7. Process Synchronization

Sungyoung Lee

College of Engineering
KyungHee University

Contents

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most *n* – 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

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

Bounded-Buffer

- Producer process

```
item nextProduced;

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

Bounded-Buffer

- Consumer process

```
item nextConsumed;

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

Bounded Buffer

- The statements

  ```
  counter++;
  counter--;
  ```

  must be performed *atomically*

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

Bounded Buffer

- The statement "**count++**" may be implemented in machine language as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- The statement "**count—**" may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Bounded Buffer

- 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

Bounded Buffer

- 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

Race Condition

- Race condition
  - The situation where several processes access – and manipulate shared data concurrently
  - The final value of the shared data depends upon which process finishes last

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

Synchronization

- Threads cooperate in multithreaded programs
  - To share resources, access shared data structures
  - Also, to coordinate their execution

- For correctness, we have to control this cooperation
  - Must assume threads interleave executions arbitrarily and at different rates
    - Scheduling is not under application writers' control
  - We control cooperation using synchronization
    - Enables us to restrict the interleaving of execution
  - (Note) This also applies to processes, not just threads
    - And it also applies across machines in a distributed system

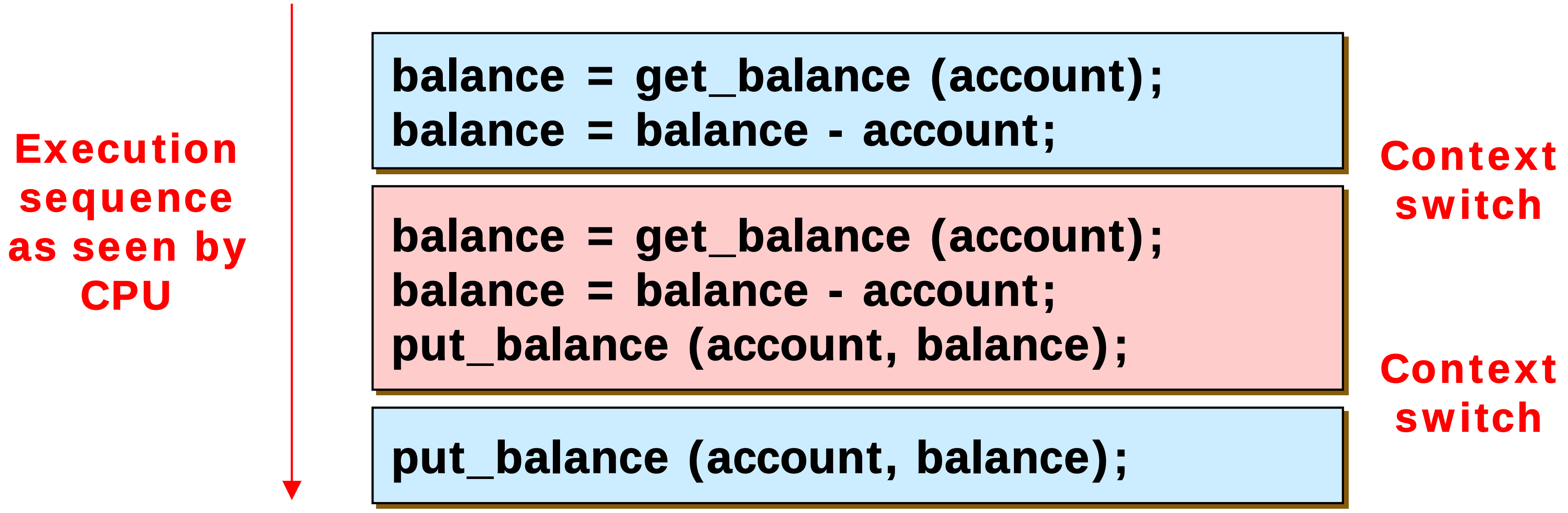
An Example

- Withdraw money from a bank account
  - Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won
  - What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```
int withdraw (account, amount)
{
   balance = get_balance (account);
   balance = balance - amount;
   put_balance (account, balance);
   return balance;
}
```

An Example (Cont'd)

- Interleaved schedules
  - ü Represent the situation by creating a separate thread for each person to do the withdrawals
  - ü The execution of the two threads can be interleaved, assuming preemptive scheduling:



```
balance = get_balance (account);
balance = balance - account;
```

```
balance = get_balance (account);
balance = balance - account;
put_balance (account, balance);
```

```
put_balance (account, balance);
```

Synchronization Problem

- Problem
  - Two concurrent threads (or processes) access a shared resource without any synchronization
  - Creates a **race condition**
    - The situation where several processes access and manipulate shared data concurrently
    - The result is non-deterministic and depends on timing
  - We need mechanisms for controlling access to shared resources in the face of concurrency
    - So that we can reason about the operation of programs
  - Synchronization is necessary for any shared data structure
    - buffers, queues, lists, etc.

The Critical-Section Problem

- *n* 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

Solution to Critical-Section Problem

1. **Mutual Exclusion**
   - ü If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**
   - ü 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**
   - ü 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
   - ü Assume that each process executes at a nonzero speed
   - ü No assumption concerning relative speed of the *n* processes

Mechanisms for Critical Sections

- **Locks**
  - Very primitive, minimal semantics, used to build others
- **Semaphores**
  - Basic, easy to get the hang of, hard to program with
- **Monitors**
  - High-level, requires language support, implicit operations
  - Easy to program with: Java “synchronized”
- **Messages**
  - Simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - Direct application to distributed systems

Locks

- A lock is an object (in memory) that provides the following two operations:
  - acquire(): wait until lock is free, then grab it
  - release(): unlock, and wake up any thread waiting in acquire()

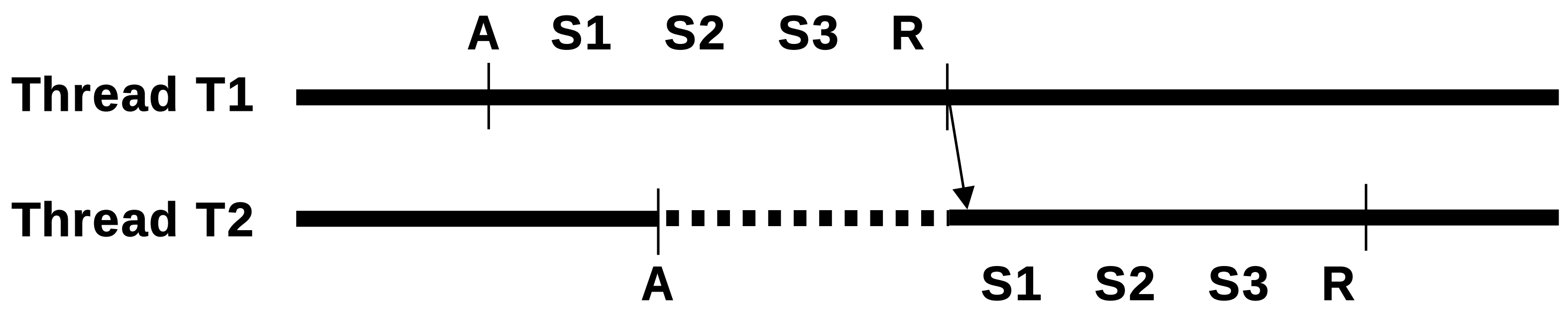
- Using locks
  - Lock is initially free
  - Call acquire() before entering a critical section, and release() after leaving it
  - Between acquire() and release(), the thread holds the lock
  - acquire() does not return until the caller holds the lock
  - At most one thread can hold a lock at a time

- Locks can spin (a spinlock) or block (a mutex)

Using Locks

```
A       int withdraw (account, amount)
        {
S1        acquire (lock);
S2        balance = get_balance (account);
S3        balance = balance - amount;
R         put_balance (account, balance);
          release (lock);
          return balance;
        }
```

Critical section



Implementing Locks

- An initial attempt

```
struct lock { int held = 0; }

void acquire (struct lock *l)  {
   while (l->held);
   l->held = 1;
}
void release (struct lock *l)   {
   l->held = 0;
}
```

The caller "busy-waits", or spins for locks to be released, hence spinlocks

- Does this work?

Implementing Locks (Cont'd)

- Problem
  - Implementation of locks has a critical section, too!
    - The acquire/release must be atomic
    - A recursion, huh?
  - Atomic operation
    - Executes as though it could not be interrupted
    - Code that executes "all or nothing"
- Solutions
  - Software-only algorithms
    - Algorithm 1, 2, 3 for two processes
    - Bakery algorithm for more than two processes
  - Hardware atomic instructions
    - Test-and-set, compare-and-swap, etc.
  - Disable/re-enable interrupts
    - To prevent context switches

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (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

- Entry section
    - Acquire a lock
- Exit section
    - Release a lock

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn = i** ⇒ P_i can enter its critical section

- Process P_i

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

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false**
  - **flag [i] = true** $\Rightarrow$ P_i ready 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

Algorithm 3

- Combined shared variables of algorithms 1 and 2

- Process P_i

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

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

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 <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ 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);
```

Synchronization Hardware

- 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
} while (1);
```

Synchronization Hardware

- 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;
```

- Process P_i

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

Problems with Spinlocks

- Horribly wasteful !
  - If a thread is spinning on a lock, the thread holding the lock cannot make progress
  - the longer the critical section, the longer the spin
  - Greater the chances for lock holder to be interrupted

- How did the lock holder yield the CPU in the first place?
  - Lock holder calls yield() or sleep()
  - Involuntary context switch

- Only want to use spinlock as primitives to build higher-level synchronization constructs

Disabling Interrupts

- Implementing locks by disabling interrupts

```
void acquire (struct lock *l)  {
    cli();          // disable interrupts;
}
void release (struct lock *l)  {
    sti();          // enable interrupts;
}
```

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- There is no state associate with the lock
- Can two threads disable interrupts simultaneously?

Disabling Interrupts (Cont'd)

- What's wrong?
  - Only available to kernel
    - Why not have the OS support these as system calls?
  - Insufficient on a multiprocessor
    - Back to atomic instructions
  - What if the critical section is long?
    - Can miss or delay important events (e.g., timer, I/O)
  - Like spinlocks, only use to implement higher-level synchronization primitives

Higher-level Synchronization

- Motivation
  - Spinlocks and disabling interrupts are useful only for very short and simple critical sections
    - Wasteful otherwise
    - These primitives are “primitive” – don’t do anything besides mutual exclusion
  - Need higher-level synchronization primitives that
    - Block waiters
    - Leave interrupts enabled within the critical section
  - Two common high-level primitives:
    - Semaphores: binary (mutex) and counting
    - Monitors: mutexes and condition variables
  - We’ll use our “atomic” locks as primitives to implement them

Semaphores

n Synchronization tool that does not require busy waiting

n Semaphore *S* = integer variable

n can only be accessed via two indivisible (atomic) operations

```
wait (S):
        while S£ 0 do no-op;
          S--;

signal (S):
        S++;
```

Critical Section of *n* Processes

- Shared data:

  semaphore mutex; *//*initially *mutex* = 1

- Process *Pi:*

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

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***

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);
        }
```

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:

```
      Pi                     Pj
      M                      M
      A                 wait(flag)
 signal(flag)                B
```

Deadlock and Starvation

n **Deadlock**

ü two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

n Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
M	M
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

n **Starvation** or indefinite blocking

ü A process may never be removed from the semaphore queue in which it is suspended

Two Types of Semaphores

- *Counting* semaphore
  - integer value can range over an unrestricted domain
- *Binary* semaphore
  - integer value can range only between 0 and 1
  - can be simpler to implement
- Can implement a counting semaphore *S* as a binary semaphore

Implementing *S* as a Binary Semaphore

- Data structures:

  binary-semaphore S1, S2;

  int C;

- Initialization:

  S1 = 1 // for mutual exclusion of C

  S2 = 0

  C = initial value of semaphore **S**

Implementing *S*

- *wait* operation

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

- *signal* operation

```
wait(S1);
C ++;
if (C <= 0)
        signal(S2);
else
        signal(S1);
```

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);
```

Bounded Buffer Problem

- No synchronization

Producer

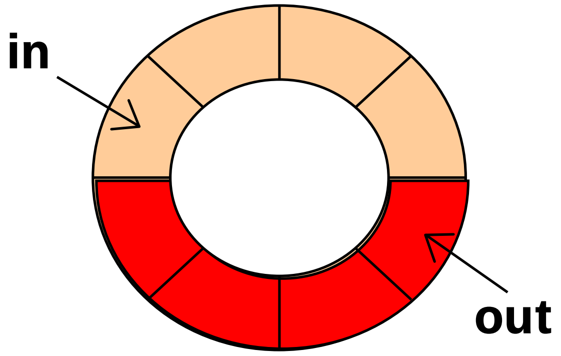
```
void produce(data)
{

   while (count==N) ;
   buffer[in] = data;
   in = (in+1) % N;
   count++;

}
```

