Guarantee Memory Access Ordering

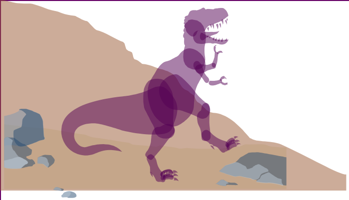
- Insert full memory barrier at multiple points

```
do {        flag[i] = true;    // I'm interested
__sync_synchronize(); // full memory barrier
            victim = i;        // You go first
            while (flag[j] and victim == i) ;
__sync_synchronize(); // full memory barrier
                critical section
__sync_synchronize(); // full memory barrier
            flag[i] = false; // I'm not interested
                    remainder section
} while (1);
```

Lamport's Bakery Algorithm

Solve the critical section problem for an arbitrary number of processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...



Bakery Algorithm

- Notation
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots, n - 1$

- Shared data

boolean choosing[n];

int number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {
      choosing[i] = true;
      number[i] = max(number[0], number[1], …,
                                  number[n – 1]) + 1;
      choosing[i] = false;
      for (j = 0; j < n; j++) {
         while (choosing[j]) ;
         while ( (number[j] != 0) &&
                  ((number[j], j) <  (number[i], i)) ) ;
      }
          critical section
      number[i] = 0;
          remainder section
} while (1);
```

Get a ticket first

Which parts are the entry and exit sections?

What is the use of choosing[]?

What if we remove the choosing[]?

```
do {
        number[i] = max(number[0], number[1], …,
                                   number[n – 1]) + 1;
        for (j = 0; j < n; j++) {
           while ( (number[j] != 0) &&
                  ((number[j], j) <  (number[i], i)) ) ;
        }
            critical section
        number[i] = 0;
            remainder section
} while (1);
```

Can the above code satisfy the mutual exclusion and why?

Hint: two processes computed the same number[i].

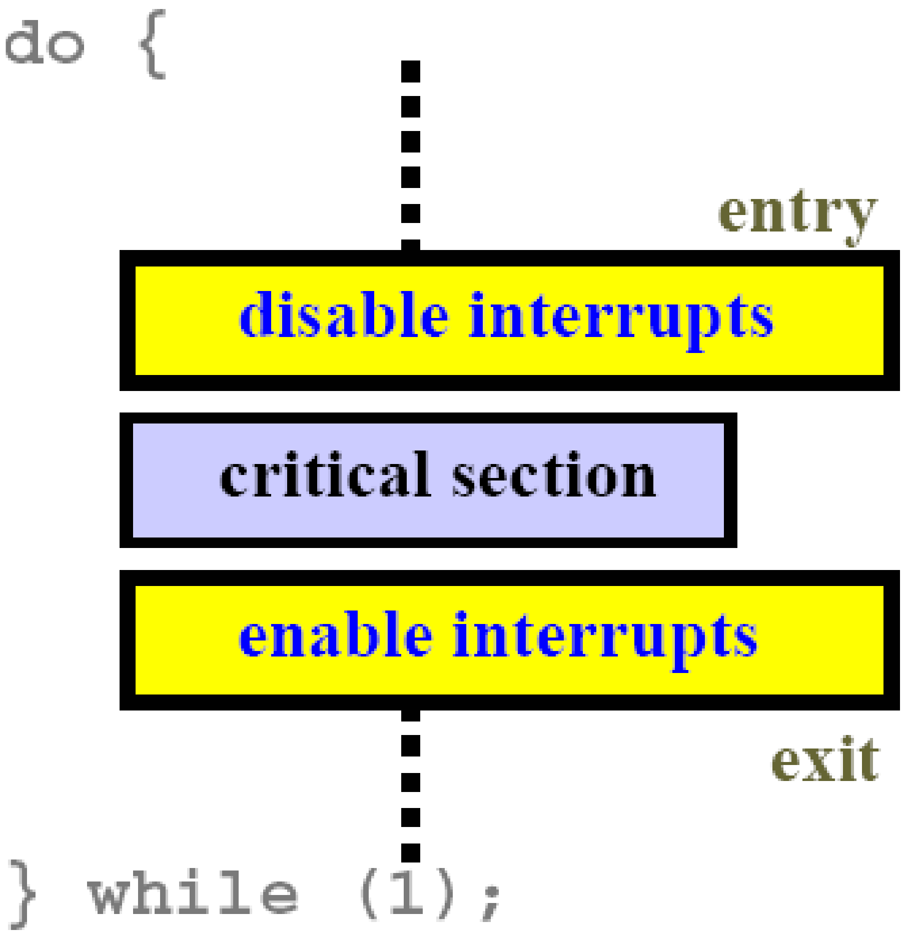
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware to Build a Lock
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples

Hardware Support

- There are two types of hardware synchronization supports:
  - Disabling/Enabling interrupts: This is slow and difficult to implement on multiprocessor systems.
  - Special machine instructions:
    - ✓ Test and set (TAS)
    - ✓ Swap
    - ✓ Atomic fetch-and-add

Interrupt Disabling



- Because interrupts are disabled, no context switch will occur in a critical section.
- Infeasible in a multiprocessor system because all CPUs must be informed.
- Some features that depend on interrupts (e.g., clock) may not work properly.

Test-and-Set (TAS)

- Test and modify the content of a machine word atomically

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
  while (TestAndSet(lock)) ;
    critical section
  lock = false;
    remainder section
} while(1);
```

Atomic Swap

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

  boolean lock = false;
- local variable

  boolean key;
- Process P_i or Interrupt Handler TH_i

```
do {
    key = true;
    while (key == true)  Swap(lock,key);
        critical section
    lock = false;
        remainder section
} while(1);
```

Cannot satisfy bounded waiting. Why?

Another Atomic CPU Instruction Fetch-and-add

- fetch-and-add instruction performs the operation

```
<< atomic >>
function FetchAndAdd(address location, int inc)
{
    int value := *location
    *location := value + inc
    return value
}
```

- can be used to implement concurrency control structures such as mutex locks and semaphores.
- An atomic fetch_add function appears in the C++11 standard

https://en.wikipedia.org/wiki/Fetch-and-add

Bounded Waiting Mutual Exclusion with TestAndSet

- Shared data (initialized to **false**):
  boolean lock = false; boolean waiting[n]; //init to false
- local variable: **boolean key;**

Enter Critical Section (Lock)

```
waiting[i] = true;
key = true;
while (waiting[i] && key)
   key=TestAndSet(lock);
waiting[i] = false;
```

Leave Critical Section (unlock)

```
j = (i+1)%n
while ((j!=i) && !waiting[j])
   j = (j+1)%n;
if (j == i)
   lock = false;
else
   waiting[j] = false;
```

Question: What is a spinlock? Why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor system?

- 用忙等待方式实现的信号量称为自旋锁。自旋锁等待进入临界区需要占有CPU周期。
- 在单处理器系统中，这将导致已进入临界区的进程得不到机会执行，反而使想进临界区的进程等待更长时间。
- 在多处理器系统中，当临界区很短时自旋锁是合适的。
- 由于有多个处理器，忙等待的进程不影响在临界区中的进程在其他处理器上执行。由于临界区很短，在临界区里的进程很快就能离开临界区，其他忙等待的进程就可以进入它的临界区。这种情况下反而避免了由于阻塞和唤醒导致的上下文切换开销。

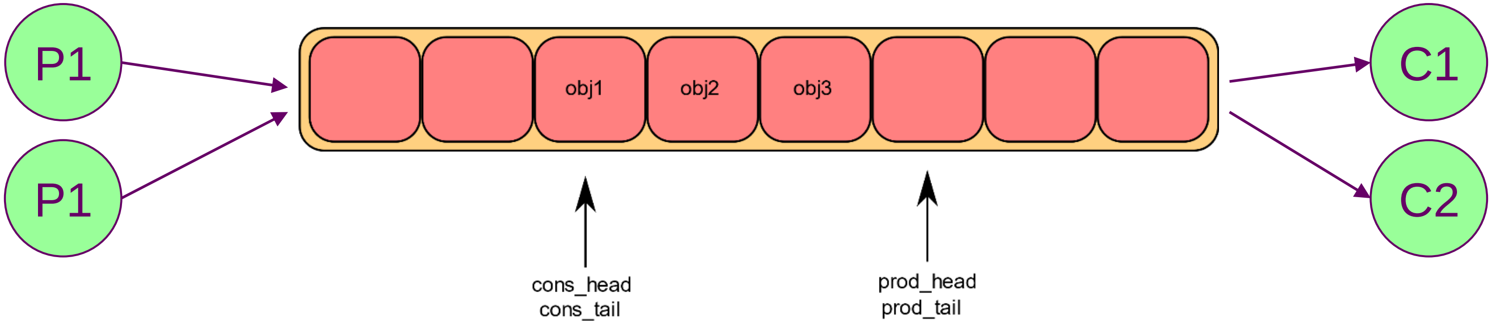
Spin Locks

- A spinlock is a lock which causes a thread trying to acquire it to simply **wait in a loop ("spin")** while repeatedly checking if the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of **busy waiting**.

```
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t*lock);
```

