Chapter 6: Process Synchronization

肖 卿 俊
办公室：计算机楼532室
电邮：csqjxiao@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/csqjxiao
电话：025-52091023

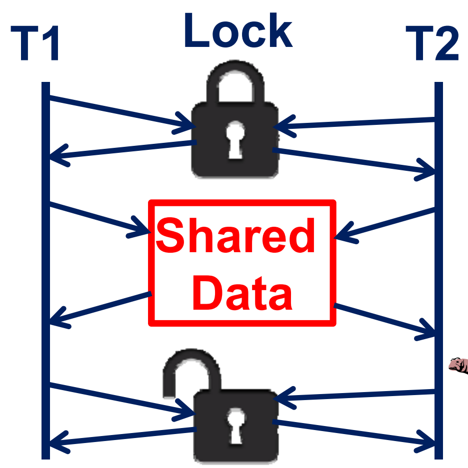
Chapter 6: Process Synchronization

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

Background

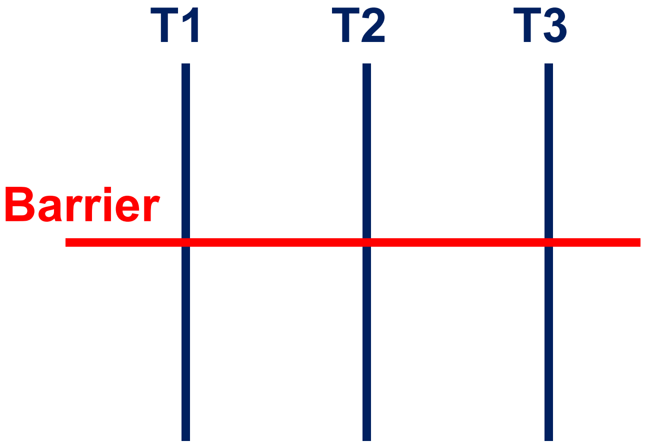
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Synchronization primitive for shared memory programming --- Lock
  - Exclusive Lock

  - Shared Lock



Background (cont.)

- Other synchronization primitive for shared memory programming --- Barrier



- As for this course, we focus primarily on the lock primitive
- Ideas are easy. Implementation is hard.

Revisit the Shared-memory Producer Consumer Problem

- Shared-memory solution to bounded-buffer problem (Chapter 3) allows at most *n* – 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.

Producer:

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

Consumer:

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

  - Suppose that we modify the producer-consumer code by adding a variable *counter*

Bounded-Buffer Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer Solution

- Producer process

```
item nextProduced;

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

Bounded-Buffer Solution

- Consumer process

```
item nextConsumed;

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

Critical Shred 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
- Monitors
- Synchronization Examples

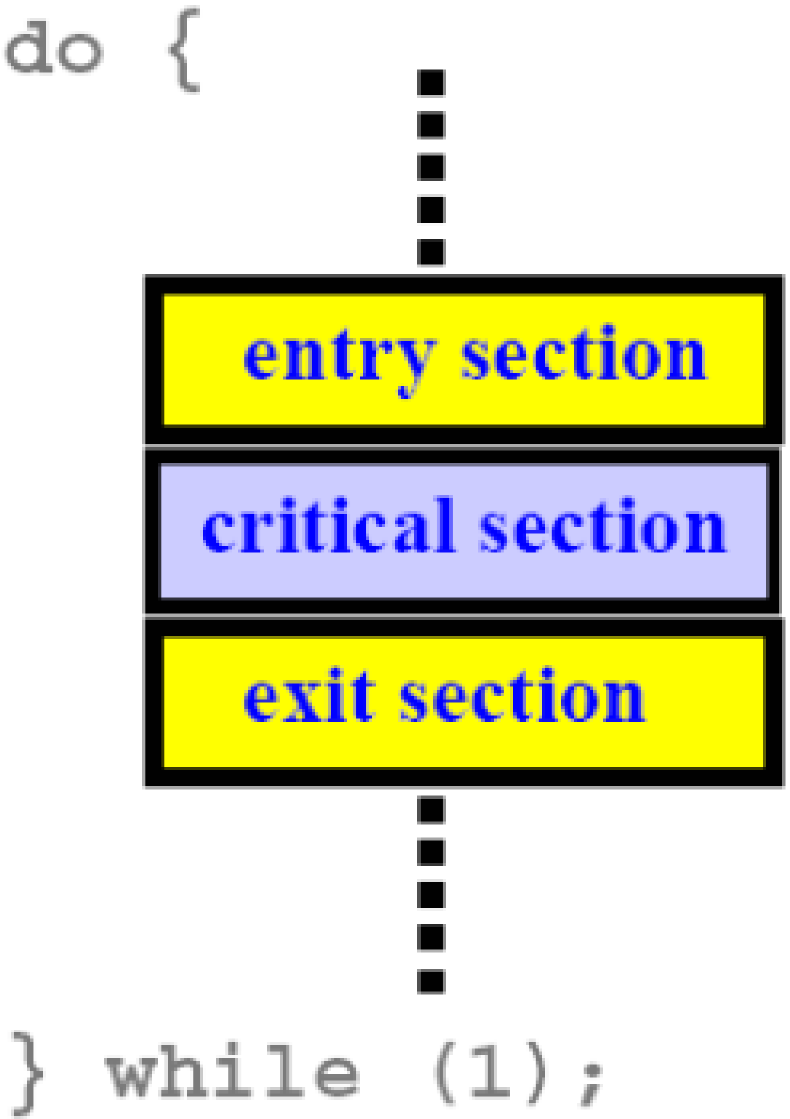
The Critical-Section Problem

- 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



- 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 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

Initial Attempts to Solve Problem

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

do { ***entry section***

critical section

exit section

remainder section

} while (1);

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

Algorithm 1

- Shared variable:
  - **boolean lock**;
    initially **lock = false**
  - **lock = true** $\Rightarrow$ the critical section has been locked
- Process P_i :

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

https://en.wikipedia.org/wiki/Test-and-set

- Does not satisfy mutual exclusion

Algorithm 2

- 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;
while (flag[j]) ;
critical section
flag[i] = false;
remainder section
} while (1);

- Satisfies mutual exclusion, but not progress requirement

Deadlock Problem of Algorithm 2

thread 0	thread 1
flag[0] = true	flag[1] = true
while(flag[1] == true):	while(flag[0] == true);
wait	wait

deadlock!