Consumer

```
void consume(data)
{

   while (counter==0) ;
   data = buffer[out];
   out = (out+1) % N;
   count--;

}
```

```
int count;
```

```
struct item buffer[N];
int in, out;
```



Bounded Buffer Problem (Cont'd)

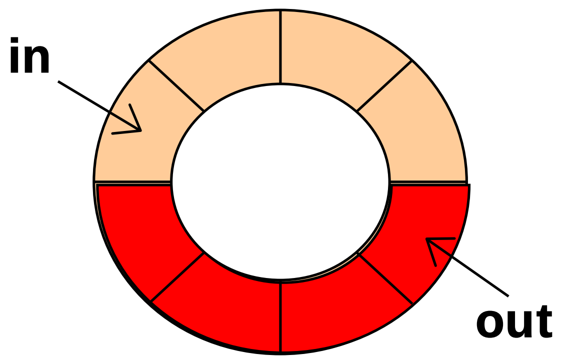
- Implementation with semaphores

```
Semaphore
    mutex = 1;
    empty = N;
    full = 0;
```

Producer

```
void produce(data)
{

   wait (empty);
   wait (mutex);
   buffer[in] = data;
   in = (in+1) % N;
   signal (mutex);
   signal (full);

}
```

```
struct item buffer[N];
int in, out;
```



Consumer

```
void consume(data)
{

   wait (full);
   wait (mutex);
   data = buffer[out];
   out = (out+1) % N;
   signal (mutex);
   signal (empty);

}
```

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):
```

Readers-Writers Problem

- Readers-Writers problem
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers at a time
  - We can only allow one writer at a time
  - Two cases
    - No reader should wait for other readers to finish simply because a writer is waiting
    - Once a writer is ready, that writer performs its write ASAP

- Implementation with semaphores
  - readcount - # of threads reading object
  - mutex – control access to readcount
  - wrt – exclusive writing or reading

Readers-Writers Problem (Cont'd)

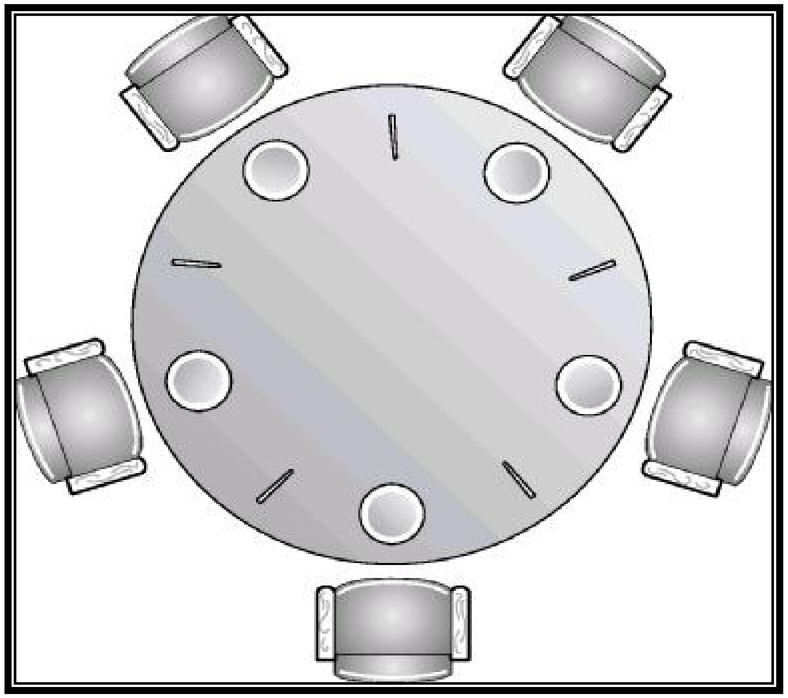
```
// number of readers
int readcount = 0;
// mutex for readcount
Semaphore mutex = 1;
// mutex for reading/writing
Semaphore wrt = 1;

void Writer ()
{
   wait (wrt);
   …
   Write
   …
   signal (wrt);
}
```

```
void Reader ()
{
   wait (mutex);
   readcount++;
   if (readcount == 1)
      wait (wrt);
   signal (mutex);
   …
   Read
   …
   wait (mutex);
   readcount--;
   if (readcount == 0)
      signal (wrt);
   signal (mutex);
}
```

Readers-Writers Problem (Cont'd)

- If there is a writer
  - The first reader blocks on wrt
  - All other readers will then block on mutex
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
- The last reader to exit signals waiting writer
  - Can new readers get in while writer is waiting?
- When writers exits, if there is both a reader and writer waiting, which one goes next is up to scheduler

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);
```

Dining Philosopher

- Dining philosopher problem
  - Dijkstra, 1965
  - Life of a philosopher: Repeat forever
    - Thinking
    - Getting hungry
    - Getting two chopsticks
    - Eating

Dining Philosopher (Cont'd)

- A simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
    while (1)  {
        think ();
        wait (chopstick[i]);
        wait (chopstick[(i+1) % N];
        eat ();
        signal (chopstick[i]);
        signal (chopstick[(i+1) % N];
    }
}
```

$\Rightarrow$ **Problem: causes deadlock**

Dining Philosopher (Cont'd)

- Deadlock-free version: starvation?

```
#define N        5
#define L(i)     ((i+N-1)%N)
#define R(i)     ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
 }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

Problems with Semaphores

- Drawbacks
  - They are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - There is no connection between the semaphore and the data being controlled by it
  - Used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - No control over their use, no guarantee of proper usage

- Thus, hard to use and prone to bugs
  - Another approach: use programming language support
    - Critical region
    - Monitor

Critical Regions

- High-level synchronization construct
- A shared variable ***v*** of type ***T***, is declared as:

  v: shared T

- Variable ***v*** accessed only inside statement

  region v when B do S

  where ***B*** is a boolean expression
- While statement ***S*** is being executed, no other process can access variable ***v***

Critical Regions

- Regions referring to the same shared variable exclude each other in time
- When a process tries to execute the region statement, the Boolean expression *B* is evaluated
    - If *B* is true, statement *S* is executed
    - If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*

Example – Bounded Buffer

- Shared data:

```
struct buffer {
        int pool[n];
        int count, in, out;
}
```

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {
        pool[in] = nextp;
        in:= (in+1) % n;
        count++;
}
```

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in **nextc**

```
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```

Implementation region *x* when *B* do *S*

- Associate with the shared variable *x*, the following variables:

  semaphore mutex, first-delay, second-delay;
  int first-count, second-count;

- Mutually exclusive access to the critical section is provided by **mutex**

- If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate *B*

Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively

- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore

- For an arbitrary queuing discipline, a more complicated implementation is required

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

- A programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Allows the safe sharing of an abstract data type among concurrent processes

- A monitor is a software module that encapsulates
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures

- Monitor protects the data from unstructured access
  - guarantees only access data through procedures, hence in legitimate ways

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

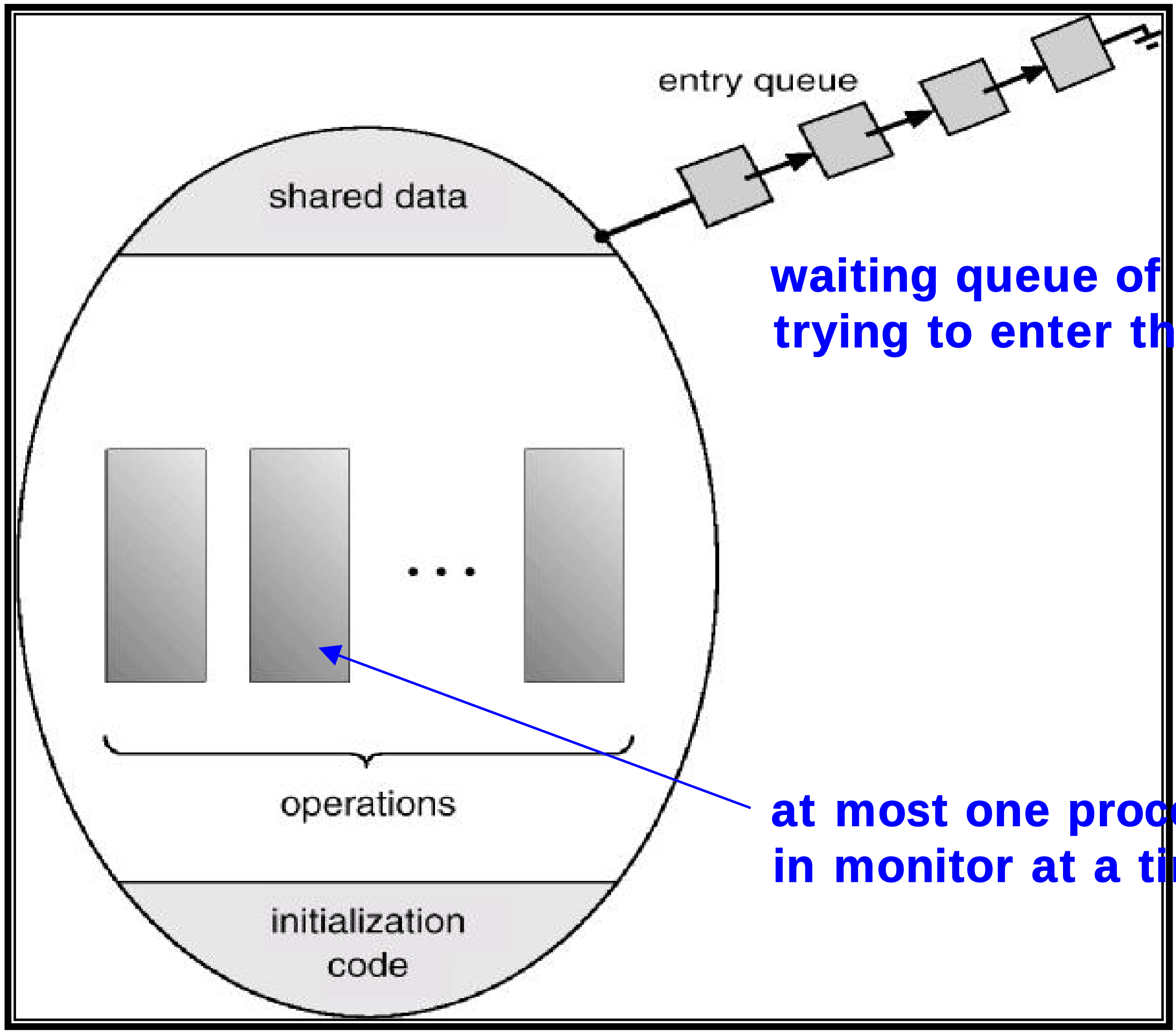
    x.wait();

    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 variable
  - provides a mechanism to wait for events (a “rendezvous point”)