An Introduction to Lock-Free Programming

- People often describe **lock-free programming** as programming without kernel-provided mutexes, which are also referred to as locks.
- An Example: DPDK Lockless Ring Buffer
  - Communication between multiple applications running on different CPU cores



http://dpdk.org/doc/guides/prog_guide/ring_lib.html

  - buffer enqueue and dequeue are implemented by atomic Compare-And-Swap (CAS) instruction

Adaptive Mutex

- Most operating systems (including Solaris, Mac OS X and FreeBSD) use a hybrid approach called "adaptive mutex". The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running. (The latter is *always* the case on single-processor systems.)

https://en.wikipedia.org/wiki/Spinlock#Alternatives

Chapter 6: Process Synchronization

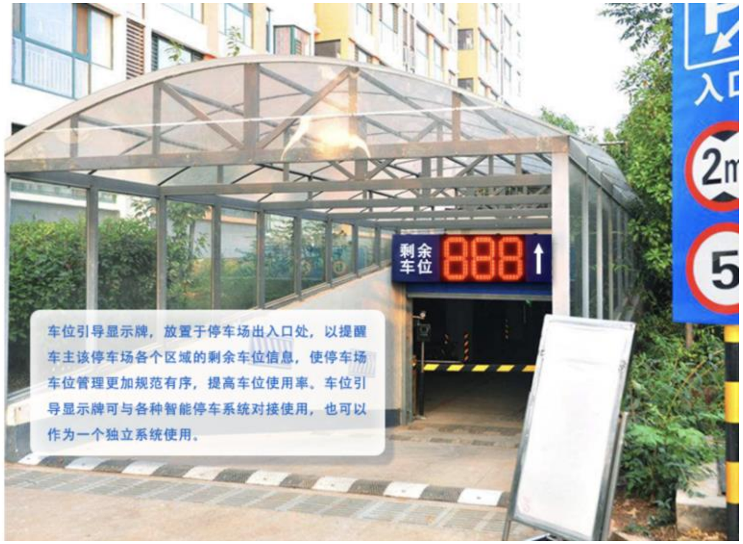
- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphore as a generic synchronization tool
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples

Dijkstra

- Edsger Wybe Dijkstra
  - (Dutch: [ˈɛtsxər ˈʋibə ˈdɛikstra])
  - 11 May 1930 - 6 August 2002 (aged 72)
- Known for
  - Dijkstra's algorithm (single-source shortest path problem)
  - Structured programming, First implementation of ALGOL 60 (“Goto Statements Considered Harmful”)
  - Semaphores, Layered approach to operating system design, software-based paged virtual memory in
    - ✓THE multiprogramming system

Concept of Semaphore

- In real-world systems, semaphores are often used as a synchronization mechanism to control access to a shared resource.
- Semaphores act as a record of the availability of a resource and help coordinating access to it, among multiple processes



- It is a synchronization tool that does not require busy waiting, but needs the support from kernel

semop() - Unix, Linux System Call
http://www.tutorialspoint.com/unix_system_calls/semop.htm

Concept of Semaphore

- Semaphore *S* — an integer variable
- It can only be accessed via two indivisible (atomic) operations: **wait** and **signal**
- They are functionally equivalent to the following busy-waiting operations.

***wait* (*S*):**

while $S \leq 0$ **do** ***no-op*;**
***S*--;**

***signal* (*S*):**

***S*++;**

Semaphore Implementation

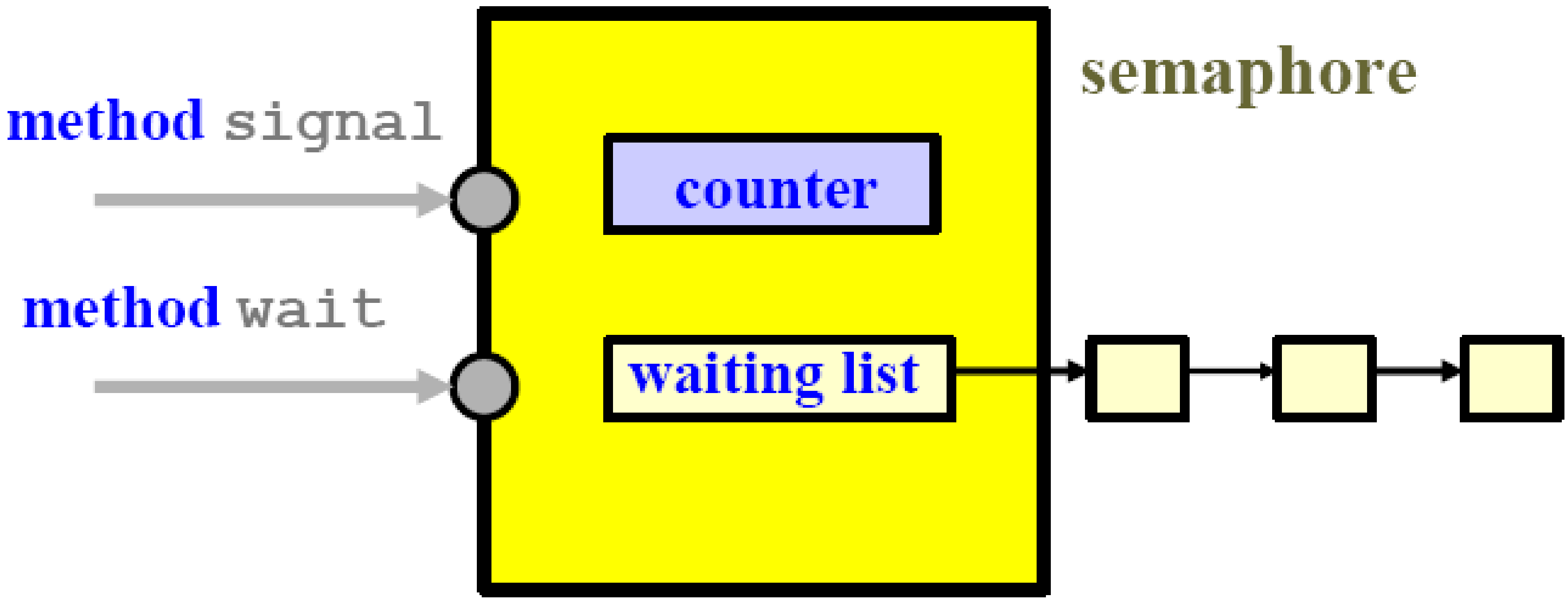
- Define a semaphore as a record

```
typedef struct {
    int counter;
    struct process *L;
    an in-kernel exclusive lock;
} semaphore;
```

- Assume two simple operations:
  - **block:** block the process that invokes it.
  - **wakeup(*P*):** resumes the execution of a blocked process ***P***.

Semaphore Schematics

Semphore = counter + kernel mutex + waiting list



A useful way to think of a semaphore as used in the real-world systems is as a record of how many units of a particular resource are available,

Semaphore Implementation (1)

- Semaphore operations now defined as follows
  - Uniprocessor solution: disable interrupts. Recall that the only way the dispatcher regains control is through interrupts or through explicit requests.

wait(*S*):

Disable interrupts;

S.counter--;

if (S.counter < 0) {

  add this process to **S.L;**

  block;

}

Enable interrupts;

signal(S):

Disable interrupts;

S.counter++;

if (S.counter <= 0) {

  remove a process P from **S.L**;

  wakeup(P);

}

Enable interrupts;

http://www.mpi-sws.org/~druschel/courses/os/lectures/proc4.pdf

Semaphore Implementation (2)

- What do we do in a multiprocessor platform to implement *wait(S)* and *signal(S)*?
  - Can’t turn off interrupts to get low-level mutual exclusion
  - Suppose hardware provides atomic test-and-set instruction

wait(*S*):

while(TAS(S.lock));

S.counter--;

if (S.counter < 0) {

add this process to **S.L;**

block;

}

S.lock = 0;

signal(S):

while(TAS(S.lock));

S.counter++;

if (S.counter <= 0) {

remove a process **P** from **S.L**;

wakeup(P);

}

S.lock = 0;

Semaphore Implementation (3)

Use atomic TAS instrument to implement spin lock

```
while (TestAndSet(S.lock)) ;   // spin lock
    critical section
S. lock = false;   // spin_unlock
    remainder section
```

wait(*S*):

```
S.counter--;
if (S.counter < 0) {
    add this process to S.L;
    block;
}
```

signal(S):

```
S.counter++;
if (S.counter <= 0) {
    remove a process P
    from S.L;
    wakeup(P);
}
```