Is the Following Algorithm Correct?

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

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

- Does not satisfy mutual exclusion

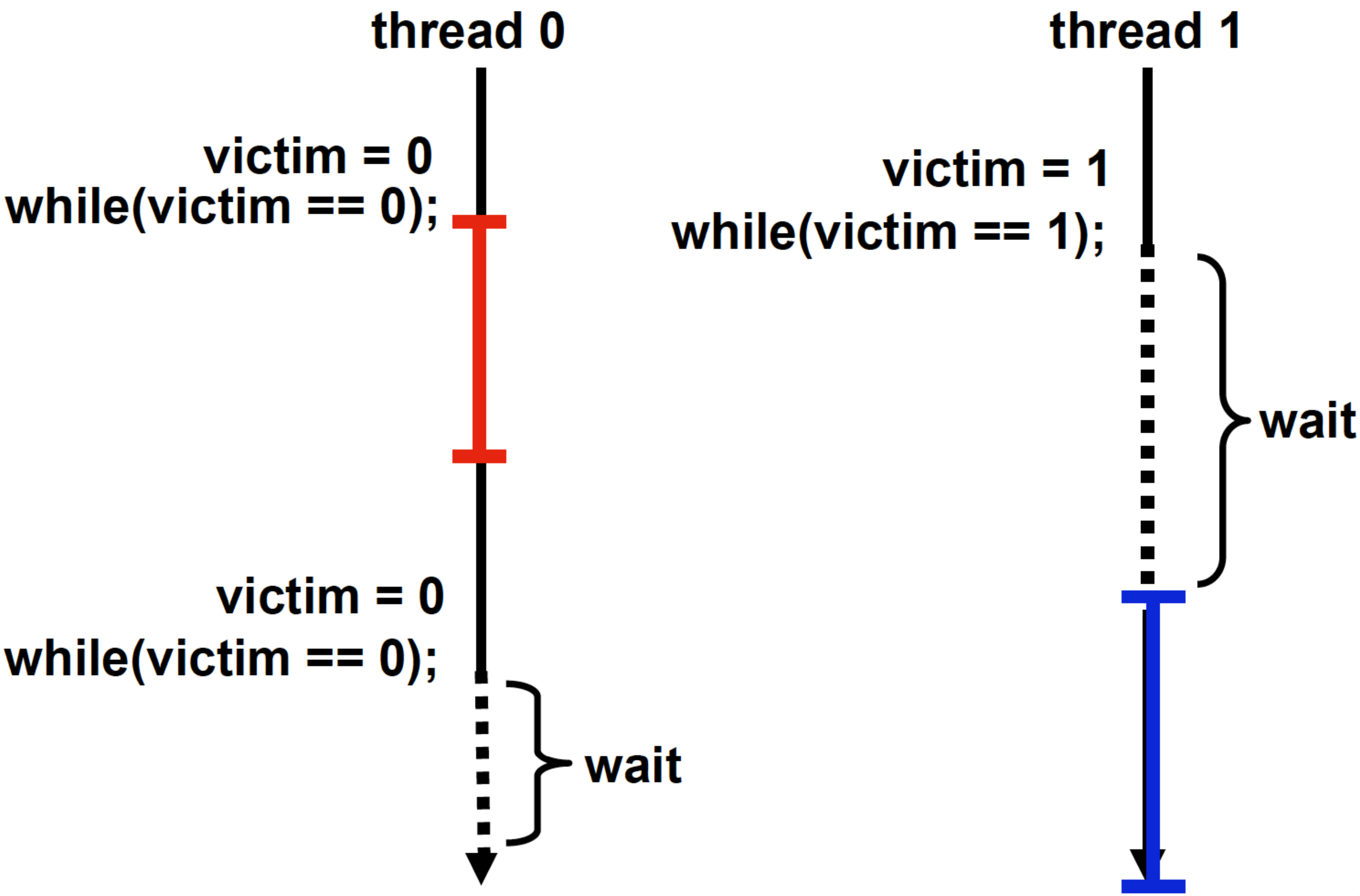
Algorithm 3

- Shared variables:
  - **int victim**; initially **victim = i (or victim = j)**
- Process P_i :

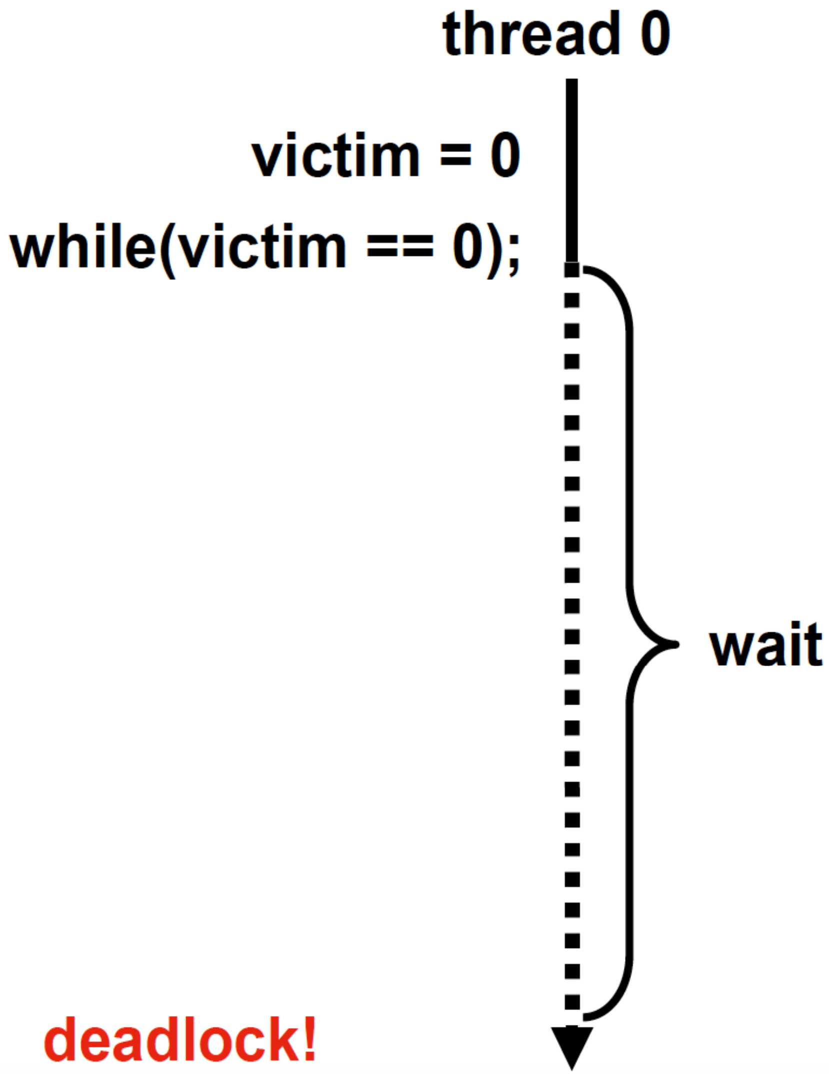
```
do {    victim = i;
        while (victim == i) ;
          critical section
        // do nothing for CS exit
          remainder section
} while (1);
```

