Mutual exclusion

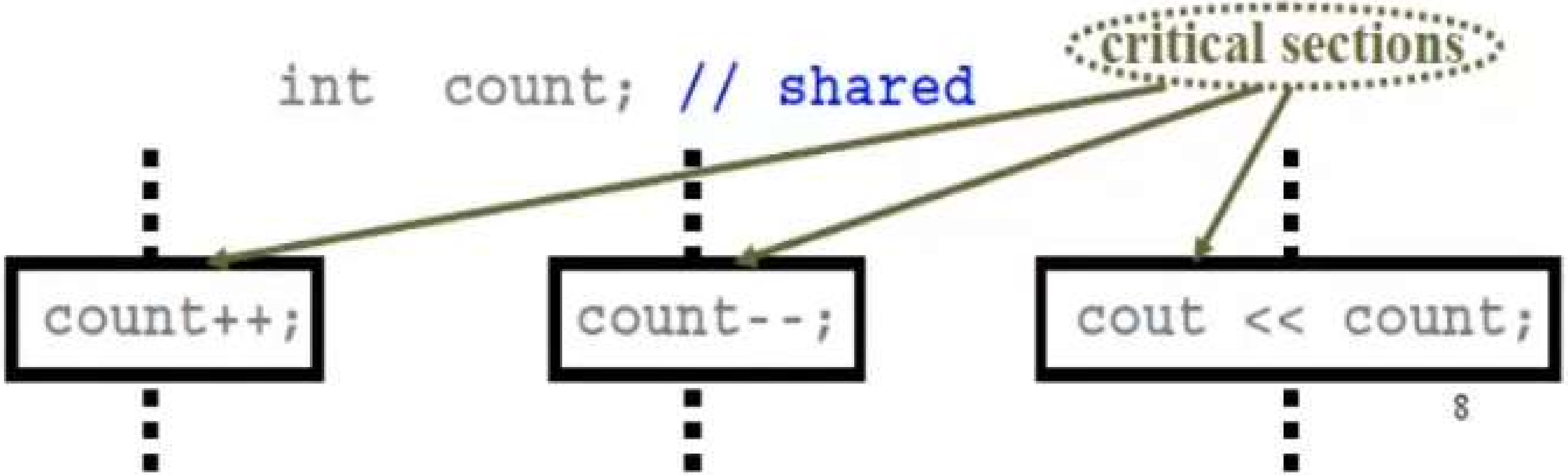
```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

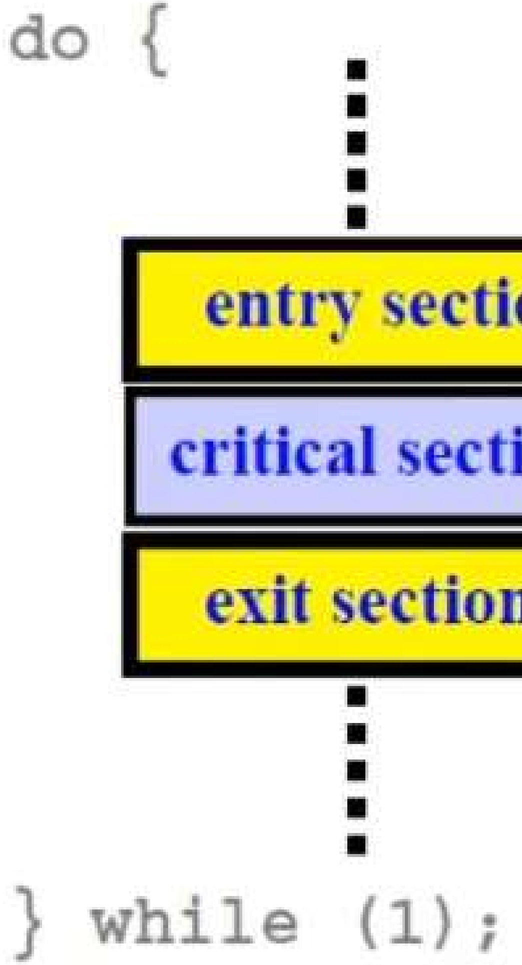
• • •

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

- A *critical section* is a section of code in which a process accesses shared resources.
- 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*** and an ***exit section***.
- **Between them is the critical section that must run in a mutually exclusive way.**

Solutions to the 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 CPU's **relative speed** and **scheduling policy**.

Cooperation among processes by sharing

- Eg:- shared variables/files/database
- Data items may be accessed in reading and writing mode and only the writing mode must be mutually exclusive
- Requirement: data coherence

```
P1:
        a = a + 1;
        b = b + 1;
P2:
        b = 2 * b;
        a = 2 * a;
```