POSIX Library's Support of Semaphore

- All POSIX semaphore functions and types are prototyped or defined in semaphore.h

  #include <semaphore.h>

- To define a semaphore object, use

  sem_t sem_name;   OR   sem_t * sem_pointer;

- For initialization, use either of the following APIs.

  int sem_init (sem_t *sem, int pshared, unsigned int initial_value);

  sem_t * sem_open (const char* name, int oflag, unsigned int initial_value);

- To increment/decrement the value of a semaphore,

  int sem_wait (sem_t * sem_pointer);

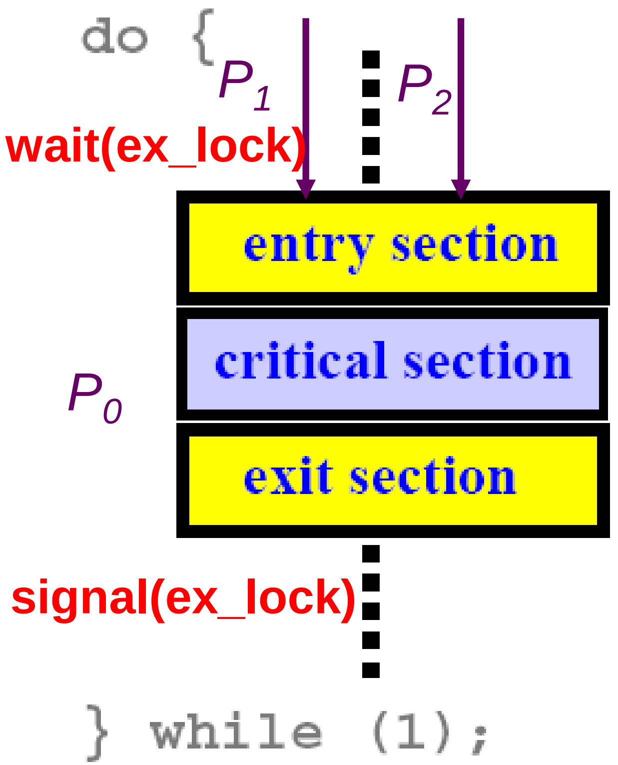
  int sem_post (sem_t * sem_pointer);

Applications of Binary Semaphore: 1. Solve the Critical Section Problem

- Shared data: **semaphore ex_lock;** ***//*** initialize =1

- Process P_i:

```
do {
    wait(ex_lock);
        critical section
    signal(ex_lock);
        remainder section
} while (1);
```



- **Give a demonstration**

Applications of Binary Semaphore: 2. Act as an Event Notification Tool

- Execute *B* in P_j only after *A* executed in P_i

- Use semaphore *flag*, which is initialized to 0

- Shared data: **semaphore flag; //** initialize =0

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

An example of using semaphore to solve critical section problem

- Shared data: **semaphore ex_lock;** ***//*** initialize to 1

P0 P1 P2 User Mode

```
Semaphore ex_lock {
    int counter;
    struct process * L;
    an in-kernel exclusive lock;
}
```

Kernel Mode

wait(*S*):

```
while(TAS(S.lock));
S.counter--;
if (S.counter < 0) {
    add this process to S.L;
    block;
}
S.lock = 0;
```

signal(S):

```
while(TAS(S.lock));
S.counter++;
if (S.counter <= 0) {
    remove a process P
    from S.L;
    wakeup(P);
}
S.lock = 0;
```

进程P1和P2同时竞争ex_lock锁的控制权，可能产生竞争条件。因为都试图修改ex_lock内部的S.counter变量。

此时必须依靠内核信号量S的S.lock自旋锁，将wait(S)和signal(S)方法都实现为关键代码段。

Difference between Binary Semaphore and Mutex

- Question: Is there any difference between binary semaphore and mutex, or they are essentially same?
- Answer: They're semantically the same, but in practice you will notice weird differences
  - Semaphore is implemented by process/thread blocking and wakeup
  - Mutex may be internally implemented by some kernels as spin locks, which could be more efficient on multi-processor systems but will slow down a single processor machine

http://gauss.ececs.uc.edu/Courses/c4029/extra/difference-between-semaphore-and-mutex.html

Side Effect of Semaphore: Deadlock and Starvation

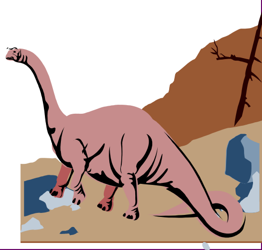
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
  - Example: Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
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 waiting.

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Conditional Variables and Monitors
- Synchronization Examples

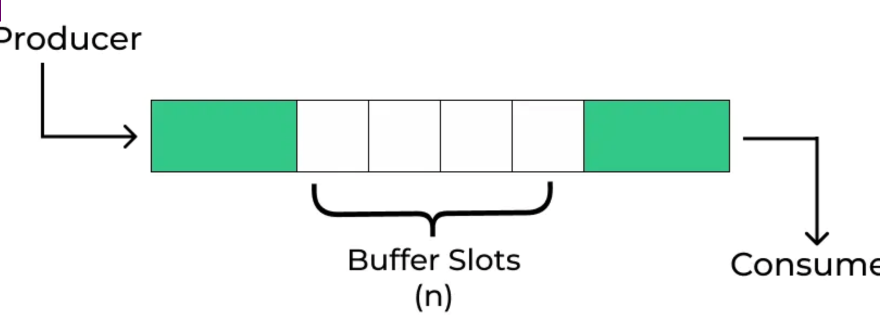


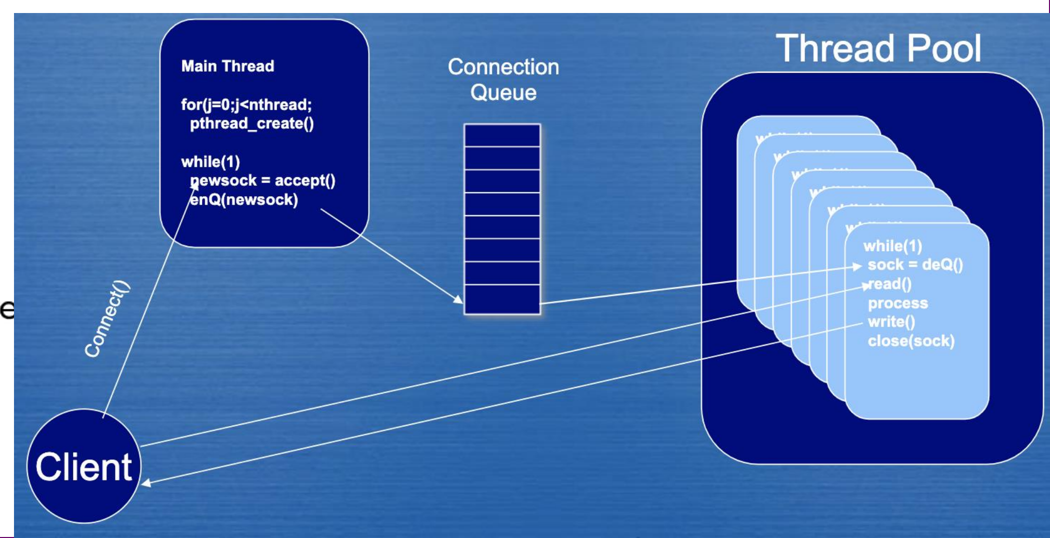
Classical Problems of Synchronization

- Bounded-Buffer Problem (or called Producer-Consumer Problem)
- Readers and Writers Problem (or called Shared-Lock Problem)
- Dining-Philosophers Problem

Producer-Consumer Problem

- Also called bounded buffer problem
- A producer produce data that is to be consumed by a consumer
- A buffer holds produced data not yet consumed
- There exists several producers and consumers
- Application: Multi-threaded web server

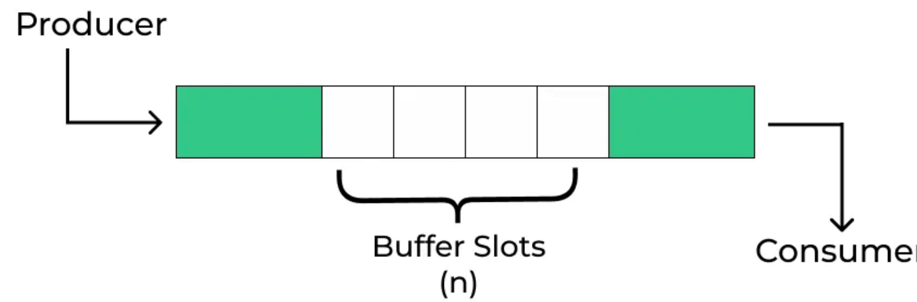




Solution 1 for Producer-Consumer

- Shared variables besides the shared buffer

  semaphore fillCount, emptyCount;



  Initially:

  fillCount = 0, emptyCount = n

- **fillCount:** the number of items in the buffer
- **emptyCount:** the number of empty slots in the buffer

Solution 1 for Producer-Consumer

Producer:

do {

…

produce an item in **nextp**

…

wait(emptyCount);

add **nextp** to buffer

signal(fillCount);

} while (1);

Consumer:

do {

wait(fillCount);

remove an item from buffer to **nextc**

signal(emptyCount);

…

consume the item in **nextc**

…

} while (1);

Question: Is this solution correct?