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

Alternating and Atomic Execution of Algorithm 3



Deadlock of Algorithm 3



Peterson's Algorithm

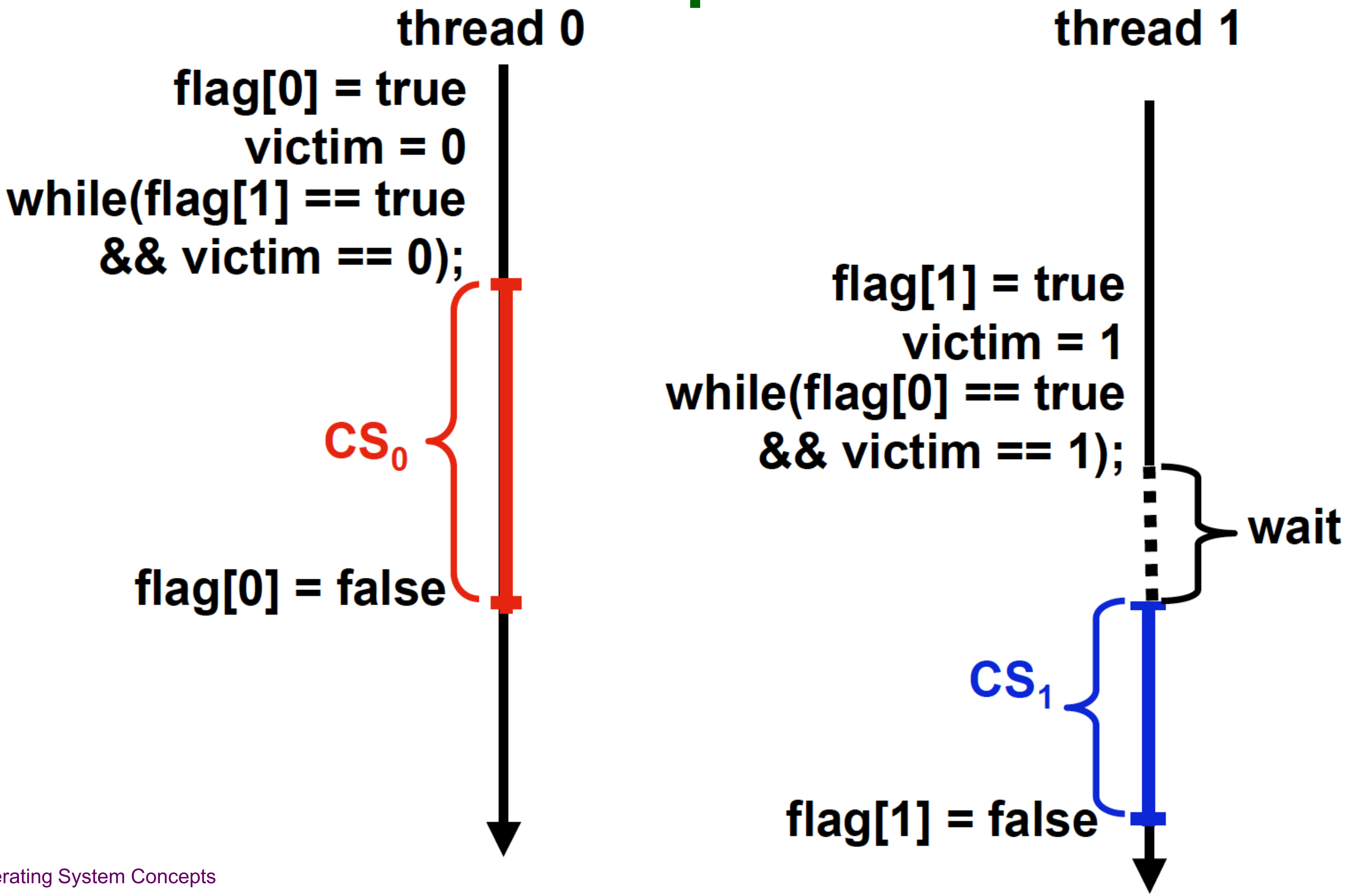
- Combined shared variables of algorithms 2, 3.
- Process P_i

```
do {      flag[i] = true;    // I'm interested
          victim = i;        // you go first
          while (flag[j] and victim == i) ;
             critical section
          flag[i] = false;
             remainder section
} while (1);
```