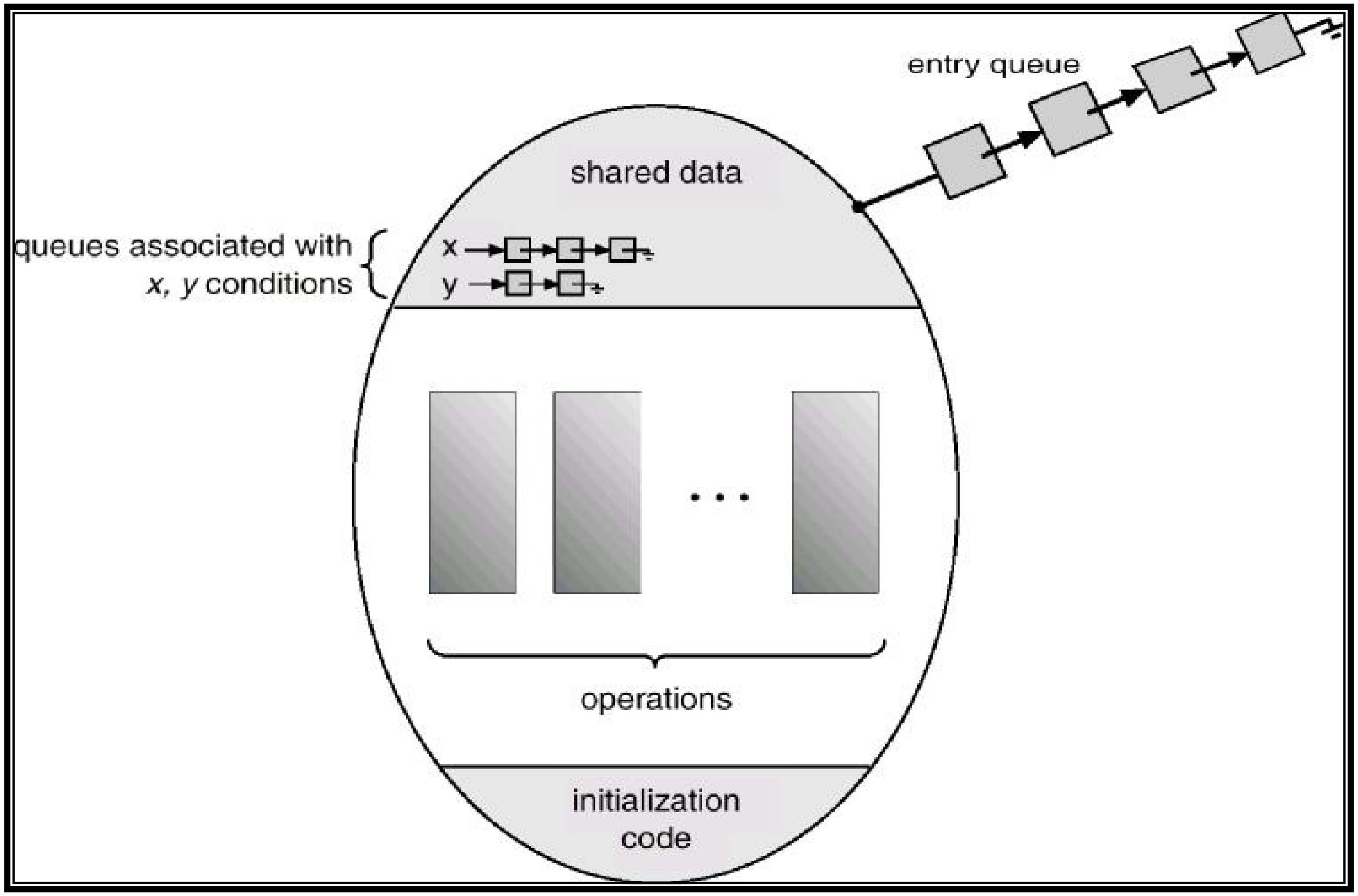
Schematic View of a Monitor



waiting queue of processes trying to enter the monitor

at most one process in monitor at a time

Monitor With Condition Variables



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();
        }
}
```

Monitors Semantics

- Hoare monitors
  - signal(c) immediately switches from the caller to a waiting thread, blocking the caller
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - Signaler must restore monitor invariants before signaling

- Mesa monitors
  - signal(c) places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
    - Being woken up is only a hint that something has changed
    - Must recheck conditional case

Hoare Monitor Implementation

```
monitor ResourceAllocation  {
        boolean busy;
        condition x;

        void acquire(int time)  {
                if (busy) x.wait(time);
                busy = true;
        }
        void release()  {
                busy = false;
                x.signal();
        }
        void init()  {
                busy = false;
        }
}
```

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

- 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

Comparison: Monitors and Semaphores

- Condition variables do not have any history, but semaphores do
  - On a condition variable signal(), if no one is waiting , the signal is a no-op

    (If a thread then does a condition variable wait(), it waits)
  - On a semaphore signal(), if no one is waiting, the value of the semaphore is increased

    (If a thread then does a semaphore wait(), the value is decreased and the thread continues)

Synchronization in Pthreads

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void producer (resource x)  {
   pthread_mutex_lock (&mutex);
   while (array "resources" is full)
      pthread_cond_wait (&not_full, &mutex);
   add "x" to array "resources";
   pthread_cond_signal (&not_empty);
   pthread_mutex_unlock (&mutex);
}
void consumer (resource *x)  {
   pthread_mutex_lock (&mutex);
   while (array "resources" is empty)
      pthread_cond_wait (&not_empty, &mutex);
   *x = get resource from array "resources"
   pthread_cond_signal (&not_full);
   pthread_mutex_unlock (&mutex);
}
```

Synchronization Mechanisms

- Disabling interrupts
- Spinlocks
  - Busy waiting
- Semaphores
  - Binary semaphore = mutex (≅ lock)
  - Counting semaphore
- Monitors
  - Language construct with condition variables
- Mutex + Condition variables
  - Pthreads

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* 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 2000 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