```
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

Cooperation among processes by Communication

- Communication provides a way to synchronize or coordinate the various activities
- This is done by messaging.
- So Mutual exclusion is not a control requirement for this sort of cooperation
- Has deadlock and starvation problems

Requirements of mutual exclusion(ME)

- ME should be enforced
- A process that halts in its noncritical section should not interfere with other processes
- No deadlock and starvation
- When no process is there in the CS, a process requiring CS should be granted permission
- Process remains in CS only for finite time

Ways to arrive at mutual exclusion

- Software approaches
  - Leave the responsibility to the processes that wish to execute concurrently.
  - Disadv is high processing overhead and bugs
- Hardware approaches
  - Special purpose machine instructions
  - Adv of reducing overhead
- Some level of support within the OS or programming language
  - Semaphores
  - monitors

Mutual exclusion : Software approaches

- Can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory
- Peterson’s Algorithm

Software Solutions for Two Processes

- Suppose we have two processes, P_0 and P_1.
- Let one process be P_i and the other be P_j, where $j = 1 - i$. Thus, if $i = 0$ (*resp.*, $i = 1$), then $j = 1$ (*resp.*, $j = 0$).
- We want to design the enter-exit protocol for a critical section so that mutual exclusion is guaranteed.

First Attempt

process P_i

if it is not my turn, I wait

```
do {
    :
    while (turn != i);      enter
        critical section
    turn = j;               exit
    :
} while (1);
```

I am done, it is your turn now

- **Global variable `turn` controls who can enter the critical section.**
- **Since `turn` is either 0 or 1, only one can enter.**
- **However, processes are forced to run in an alternating way.**

Buzy Waiting : waiting process repeatedly reads the value of turn(global memory location) until its allowed to enter its critical section

Disadvantage:- Pace of execution is dictated by the slower of the two processes If one process fails the other process is permanently blocked.

Second attempt

- **Variable `flag[i]` is the "state" of process P_i: *interested* or *not-interested*.**
- **P_i expresses its intention to enter, waits for P_j to exit, enters its section, and finally changes to "I am out" upon exit.**

If one process fails outside the critical section, other process is not blocked. But if it fails within the critical section or after setting the flag then it blocks the other process

Third attempt

```
bool  flag[2];

do {
        flag[i] = TRUE;
        while (flag[j]);
            critical section
        flag[i] = FALSE;
}
```

I am interested

wait for you

enter

exit

I am not interested

- The correctness of this algorithm is *timing dependent*!
- If both processes set `flag[i]` and `flag[j]` to `TRUE` at the same time, then both will be looping at the `while` forever and no one can enter.
- **Bounded waiting does not hold.**

Eliminates the problems in second attempt. This guarantees mutual exclusion but creates deadlock, because each process can insist on its right to enter its critical section.

Livelock

- The process keeps setting and resetting the flags alternatively and gets neither process could enter its critical section.
- Alteration in the relative speed of the two processes will break this cycle and allow one to enter the critical section
- This is called as **livelock**

Peterson's Solution

- Two process solution
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
bool flag[2];    // process Pi
int  turn;

do {

   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);

   critical section

   flag[i] = FALSE;

}
```

I am interested

yield to you first

enter

I am done

exit

wait while you are interested and it is your turn.

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process P_i

- If both processes are in their critical sections, then
  - `flag[j] && turn == j` (P_i) and `flag[i] && turn == i` (P_j) are both `FALSE`.
  - `flag[i]` and `flag[j]` are both `TRUE`
  - Thus, `turn == i` and `turn == j` are `FALSE`.
  - Since `turn` can hold one value, only one of `turn == i` or `turn == j` is `FALSE`, but not both.
  - We have a contradiction and P_i and P_j cannot be in their critical sections at the same time.

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process P_i

- If P_i is waiting to enter, it must be executing its `while` loop.
- Suppose P_j is not in its critical section:
  - If P_j is not interested in entering, `flag[j]` was set to `FALSE` when P_j exits. Thus, P_i may enter.
  - If P_j wishes to enter and sets `flag[j]` to `TRUE`, it will set `turn` to `i` and P_i may enter.
- In both cases, processes that are not waiting do not block the waiting processes from entering.

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process P_i

- When P_i wishes to enter:
  - If P_j is *outside* of its critical section, then `flag[j]` is `FALSE` and P_i may enter.
  - If P_j is *in* its critical section, eventually it will set `flag[j]` to `FALSE` and P_i may enter.
  - If P_j is *in the entry section*, P_i may enter if it reaches `while` first. Otherwise, P_j enters and P_i may enter *after* P_j sets `flag[j]` to `FALSE` and exits.
- Thus, P_i waits at most one round!

Mutual exclusion-Hardware approach

- Interrupt Disabling
- Special machine instructions
  - Compare and swap
  - exchange

Interrupt Disabling

- In an uniprocessor system concurrent processes can have only interleaved execution
- Process runs until its interrupted
- To guarantee ME its enough to prevent a process from being interrupted

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Disadvantages
Degree of interleaving is limited
Does not work in a multiprocessor architecture

Special machine instructions

- Processor designers have proposed several machine instructions that carry out two actions automatically(Read-Write/Read-Test) through a single instruction fetch cycle
- Two such instruction
  - Compare and swap
  - Exchange instruction

Compare and swap

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

1. Checks a memory location (*word) against a test value(testval).
2. If memory location currval=testval, its replaced by newval
3. Otherwise left unchanged

Compare and swap

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

Shared variable bolt is initialized to 0

The only process that can enter critical section is one that finds bolt equal to 0

All other processes would go to buzy waiting mode

Exchange