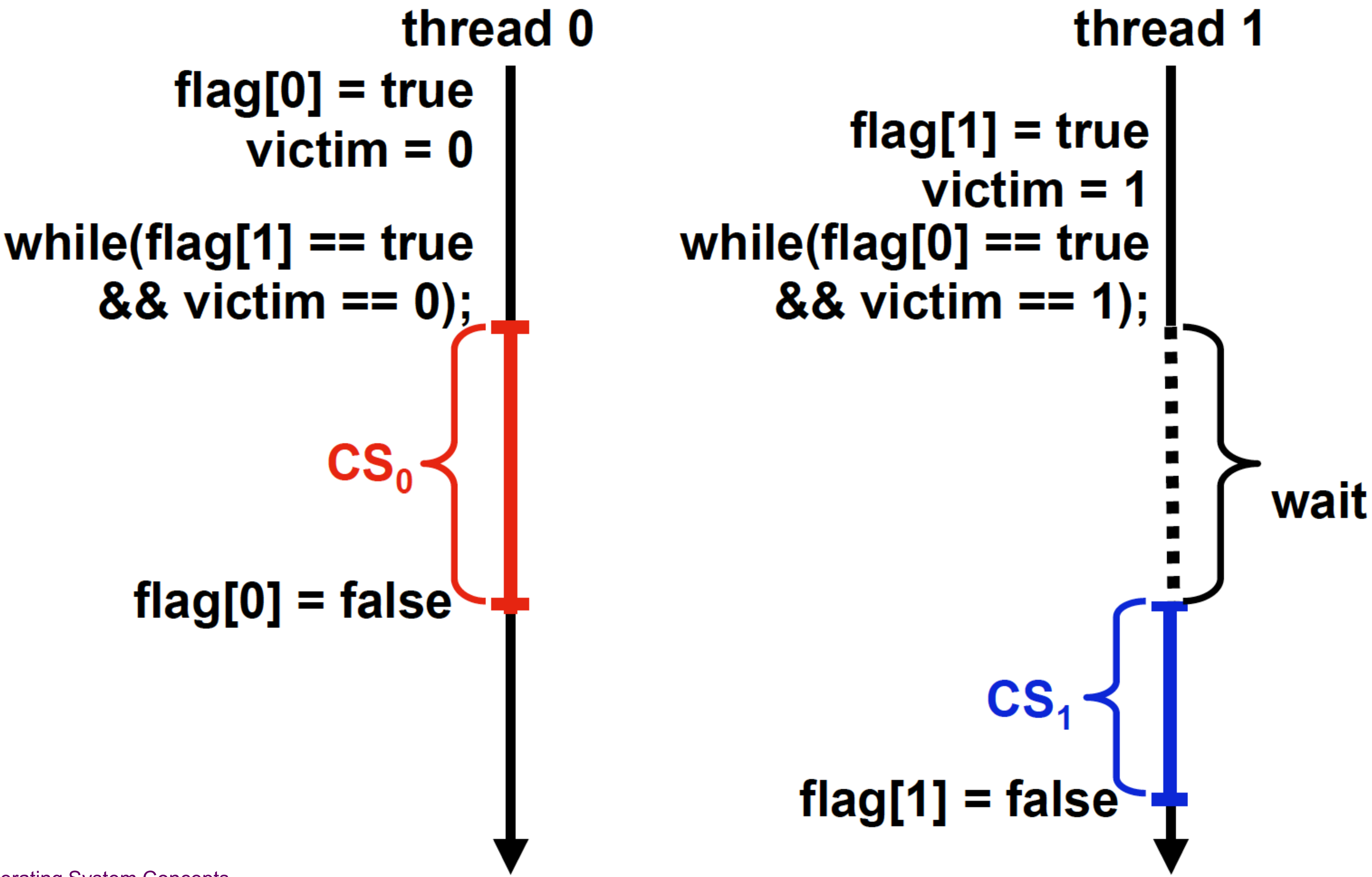
Gary Peterson. Myths about the Mutual Exclusion Problem. Information Processing Letters, 12(3):115-116, 1981.

- Meets all the three requirements; solves the critical-section problem for two processes.

Peterson’s Lock: Serialized Acquires



Peterson's Lock: Concurrent Acquires



Proof of Peterson's Algorithm

- The mutual exclusion requirement is assured.
- The progress requirement is assured. The turn 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 have the turn. That process will proceed.
- Finally, bounded waiting is assured. When a process that has exited the CS reenters, it will give away the turn. If the other process is already waiting, it will be the next to proceed.

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

Is the following code correct?

- What if we change **victim = i** to **victim = j**?
- Process P_i

```
do {        flag[i] = true;   // I'm interested
            victim = j;       // I go first
            while (flag[j] and victim == i) ;
                  critical section
             flag[i] = false;
                  remainder section
} while (1);
```

- Does not satisfy mutual exclusion, and not bounded waiting

Lamport's Bakery Algorithm

Critical section for n 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, …, $n - 1$

- 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);
```

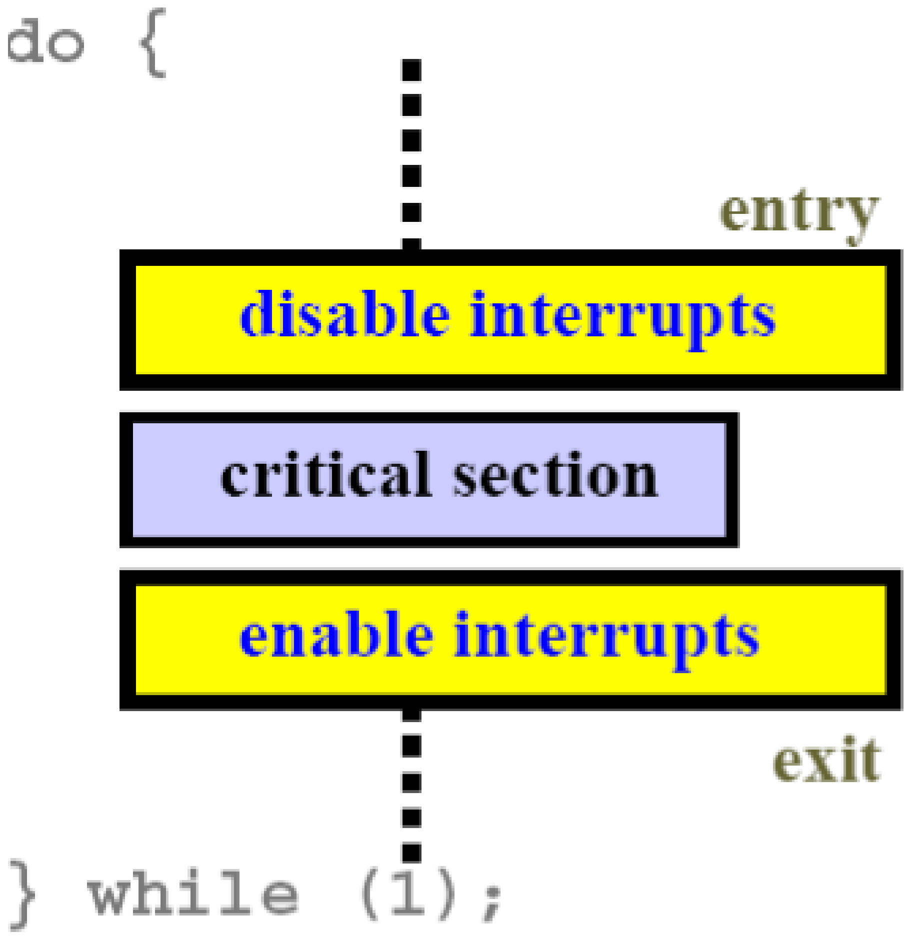
Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- 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 (TS)
    - ✓Swap

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

- Test and modify the content of a 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
}
```

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;
```

- local variable

```
        boolean key;
```

- Process P_i

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

Bounded Waiting Mutual Exclusion with TestAndSet

Enter Critical Section

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

Leave Critical Section

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

Chapter 6: Process Synchronization

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

Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):

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

***S*--;**

signal (*S*):

***S*++;**

Critical Section of *n* Processes

- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process P_i:

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

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

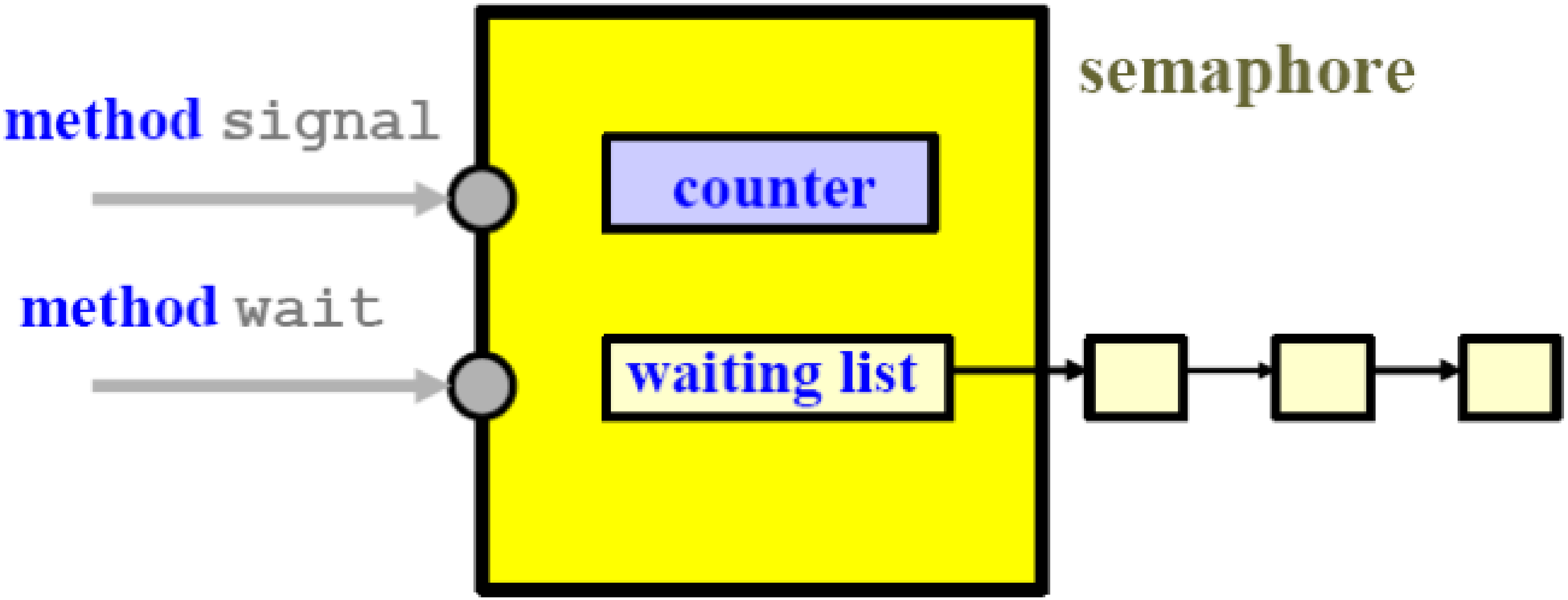
P_i	P_j
$\vdots$	$\vdots$
A	*wait*(*flag*)
signal(*flag*)	*B*

Semaphore Implementation

- Define a semaphore as a record

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

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


semaphore
method signal
method wait
counter
waiting list

Implementation

- Semaphore operations now defined as

```
wait(S):
        S.value--;
        if (S.value < 0) {
                add this process to S.L;
                block;
        }
signal(S):
        S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);
        }
```

Deadlock and Starvation

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

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 suspended.

Chapter 6: Process Synchronization

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

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

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

Bounded-Buffer Problem Consumer Process

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

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

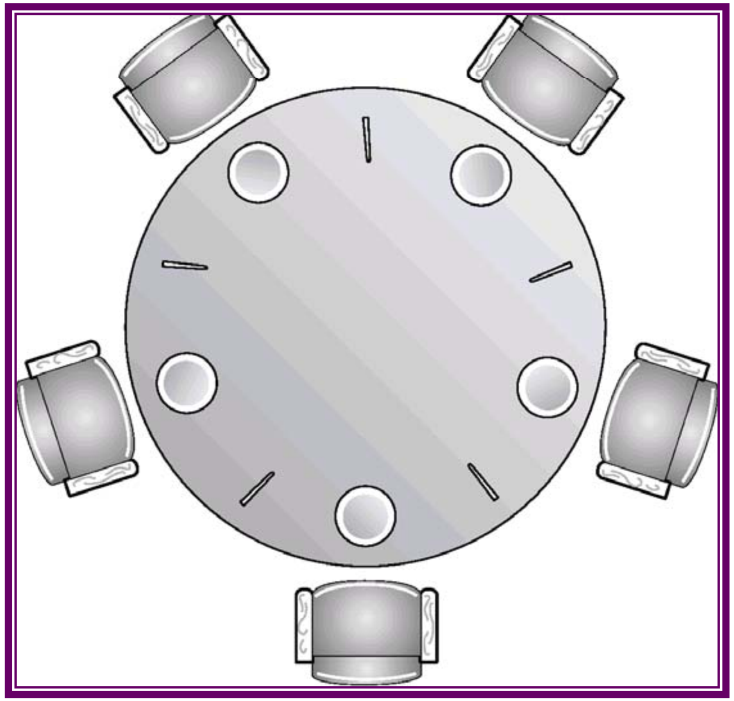
Readers-Writers Problem Writer Process

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

Readers-Writers Problem 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):
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

Initially all values are 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);
```

Chapter 6: Process Synchronization

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

Monitors

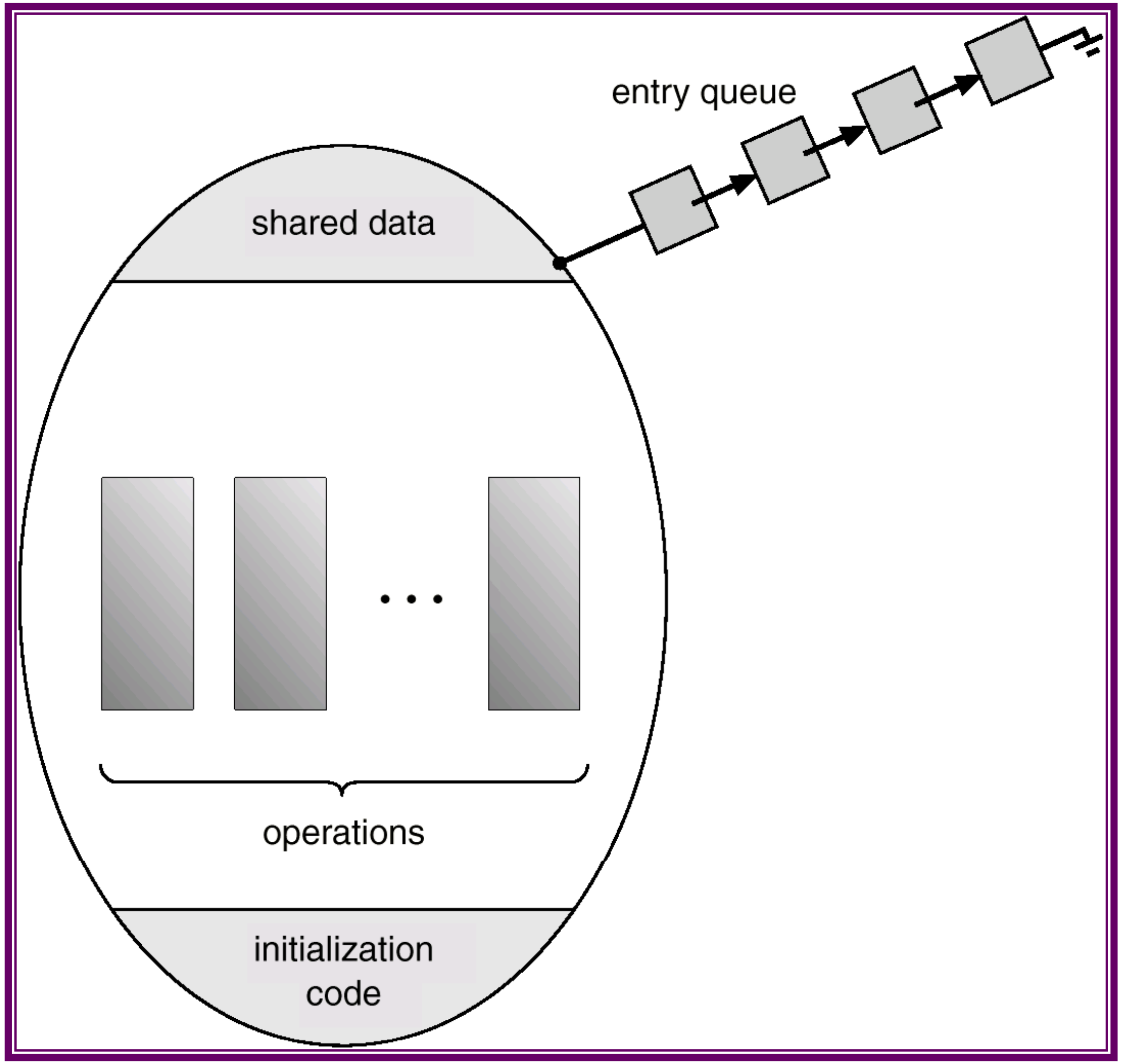
- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

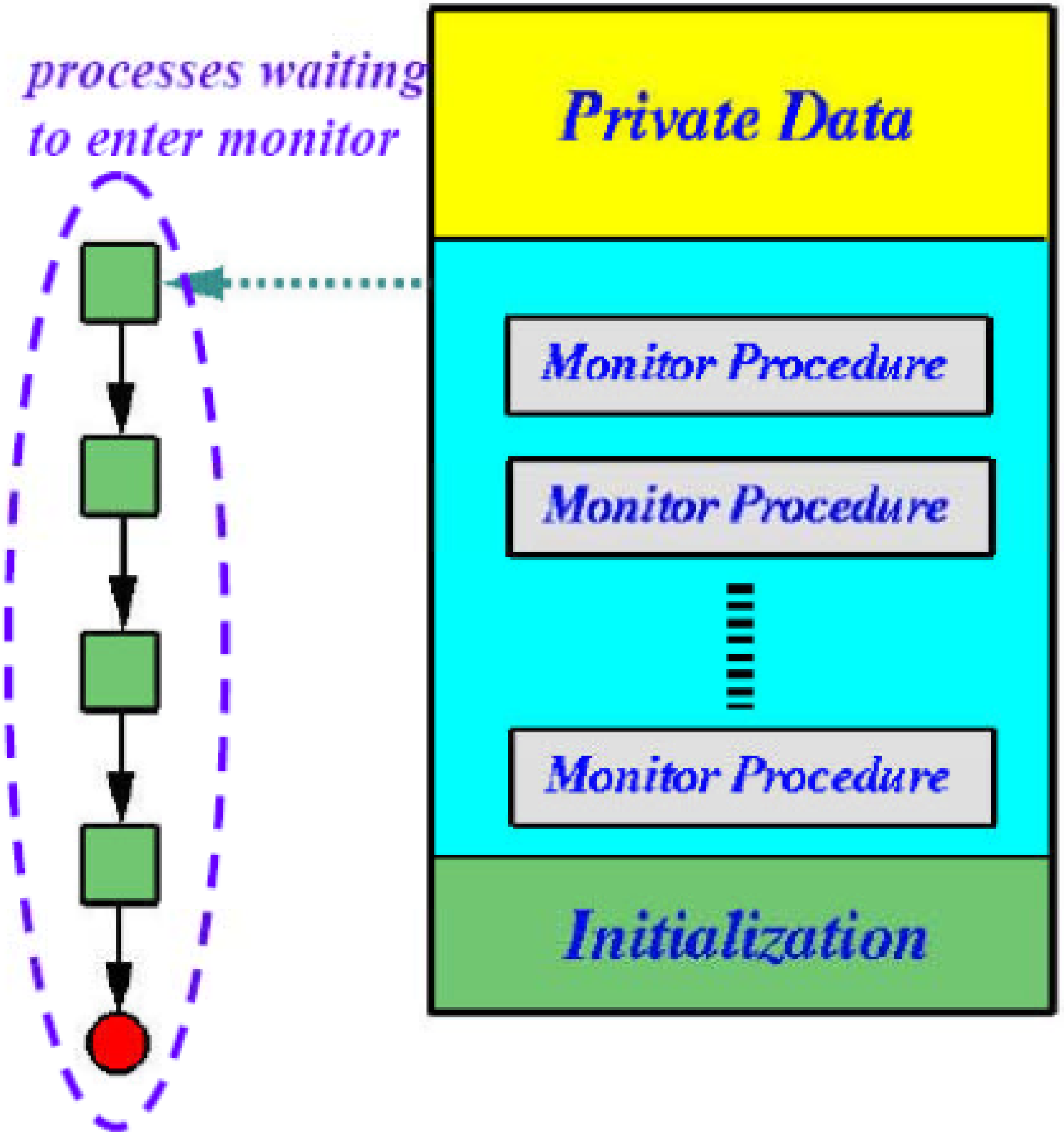
```
monitor monitor-name
{    shared variable declarations
     procedure body P1 (…) {
       . . .}
     procedure body P2 (…) {
       . . .}
     procedure body Pn (…) {
        . . .}
     { initialization code}
}
```

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




processes waiting
to enter monitor
Private Data
Monitor Procedure
Monitor Procedure
Monitor Procedure
Initialization

Monitors

- 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 suspended until another process invokes

    x.signal();

  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

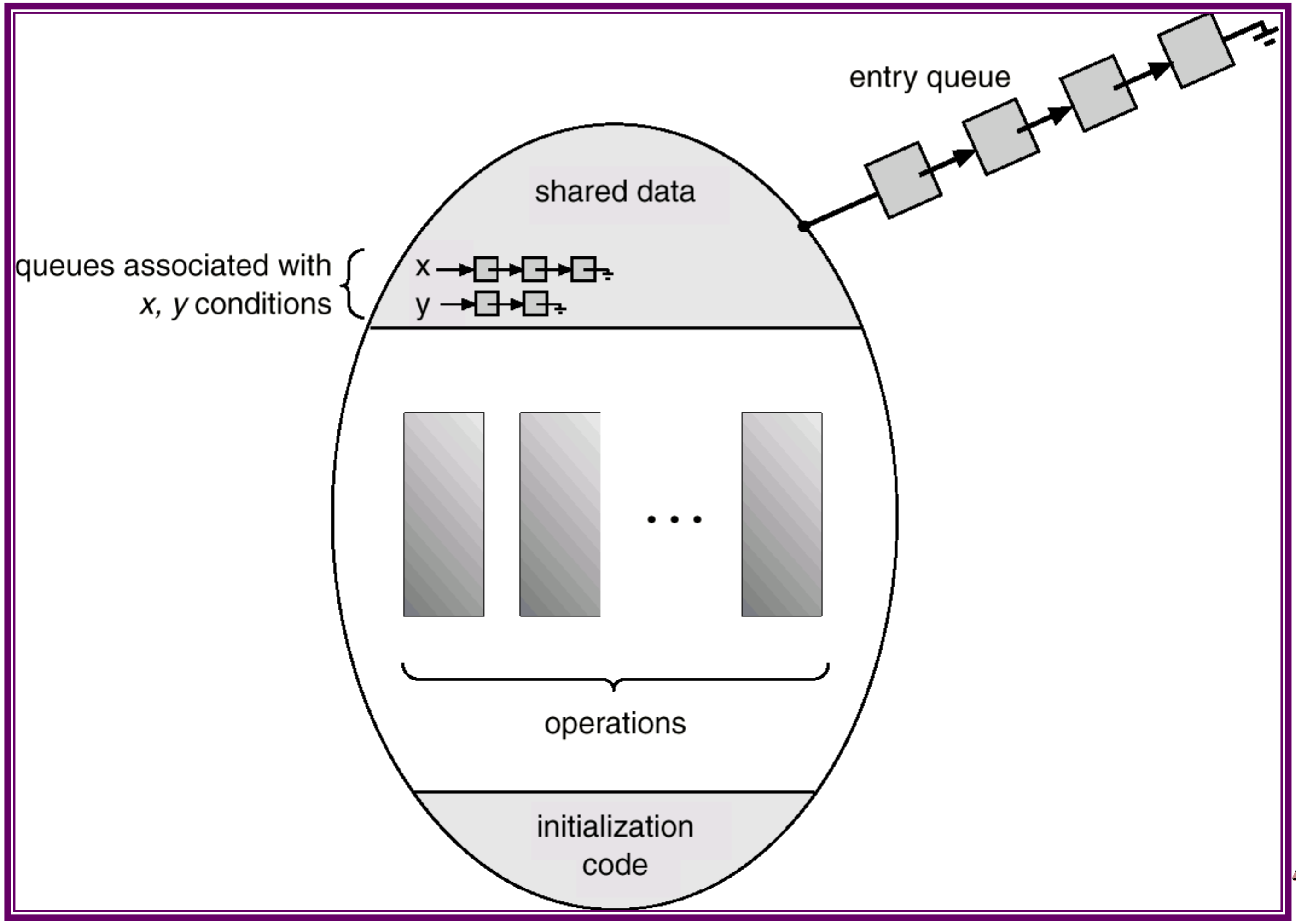
Condition Signal

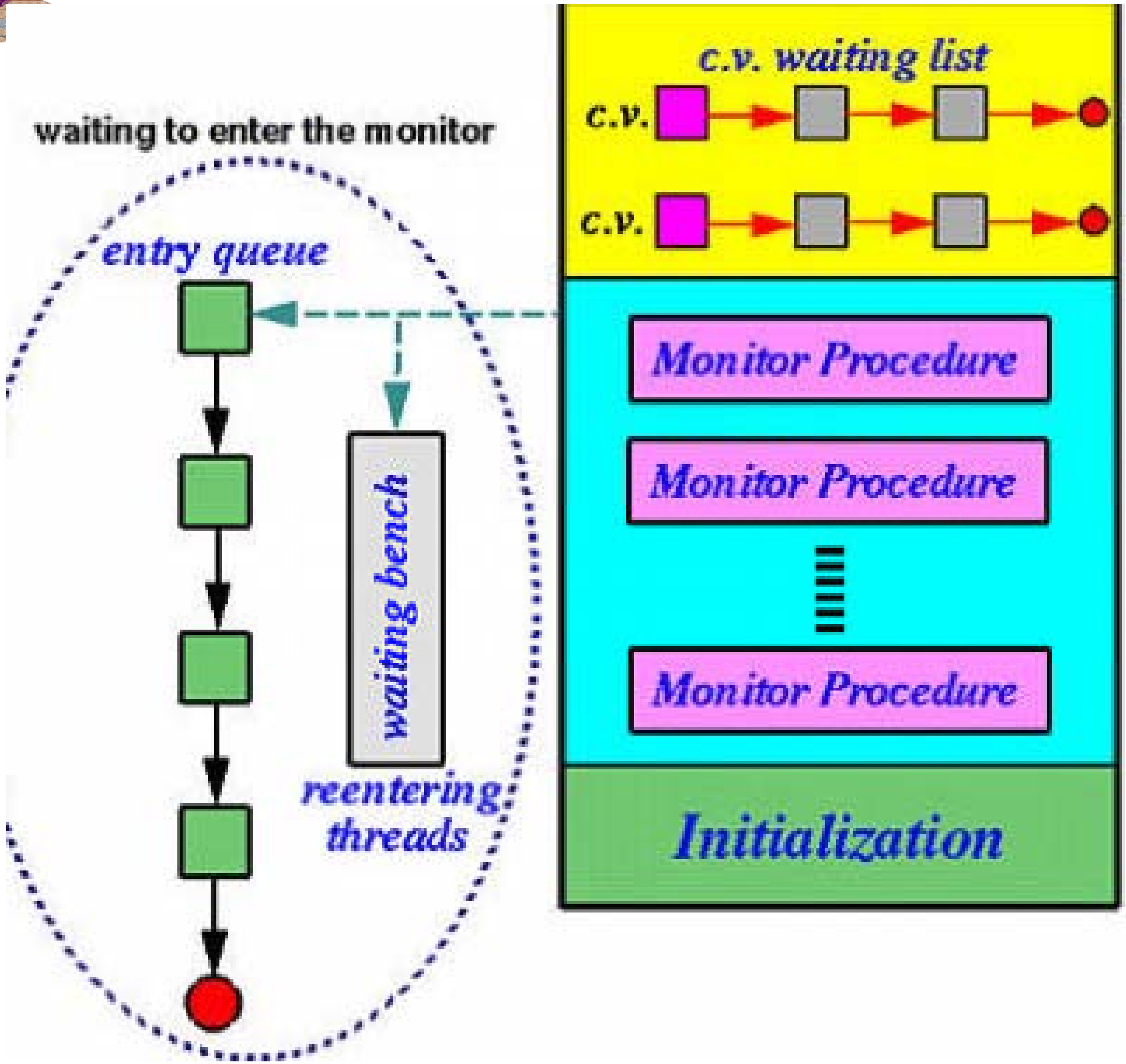
- **Consider the released process (from the signaled condition) and the process that signals. There are two processes executing in the monitor, and mutual exclusion is violated!**
- There are two common and popular approaches to address this problem:
  - The released process takes the monitor and the signaling process waits somewhere.
  - The released process waits somewhere and the signaling process continues to use the monitor.

Semaphore vs. Condition

Semaphores	Condition Variables
Can be used anywhere, but not in a monitor	Can only be used in monitors
`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*

Monitor With Condition Variables




waiting to enter the monitor
entry queue
waiting bench
reentering threads
c.v. waiting list
c.v.
c.v.
Monitor Procedure
Monitor Procedure
Monitor Procedure
Initialization

Dining Philosophers Example

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)                // following slides
  void putdown(int i)    // following slides
  void test(int i)             // following slides
  void init() { for (int i = 0; i < 5; i++)
                               state[i] = thinking;
  }
}
```

Dining Philosophers

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

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

Dining Philosophers

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

Monitor Implementation Using 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

- 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--;
```

Monitor Implementation

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

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

Monitor Implementation

- *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.

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.

Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- 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.

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.