Solution 1 for Producer-Consumer

- This solution contains a serious race condition that can result in two or more processes modifying into the same cursor *in* or *out* at the same time.

- To understand how this is possible, recall how the procedures “add **nextp** to buffer” and “remove an item from buffer” are implemented, by
  in = (in+1)%BUF_SIZE, and
  out = (out+1)%BUF_SIZE.

Solution 2 for Producer-Consumer

- Shared data

  semaphore fillCount, emptyCount, mutex;

  Initially:

  fillCount = 0, emptyCount = n, mutex = 1

- **mutex**: guarantee the mutual exclusive access of the shared buffer

Solution 2 for Producer-Consumer

Producer:

```
do {
      …
  produce an item in nextp
      …
   wait(mutex);
   wait(emptyCount);
   add nextp to buffer
   signal(fillCount);
   signal(mutex);
} while (1);
```

Consumer:

```
do {
   wait(mutex);
   wait(fillCount);
   remove an item from
   buffer to nextc
   signal(emptyCount);
   signal(mutex);
      …
   consume the item in nextc
      …
} while (1);
```

Question: Is this solution correct? Give a demo

Solution 3 for Producer-Consumer

Producer:

```
do {
      …
  produce an item in nextp
      …
   wait(emptyCount);
   wait(mutex);
   add nextp to buffer
   signal(mutex);
   signal(fillCount);
} while (1);
```

Consumer:

```
do {
   wait(fillCount);
   wait(mutex);
   remove an item from
   buffer to nextc
   signal(mutex);
   signal(emptyCount);
      …
   consume the item in nextc
      …
} while (1);
```

Give a code demonstration

A Short Summary

- 结论1：需要用mutex确保对关键共享资源的互斥访问，比如 shared bounded buffer
- 结论2：信号量wait的顺序很重要
  - 例子：如果wait(mutex)错误放在了wait(fillCount)或者wait(emptyCount)之前，会导致死锁

- 问题1：信号量signal的顺序重要吗？
  - 例子：signal(mutex)和signal(fillCount)可以交换吗？
- 问题2：能否把produce an item和consume an item放到wait(mutex)和signal(mutex)之间？

When the buffer size is only one, can we remove the mutex variable?

Initially: **semaphore fillCount = 0, emptyCount = 1**

Producer:

do {

…

produce an item in **nextp**

…

wait(emptyCount);

add **nextp** to buffer

signal(fillCount);

} while (1);

Consumer:

do {

wait(fillCount);

remove an item from buffer to **nextc**

signal(emptyCount);

…

consume the item in **nextc**

…

} while (1);

Please give out your reasons.

信号量：生产者消费者习题

- 考虑三个吸烟者进程和一个经销商进程的系统
  - 每个吸烟者连续不断地做烟卷并抽他做好的烟卷，做一支烟卷需要烟草、纸和火柴三种原料。
  - 这三个吸烟者分别掌握有烟草、纸和火柴。
  - 经销商源源不断地提供上述三种原料，但他只随机的将其中的两种原料组合（A:烟草+纸，B:纸+火柴，C:烟草+火柴）放在桌上，具有另一种原料的吸烟者就可以做烟卷并抽烟，且在做完后给经销商发信号，然后经销商再拿出两种原料放在桌上，如此反复
- 基于信号量设计一个同步算法描述他们的活动
  - ✓Semaphore empty=1, fullA=0, fullB=0, fullC=0;
  - ✓Broker: SmokerA: SmokerB: SmokerC:

信号量：生产者消费者习题

- 可以考虑：设置三个信号量fullA、fullB和fullC，分别代表三种原料组合，初值均为0，即
  - fullA表示烟草和纸的组合，
  - fullB表示纸和火柴的组合，
  - fullC表示烟草和火柴的组合。
- 桌面上一次只能放一种组合，可以看作是只能放一个产品的共享缓冲区，设置信号量empty初值为1，控制经销商往桌子上放原料

信号量：生产者消费者习题

- 算法

```
Semaphore fullA=fullB=fullC=0, empty=1;

process smokerA() {
    do {
        wait(fullA);
        take tobacco and paper from the table;
        signal(empty);  // signal an empty table event
        make cigarette;
        smoke cigarette;
    } while (1);
}
```

信号量：生产者消费者习题

```
process smokerB() {
    do {
        wait(fullB);
        take paper and match
        from the table;
        signal(empty);
        make cigarette;
        smoke cigarette;
    } while (1);
}
```

```
process smokerC() {
    do {
        wait(fullC);
        take tobacco and match
        from the table;
        signal(empty);
        make cigarette;
        smoke cigarette;
    } while (1);
}
```

信号量：生产者消费者习题

```
process provider( ) {
    integer i;
    do {
        i = random() % 3;  // produce a combination
        wait(empty); // wait for an empty table event
        switch(i) {
            case 0: put T&P on table; signal(fullA); break;
            case 1: put P&M on table; signal(fullB); break;
            case 2: put T&M on table; signal(fullC); break;
        }
    } while(1);
}
```

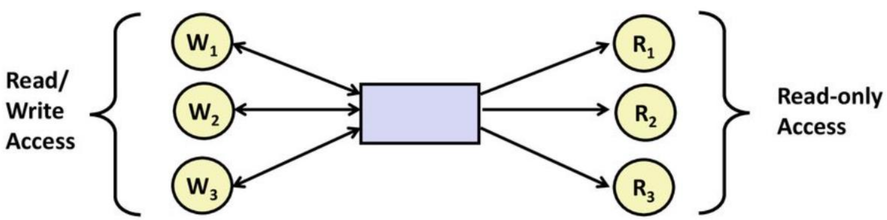
Classical Problems of Synchronization

- Bounded-Buffer Problem (or called Producer-Consumer Problem)
- Readers and Writers Problem (or called Shared-Lock Problem)
- Dining-Philosophers Problem

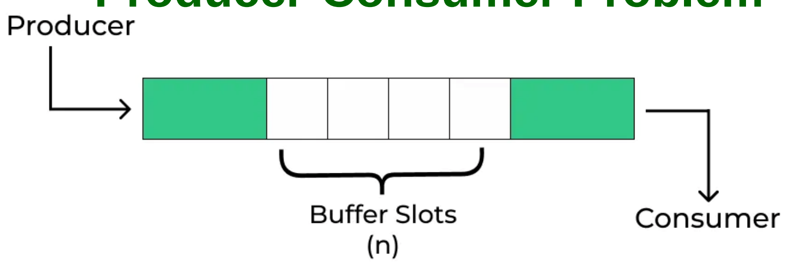
Reader-Writer Locks

- Imagine a number of concurrent operations, including **reads** and **writes**.
  - Writes change the state of the data
  - Reads do not.
    - ✓ Many reads can proceed concurrently, as long as we can guarantee that no write is on-going.

Reader-Writer Problem



Producer-Consumer Problem



- Occurs frequently in real systems, e.g. online airplane booking, N-thread caching web proxy

Readers-Writers Problem (or Shared-Lock Problem)

- Shared data

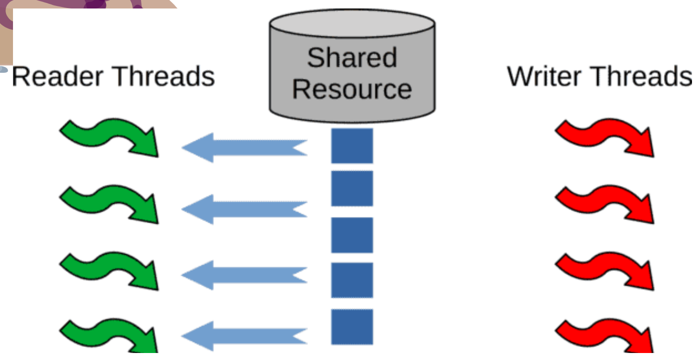
int readcount;

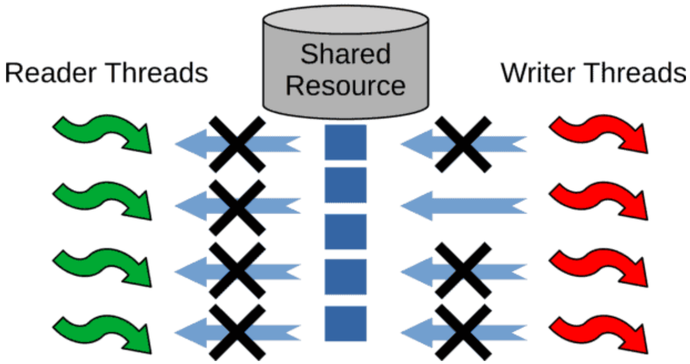
semaphore mutex, wrt;

Initially **readcount = 0, mutex = 1, wrt = 1**

- **readcount**: the number of readers browsing the shared content
- **mutex**: guarantee the mutual exclusive access to the readcount variable
- **wrt**: the right of modifying the shared content

Readers-Writers Problem (solution 1)





Writer Process

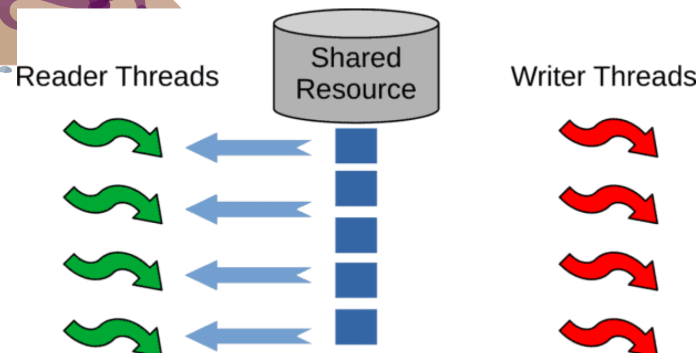
```
wait(wrt);
        …
    writing is performed
        …
signal(wrt);
```

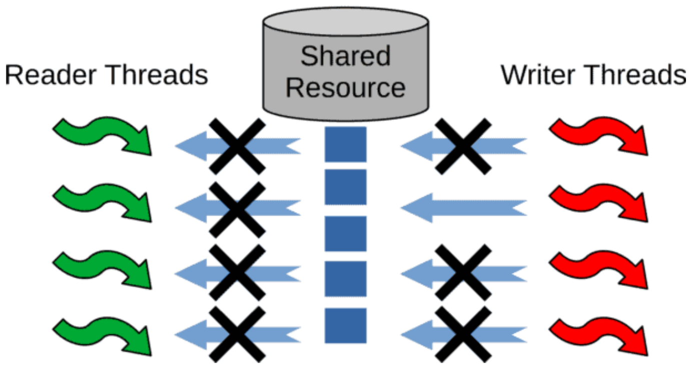
Reader Process

```
wait(mutex);
readcount++;
signal(mutex);
if (readcount == 1)
     wait(wrt);
       …
   reading is performed
       …
wait(mutex);
readcount--;
signal(mutex);
if (readcount == 0)
   signal(wrt);
```

Question: Is this solution correct?

Readers-Writers Problem (solution 2)





Writer Process

```
wait(wrt);
        …
    writing is performed
        …
signal(wrt);
```

Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
     wait(wrt);
signal(mutex);
      …
   reading is performed
      …
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex);
```

```
typedef struct _rwlock_t {
    sem_t * writelock;
    sem_t * lock;
    int readers;
} rwlock_t;
void rwlock_acquire_readlock(rwlock_t * rw) {
    sem_wait(rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(rw->writelock);
    sem_post(rw->lock);
}
void rwlock_release_readlock(rwlock_t * rw) {
    sem_wait(rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(rw->writelock);
    sem_post(rw->lock);
}
void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(rw->writelock);
}
void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(rw->writelock);
}
```

Give a demo

Exercise

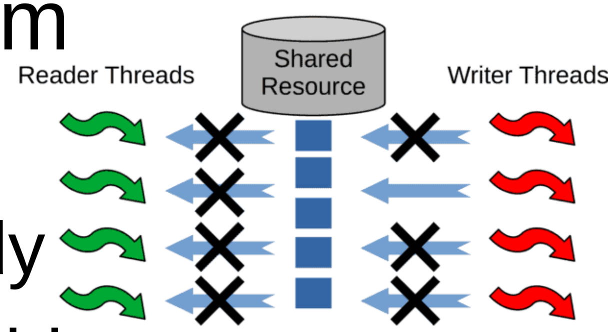
- 由于读者优先，存在写者饥饿问题

R3: read 21	R2: read 21	R3: read 21	R1: read 21
R1: read 21	R4: read 21	R1: read 21	R2: read 21
R2: read 21	R5: read 21	R2: read 21	R4: read 21
R4: read 21	R3: read 21	R4: read 21	R2: done 21
R5: read 21	R1: read 21	R5: read 21	R5: read 21
R3: read 21	R2: read 21	R3: read 21	R1: read 21
R1: read 21	R4: read 21	R1: read 21	R4: read 21
R2: read 21	R5: read 21	R2: read 21	R1: done 21
R4: read 21	R3: read 21	R4: read 21	R5: read 21
R5: read 21	R1: read 21	R5: read 21	R4: done 21
R3: read 21	R2: read 21	R3: read 21	R5: done 21
R1: read 21	R4: read 21	R1: read 21	W2: write 22
R2: read 21	R5: read 21	R2: read 21	W1: write 23
R4: read 21	R3: read 21	R4: read 21	W2: write 24
R5: read 21	R1: read 21	R5: read 21	W1: write 25
R3: read 21	R2: read 21	R3: read 21	W2: write 26
R1: read 21	R4: read 21	R1: read 21	W1: write 27

- 用信号量解决无饥饿的读者——写者问题。

More Info about Reader-Writer Locks

- The first readers–writers problem
  - requires that no reader be kept waiting unless a writer has already obtained access right of shared object.



- The second readers-writers problem
  - requires that once a writer is ready, that writer perform its write as soon as possible.
- Discussion:
  - Which problem is solved by the previous codes?
  - Answer: The first readers-writers problem.
  - How to solve the second readers-writers problem?

The (No-starve) Readers-Writers Problem

```
semaphore lock= 1;
semaphore writelock=1;
int read_count = 0;
semaphore wflock =1;

void write() {
        do {
                wait(wflock);
                wait(writelock);
                /* writing */
                signal(writelock);
                signal(wflock);
  }
  while (1);
}
```

```
void read() {
        do {
                // acquire_readlock
                wait(wflock);
                signal(wflock);
                wait(lock);
                read_count ++;
                if (read_count == 1)
                    wait(writelock);
                signal(lock);
                /* reading */
                // release_readlock
                wait(lock);
                read_count --;
                if (read_count == 0)
                    signal(writelock);
                signal(lock);
  }
  while (1);
}
```

写者利用wflock将后续准备进入的readers阻塞在acquire_readlock

```c
typedef struct
_rwlock_t {
    sem_t * writelock;
    sem_t * lock;
    int readers;
    sem_t * wflock;
} rwlock_t;

void
rwlock_init(rwlock_t *
rw) {
    rw->readers = 0;
    rw->lock =
sem_open(..., 1);
    rw->writelock =
sem_open(..., 1);
    rw->wflock =
sem_open(..., 1);
}
```

```c
void rwlock_acquire_readlock(rwlock_t * rw) {
    sem_wait(rw->wflock);
    sem_post(rw->wflock);
    sem_wait(rw->lock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(rw->writelock);
    sem_post(rw->lock);
}
void rwlock_release_readlock(rwlock_t * rw) {
    sem_wait(rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(rw->writelock);
    sem_post(rw->lock);
}
void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(rw->wflock);
    sem_wait(rw->writelock);
}
void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(rw->writelock);
    sem_post(rw->wflock);
}
```

Give a demo

The (Writer-priority) Readers-Writers Problem

```
int write_count = read_count = 0;
semaphore readcount_lock= 1;
semaphore writecount_lock= 1;
semaphore readlock=1; //0表示不能读
semaphore writelock=1; //0表示不能写
```

```
void write() {
    do {
            wait(writecount_lock);
            write_count ++;
            if (write_count == 1)
                wait(readlock);
            signal(writecount_lock);
            wait(writelock);

            /* writing */

            signal(writelock);
            wait(writecount_lock);
            write_count --;
            if (write_count == 0)
                signal(readlock);
            signal(writecount_lock);
    }
    while (1);
}
```

Give a demo

```
void read() {
    do {
            wait(readlock);
            wait(readcount_lock);
            read_count ++;
            if (read_count == 1)
                wait(writelock);
            signal(readcount_lock);
            signal(readlock);