```
void exchange (int register, int memory)
{
    int   temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Exchange contd…

```
   /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

Shared variable bolt is initialized to zero

Each process has a local variable key that is initialized to 1

Process that find bolt =0 alone enters the critical section

Excludes all other processes by setting bolt=1

Once on exiting it resend bold to 0

If *bolt* = 0, then no process is in its critical section. If *bolt* = 1, then exactly one process is in its critical section, namely the process whose *key* value equals 0.

Properties of machine-instruction approroach

It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.

It is simple and therefore easy to verify.

It can be used to support multiple critical sections; each critical section can be defined by its own variable.

Disadvantage

- Buzy waiting is employed
- Starvation is possible
- Deadlock is possible

Semaphores

PURPOSE:

We want to be able to write more complex constructs and so need a language to do so. We thus define semaphores which we assume are atomic operations:

```
WAIT ( S ):
    while   ( S  <=  0
);
    S  =  S  -  1;
```

```
SIGNAL ( S ):
    S  =  S  +  1;
```

As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

FORMAT:

wait (mutex); <-- Mutual exclusion: mutex init to 1.

CRITICAL SECTION

signal(mutex);

REMAINDER

Semaphores

Semaphores can be used to force synchronization (precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at beginning. For example, here we want P1 to execute before P2.

P1: **P2:**

```
P1:                         P2:
    statement 1;                wait ( synch );
    signal ( synch );           statement 2;
```

Semaphores

We don't want to loop on busy, so will suspend instead:

- Block on semaphore == False,
- Wakeup on signal (semaphore becomes True),
- There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,
- Wakeup one of the blocked processes upon getting a signal (choice of who depends on strategy).

To PREVENT looping, we redefine the semaphore structure as:

```
typedef struct {
  int                     value;
  struct process    *list;      /*  linked list of PTBL waiting on  S  */
} SEMAPHORE;
```

Semaphores

```
typedef struct {
    int         value;
    struct process      *list;      /* linked list of PTBL waiting on  S  */
} SEMAPHORE;
```

```
SEMAPHORE s;
wait(s) {
    s.value = s.value - 1;
    if ( s.value < 0 ) {
     add this process to s.L;
     block;
    }
}
```

```
SEMAPHORE s;
signal(s) {
    s.value = s.value + 1;
    if ( s.value <= 0 ) {
     remove a process P from s.L;
     wakeup(P);
    }
}
```

- It's critical that these be atomic - in uniprocessors we can disable interrupts, but in multiprocessors other mechanisms for atomicity are needed.
- Popular incarnations of semaphores are as "event counts" and "lock managers". (We'll talk about these in the next chapter.)

Semaphores

DEADLOCKS:

May occur when two or more processes try to get the same multiple resources at the same time.

P1:	**P2:**
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
.....	**.....**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

- How can this be fixed?

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

$$
\begin{array}{l}
\texttt{do \{} \\
\quad \boxed{\textit{entry section}} \\
\qquad \text{critical section} \\
\quad \boxed{\textit{exit section}} \\
\qquad \text{remainder section} \\
\text{\} while (true);}
\end{array}
$$

Algorithm for Process P_i

```
do {

    while (turn == j);

        critical section

    turn = j;

        remainder section

    } while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** - allows preemption of process when running in kernel mode
- **Non-preemptive** - runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Process P_i

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

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
 acquire lock
    critical section
 release lock
    remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
} while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
    /* critical section */
 lock = 0;
    /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)

      key = test_and_set(&lock);