            /* reading */

            wait(readcount_lock);
            read_count --;
            if (read_count == 0)
                signal(writelock);
            signal(readcount_lock);
    }
    while (1);
}
```

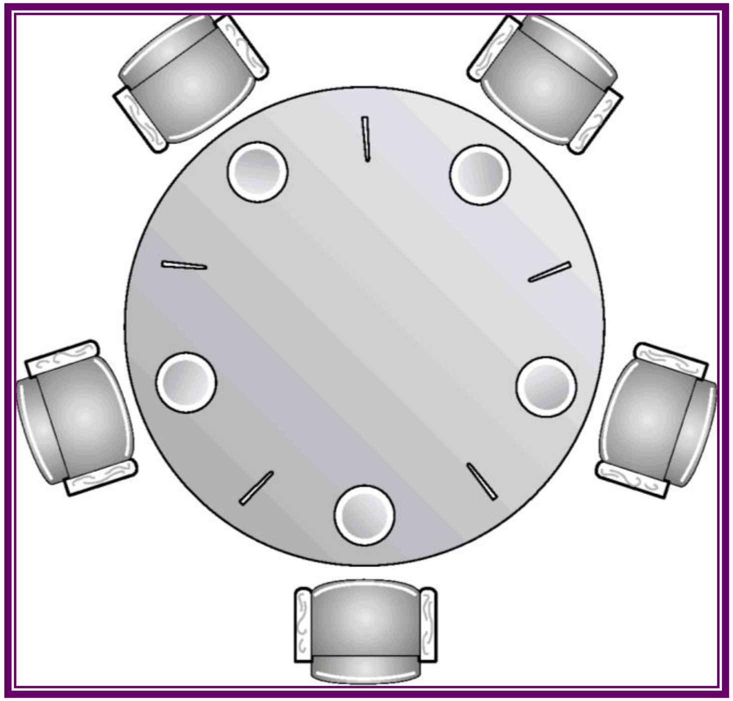
Classical Problems of Synchronization

- Bounded-Buffer Problem (or called Producer-Consumer Problem)
- Readers and Writers Problem (or called Shared-Lock Problem)
- Dining-Philosophers Problem

The Dining Philosophers

- Originally formulated in 1965 by Edsger Dijkstra
- Tony Hoare gave the problem its present formulation

Dining-Philosophers Problem



Here is the basic loop of each philosopher:

```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}
```

Shared data

semaphore chopstick[5];

Initial values of all semaphores are set to 1

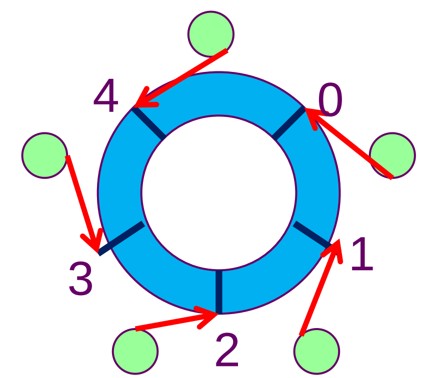
Dining-Philosophers Problem

- Philosopher *i*:

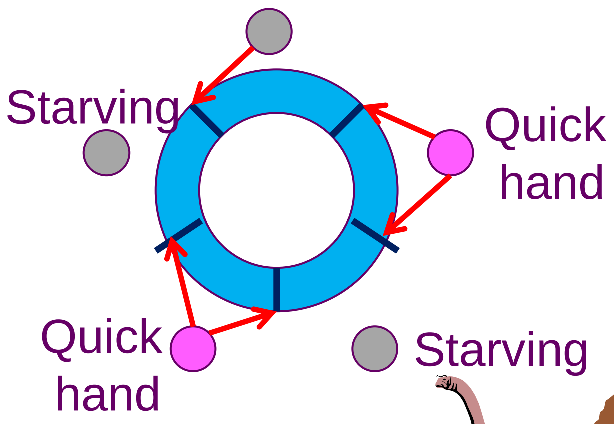
```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        …
        eat
        …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        …
        think
        …
} while (1);
```

Give a demo

- Challenges
  - Deadlock



  - Starvation



  - Lack of Fairness

Semaphore 学习的四重境界

1. 理解基础概念

2. 熟练掌握经典问题（PC, RW, DP）。

3. 熟悉经典问题的变种，能够将应用题恰当的归约到某个经典问题的变种。

4. 能够将经典问题灵活组合应用，随心所欲，信手拈来。

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Condition Variables and Monitors
- Synchronization Examples

Condition Variable

- Semaphore and condition variables are very similar and are used mostly for the same purposes.
  - Semaphore can be easily understood as an in-kernel counter for the units of a type of resource.
  - Condition is an advanced event notification tech.
- However, there are minor differences that could make one preferable.
  - For example, to implement barrier synchronization, you would not be able to use a semaphore. But a condition variable is ideal.

Condition Variable

- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.

  Semaphore = counter + mutex + waiting list

  Conditional Variable = waiting list

- A problem of semaphore: We cannot read the in-kernel counter hiding inside a semaphore
- A condition variable must be used inside a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.

大家工作中条件变量会用的更多

- Java在企业开发市场占比80%

- Java最常用的同步机制
  - （1）synchronized关键字实现的条件变量。每一个Java对象就有一把看不见的锁，称为内部锁或者Monitor锁，内部。
  - （2）Lock接口及其实现类，如ReentrantLock.ReadLock和ReentrantReadWriteLock.WriteLock。

不可不说的Java“锁”事
https://tech.meituan.com/2018/11/15/java-lock.html

Condition Variable vs. Semaphore

Semaphore	Condition Variable
Can be used anywhere	Must be used inside the protection of a mutex
wait() does not always block its caller	wait() always blocks its caller
signal() either releases a process, or increases the semaphore counter	signal() either releases a process, or the signal is lost as if it never occurs
If signal() releases a process, the caller and the released both continue	If signal() releases a process, either the caller or the released continues, but not both

Condition Variable in Pthread Library

- Creating/Destroying:
  - pthread_cond_t cond = THREAD_COND_INITIALIZER;
  - pthread_cond_init
  - pthread_cond_destroy
- Waiting on condition:
  - pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) - unlocks the mutex and waits for the condition variable *cond* to be signaled.
- Waking thread based on condition:
  - pthread_cond_signal(pthread_cond_t *cond) - restarts one of the threads that are waiting on the condition variable *cond*.
  - pthread_cond_broadcast(pthread_cond_t *cond) - wake up all threads blocked by the specified condition variable.

Barrier Problem

- Suppose we wanted to perform a multi-threaded calculation that has two stages, but we don't want to advance to the second stage until the first stage is completed.

- We could use a synchronization method called a barrier. When a thread reaches a barrier, it will wait at the barrier until all the threads reach the barrier, and then they'll all proceed together.

Barrier Problem

- Pthreads has a **pthread_barrier_wait()** function that implements this. You'll need to declare a **pthread_barrier_t** variable and initialize it with **pthread_barrier_init()**.
  - **pthread_barrier_init()** takes the number of threads that will be participating in the barrier as an argument.
- Now let's implement our own barrier and use it to keep all the threads in sync in a large calculation.

Barrier Implementation by Condition Variable

```
#define N (16)
double data[256][8192] ;
pthread_mutex_t m;
pthread_cond_t cv;
int main() {
    int tids[N], i;
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&cv, NULL);
    for(i = 0; i < N; i++) { tids[i] = i;
        pthread_create(&ids[i], NULL, calc, &(tids[i]));
    }
    for(i = 0; i < N; i++) pthread_join(ids[i], NULL);
```

https://github.com/angrave/SystemProgramming/wiki/Synchronization%2C-Part-6%3A-Implementing-a-barrier

Barrier Implementation by Condition Variable

double data[256][8192]

void *calc(void *ptr) {

1. Threads do first calculation (use and change values in data)

2. Barrier! Wait for all threads to finish first calculation before continuing

3. Threads do second calculation (use and change values in data)

}

https://github.com/angrave/SystemProgramming/wiki/Synchronization%2C-Part-6%3A-Implementing-a-barrier

If using condition variable, the state of counter can be access. But when using semaphore, the state of inner count cannot be accessed.

```
#int remain = N;
void *calc(void *ptr) {
 // The thread does first calculation
 pthread_mutex_lock(&m);
 remain--;
 if (remain ==0)  pthread_cond_broadcast(&cv);
 else {
 while(remain != 0) pthread_cond_wait(&cv,&m);
 }
 pthread_mutex_unlock(&m);
 // The thread does second calculation
}
```

Give a demo

Object-Oriented Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

monitor *monitor-name*

{ shared variable declarations

procedure body *P1* **(…) {**

. . .}

procedure body *P2* **(…) {**

. . .}

procedure body *Pn* **(…) {**

. . .}

{ initialization code **}**

}

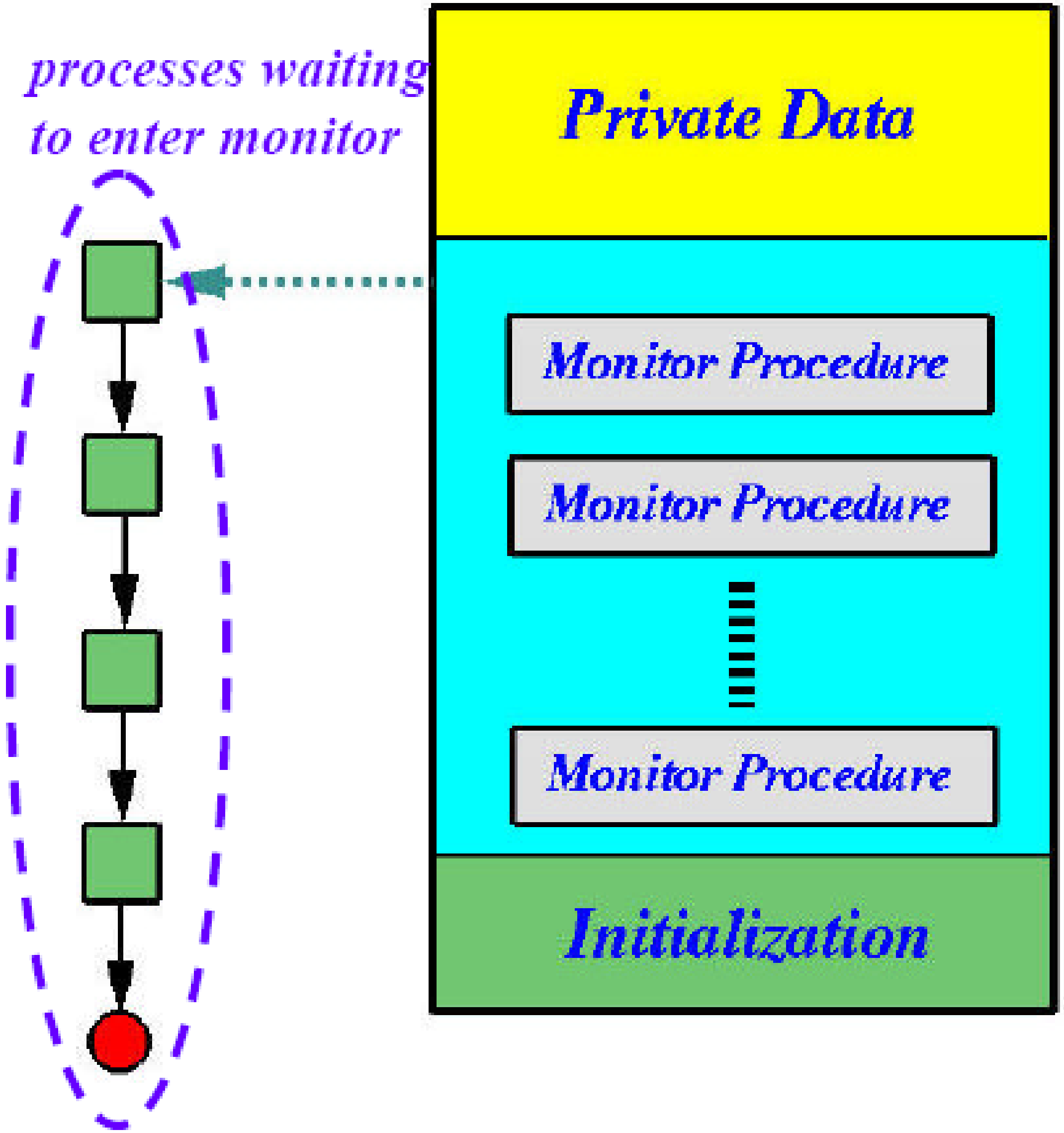
Monitors: Mutual Exclusion

- *No more than one process* can be executing *within* a monitor. Thus, *mutual exclusion* is guaranteed within a monitor.

- When a process calls a monitor procedure and enters the monitor successfully, it is the *only* process executing in the monitor.

- When a process calls a monitor procedure and the monitor has a process running, the caller will be blocked outside of the monitor.

Schematic View of a Monitor



Monitors: Event Notification

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

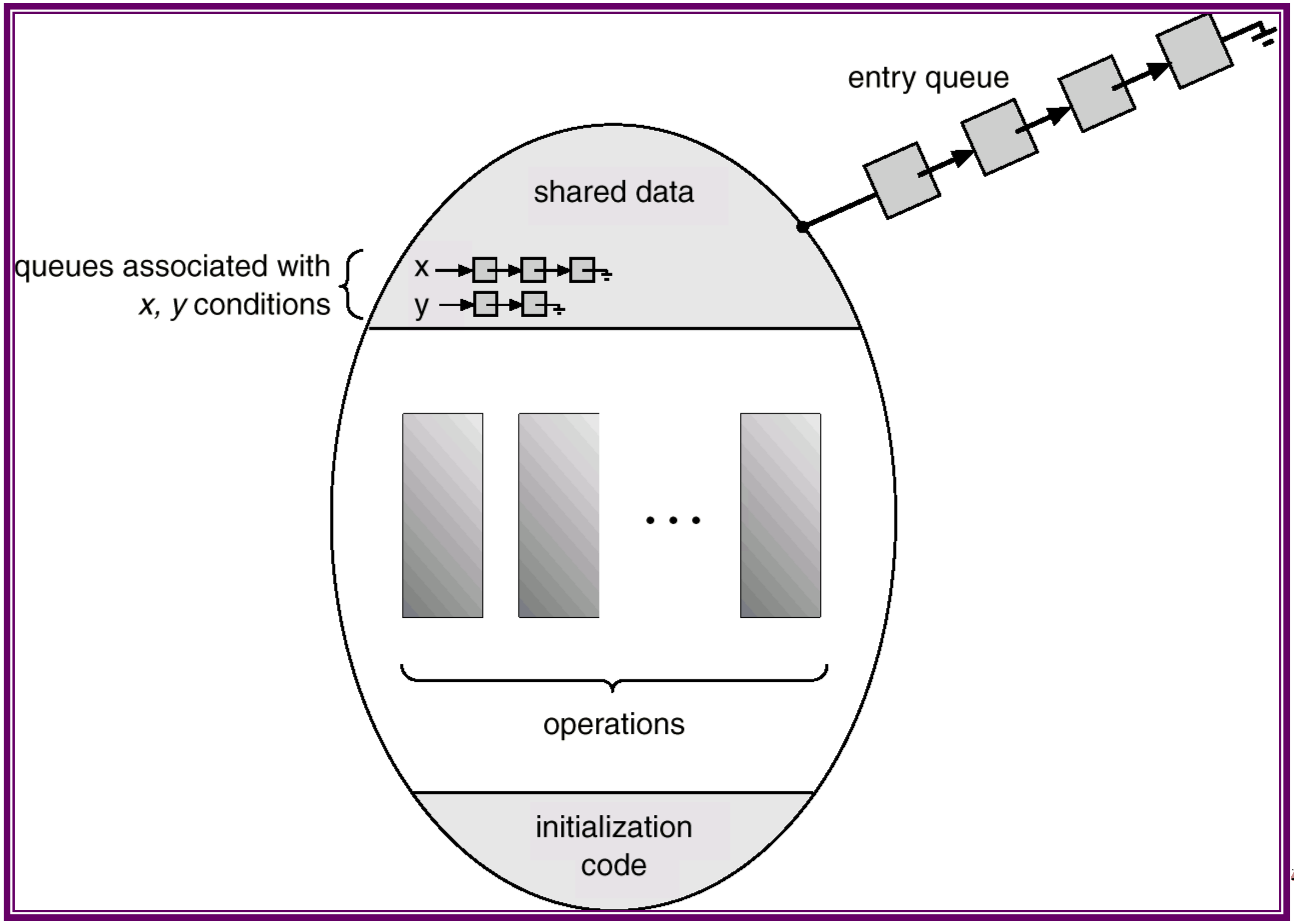
- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

    means that the process invoking this operation is blocked until another process invokes

    x.signal();
  - The **x.signal** operation wakeup exactly one blocked process. If no process is waiting for the condition, then the **signal** operation has no effect.

Schematic View of a Monitor With Condition Variables



A Subtle Issue of Condition Variable

- **Consider the released process (from the signaled condition) and the process that signals. There are two processes executing in the monitor, and mutual exclusion is violated!**

- There are two common and popular approaches to address this problem:
  - The released process takes over the monitor and the signaling process waits somewhere.
  - The released process waits somewhere and the signaling process continues to use the monitor.

Java's Monitor Supports

- Synchronized methods for mutual exclusion

```
class classname {
        synchronized return_type methodname() {……}
}
```

- Coordination support for event notification

Method	Description
void Object.wait();	Enter a monitor's wait set until notified by another thread
void Object.wait(long timeout);	Enter a monitor's wait set until notified by another thread or timeout milliseconds elapses
void Object.notify();	Wake up one thread waiting in the monitor's wait set. (If no threads are waiting, do nothing.)
void Object.notifyAll();	Wake up all threads waiting in the monitor's wait set. (If no threads are waiting, do nothing.)

http://www.ibm.com/developerworks/cn/java/j-lo-synchronized/index.html

http://www.artima.com/insidejvm/ed2/threadsynchP.html

Producer-Consumer Example

```
procedure producer() {
    do {
        item = produceItem();
        PCbuffer.add(item);
    } while (true);
}

procedure consumer() {
    do {
        item = PCbuffer.remove();
        consumeItem(item);
    } while (true);
}
```

```
monitor PCbuffer {
    int itemCount; // <= BUFSIZE
    condition full, empty;
    putItemIntoBuffer(item) {…}
    Item removeItemFromBuffer()
    {…}
    procedure void add(item) {
        ...   // how to implement?
    }
    procedure item remove() {
        …  // how to implement?
    }
}
```

Producer-Consumer Example

```
procedure void add(item) {
    if (itemCount == BUFSIZE)
        full.wait();
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1)
        empty.signal();
    return;
}
```

```
procedure item remove() {
    if (itemCount == 0)
        empty.wait();
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFSIZE - 1)
        full.signal();
    return item;
}
```

Note that **if** statement has been used in the above code, both when testing if the buffer is full or empty.

With multiple consumers, there is a race condition between the consumer who gets notified that an item has been put into the buffer and another consumer who is waiting on the monitor.

Producer-Consumer Example

```
procedure void add(item) {
    while (itemCount == BUFSIZE)
        full.wait();
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1)
        empty.signal();
    return;
}
```

```
procedure item remove() {
    while (itemCount == 0)
        empty.wait();
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFSIZE - 1)
        full.signal();
    return item;
}
```

- Note that **while** statement has been used in the above code, both when testing if the buffer is full or empty.
- With multiple consumers, there is a race condition between the consumer who gets notified that an item has been put into the buffer and another consumer who is waiting on the monitor.

Producer-Consumer Example

```
procedure void add(item) {
    while (itemCount == BUFSIZE)
        full.wait();
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1)
        empty.signal();
    return;
}
```

```
procedure item remove() {
    while (itemCount == 0)
        empty.wait();
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFSIZE - 1)
        full.signal();
    return item;
}
```

- With multiple producers, there is also a race condition between the producer who gets notified that the buffer is no longer full and another producer is already waiting on the monitor.
- If the **while** was instead an **if**, too many items might be put into the buffer or a remove might be attempted on an empty buffer.

Dining Philosophers without Deadlock

```
monitor dining_philosopher_sync_table
{
    enum {thinking, hungry, eating}   state[5];
    condition self[5];
procedure void pickup(int i) ;        // pick up chopsticks
 procedure void putdown(int i) ;  // put down chopsticks
private void test(int i) ; // test if P_i is eligible for eating
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```

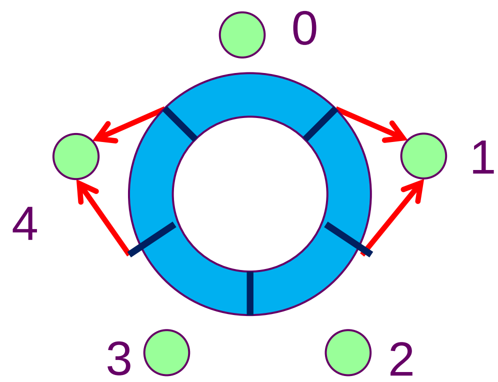
Dining Philosophers without Deadlock

```
void pickup(int i) {
      state[i] = hungry;
      test(i);
      while(state[i] != eating)
            self[i].wait();
}
```

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

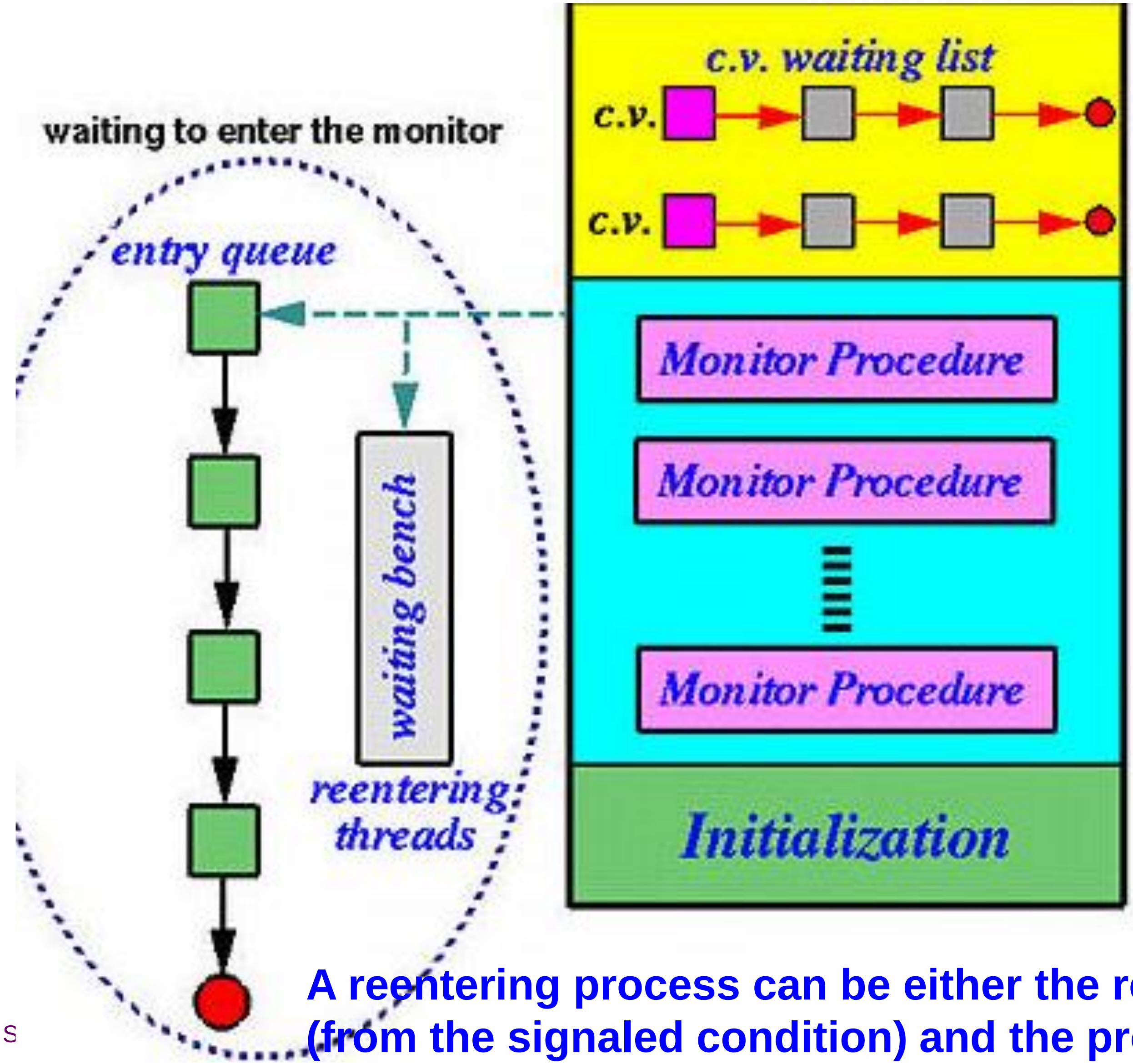
```
void test(int i) {
      if ( (state[(i + 4) % 5] != eating) &&
           (state[i] == hungry) &&
           (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
      } }
```

The code has NO deadlock!!! Why?



When P_1 and P_4 finish eating at the same time, will P_2 and P_3 compete for their common chopstick after their wakeup?

Another Subtle Issue of Monitor: Queue of Reentering Threads/Proc



A reentering process can be either the released process (from the signaled condition) and the process that signals

For Better Understanding, Let's Implement Monitor by Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each external procedure ***F*** will be replaced by

```
wait(mutex);
     …  //  body of F;
if (next-count > 0)
   signal(next);
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation Using Semaphores

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

Monitor Implementation (Cont.)

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Condition Enhanced with a Priority Number

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Condition Variables and Monitors
- Synchronization Examples

Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables* , *semaphore*, and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Adaptive Mutex

- Most operating systems (including Solaris, Mac OS X and FreeBSD) use a hybrid approach called "adaptive mutex". The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running. (The latter is *always* the case on single-processor systems.)

https://en.wikipedia.org/wiki/Spinlock#Alternatives

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Also provides *dispatcher objects* which may act as mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.