   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;

   while ((j != i) && !waiting[j])

      j = (j + 1) % n;

   if (j == i)

      lock = false;

   else

      waiting[j] = false;

   /* remainder section */

} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock

acquire() and release()

- acquire() {

```
while (!available)
    ; /* busy wait */
available = false;
}
```

- release() {

```
available = true;
}
```

- do {

```
acquire lock
   critical section
release lock
  remainder section
} while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore *S* - integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called `P()` and `V()`
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- Counting semaphore - integer value can range over an unrestricted domain
- Binary semaphore - integer value can range only between 0 and 1
  - Same as a mutex lock
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block - place the process invoking the operation on the appropriate waiting queue
  - wakeup - remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

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


signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        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
- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Problems with Semaphores

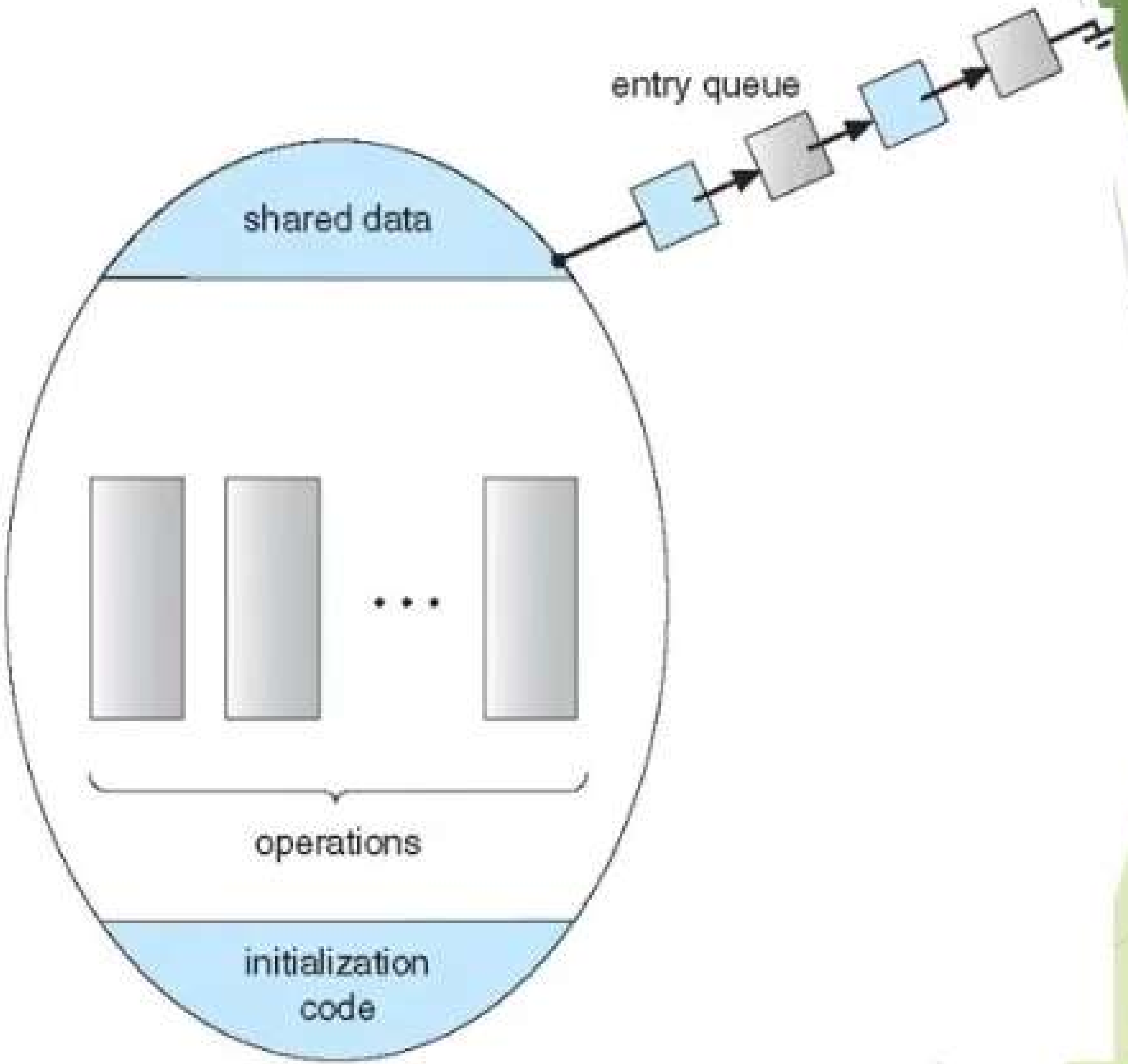
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```

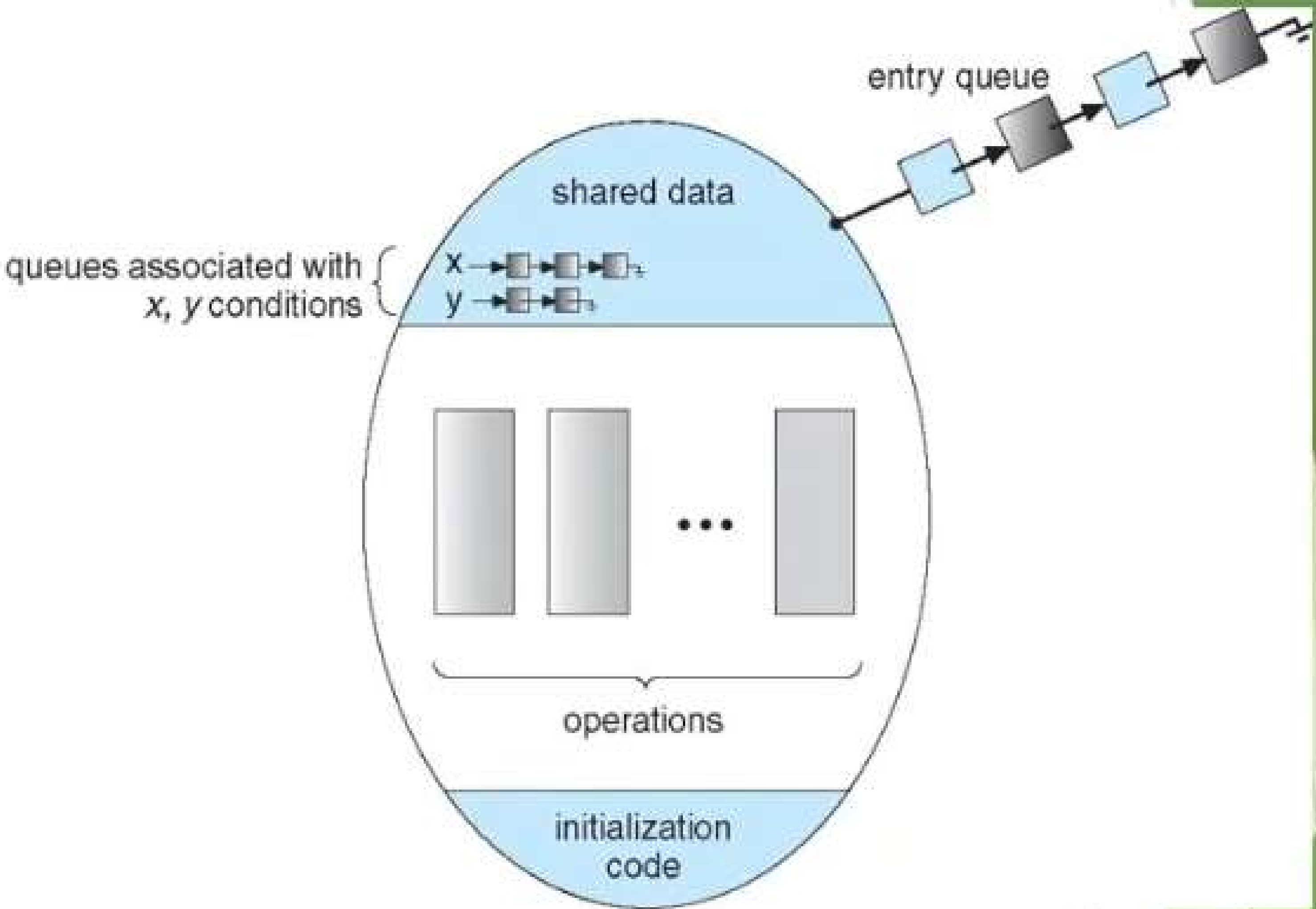
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` - a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` - resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
         /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
         ...
         /* add next produced to the buffer */
         ...
    signal(mutex);
    signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers - only read the data set; they do *not* perform any updates
  - Writers - can both read and write
- Problem - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered - all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
         ...
        /* reading is performed */
         ...
    wait(mutex);
        read count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation - no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation - once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Interprocess Communication - Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication - Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) - send a message to process P
  - **`receive`**(*Q, message*) - receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

`send`(*A*, *message*) - send a message to mailbox A

`receive`(*A*, *message*) - receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;

while (true) {
    /* produce an item in next produced */

send(next_produced);

}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity - no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity - finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity - infinite length
     Sender never waits

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris 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
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **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
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks