Process Synchronization

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

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Bounded-Buffer

- Shared data

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

Bounded-Buffer

- Producer process

```
item nextProduced;
    while (1) {
            while (counter == BUFFER_SIZE)
                    ; /* do nothing */
            insert_item(nextProduced);
            counter++;
    }
```

Bounded-Buffer

- Consumer process

```
item nextConsumed;
   while (1) {
         while (counter == 0)
                  ; /* do nothing */
         nextConsumed =remove_item();
         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**.

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**. 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 process that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. No process should have to wait forever to enter its critical region
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

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.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn = i** $\Rightarrow P_i$ 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
  - Consider the case when process P0 finishes its critical section: turn=1.
  - Suppose that P0 needs to enter its critical region again.
  - It has to wait for P1, which may be indefinitely!

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ 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.
  - Suppose that P0 executes flag[0]=true and then a context switch occurs: P1 starts executing, sets flag[1]=true.
  - At this point flag[0]=flag[1]=true and the two processes will loop forever.

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.

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
}
```

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;
boolean waiting[n];
```

- Process P_i

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

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 *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

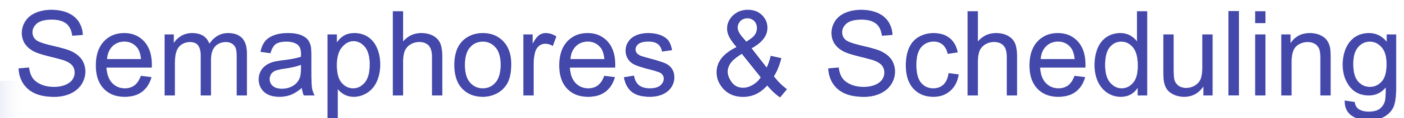
- Semaphore operations now defined as

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);
}
```



Semaphores & Scheduling

- One simple implementation would be to wake-up the processes in a first come first serve basis.
- Is it optimal?
- Suppose that processes P1,P2 and P3 are waiting on a semaphore with P3 having the highest priority.
- In FCFS P3 has to wait for P1 and P2 to finish.

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*

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);
M	M
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.

Multiple Access To Resources

- Semaphores can be used in a more general way.
- Suppose that a system has two printers.
- Processes P1, P2 and P3 need to printer and don't care which printer to use.
- Define:

semaphore p=2

- Assume that they request the printer in order then

P1:

wait(p); /* p=1 , doesn't block*/

P2:

wait(p); /* p=0, doesn't block */

P3:

wait(p); /* p=-1, blocks */

Classical Problems of Synchronization

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

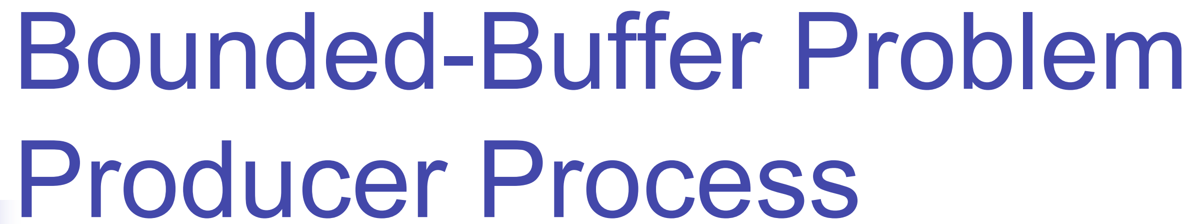
Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1



Bounded-Buffer Problem Producer Process

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

Bounded-Buffer Problem Consumer Process

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

Readers-Writers Problem

- Shared data accessed by many processes.
- Some processes read and some write.
- While a process is writing no other process can read or write.
- If a process is reading other processes can read also
- Typical scenario: database access.

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0
- **mutex used for exclusive access to readcount.**
- **wrt used for exclusive access to file: writer or many readers**

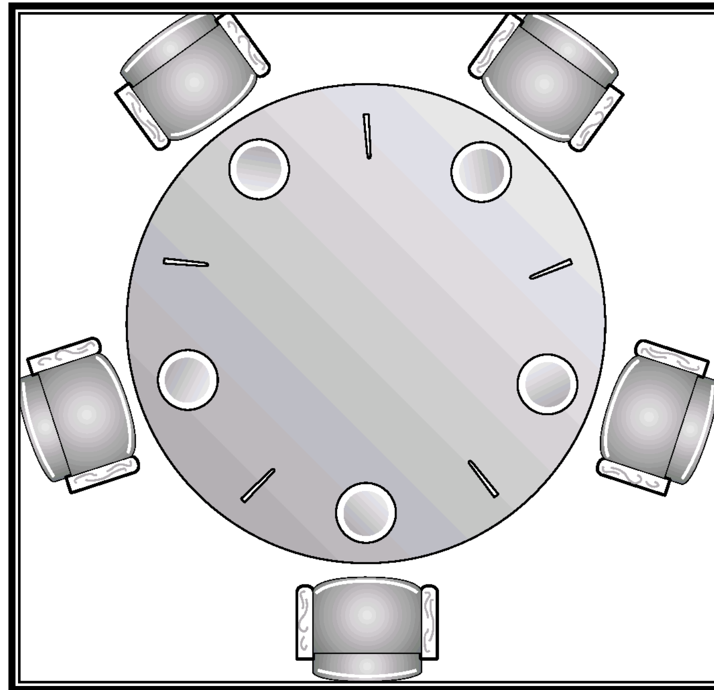
Readers-Writers Problem Writer Process

```
wait(wrt);
        ...
    writing is performed
    no writers or readers can access file
        ...
signal(wrt);
```

Readers-Writers Problem Reader Process

```
wait(mutex);/* get exclusive access to readcount */
readcount++;
if (readcount == 1)/* first reader block writers */
      wait(wrt);
signal(mutex);/* release lock on readcount */
      ...
reading is performed
      ...
wait(mutex);/* get exclusive access to readcount */
readcount--;
if (readcount == 0)
      signal(wrt);/* last reader unblock writing */
signal(mutex);/* release lock on readcount */
```

Dining-Philosophers Problem



- Each philosopher needs 2 chopsticks to eat. Shared data

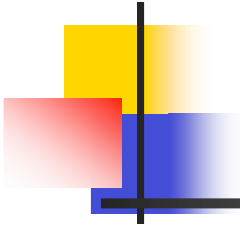
semaphore chopstick[5];

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);
```



- Suppose that all five philosophers acquire their left chopsticks simultaneously. What happens in that case ?
- A deadlock will occurs since no philosopher can acquire a right chopstick to eat.

Solution 1

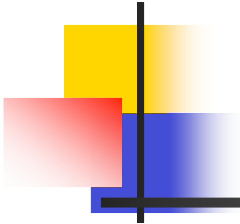
- Suppose that we require that after acquiring a left chopstick a philosopher checks to see if the right is available.
- If not he releases the left chopstick.
- This could lead to starvation if all five philosophers are in sync.

Solution 2

- A solution to the previous problem is to protect the action of picking up a chopstick by a semaphore.
- If a philosopher acquires the mutex no other philosopher can pick up a left chopstick.

Philosopher *i*:

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



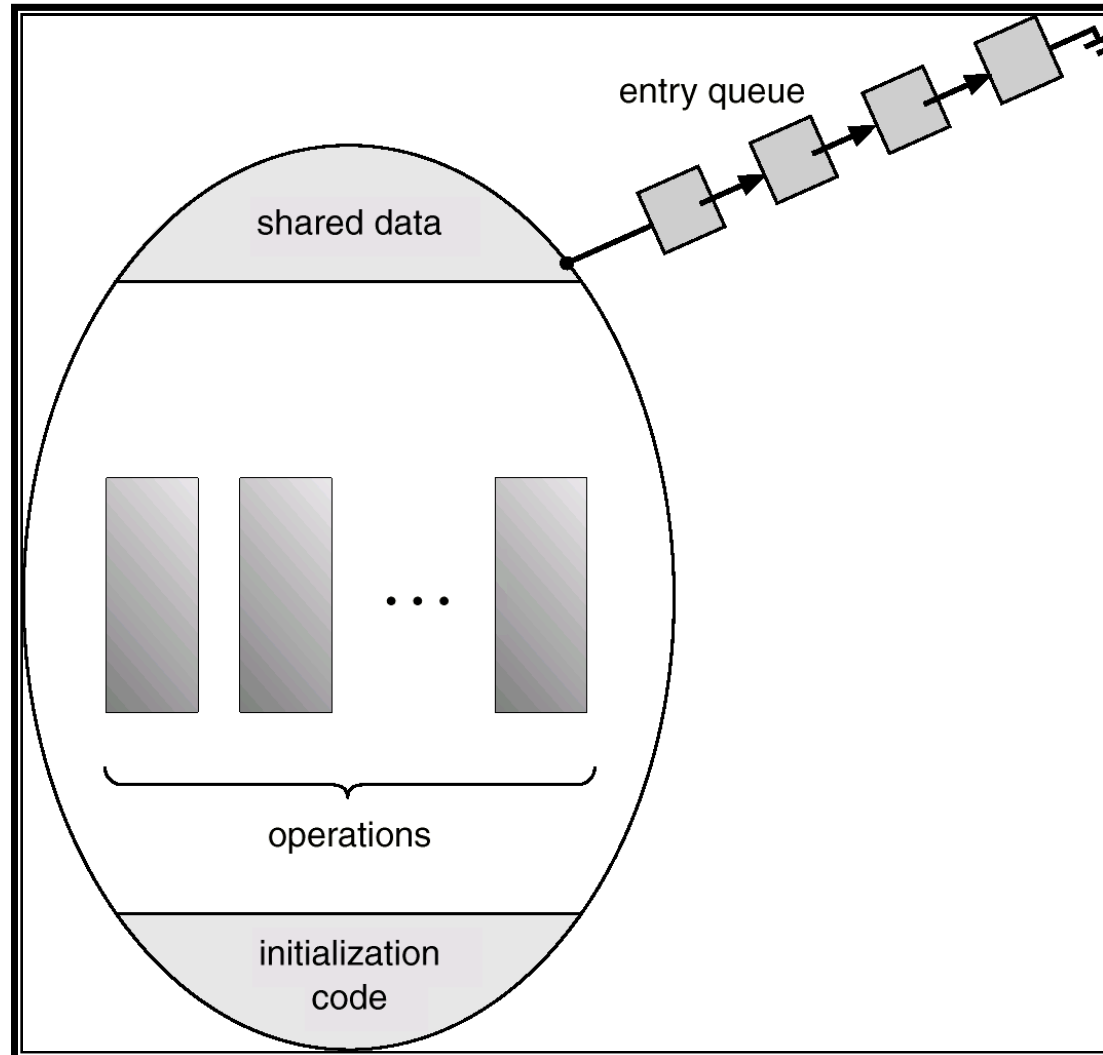
- While the previous solution works, only one philosopher can be eating.
- With five chopsticks it should be possible for 2 philosopher eating simultaneously.
- We will give a solution using monitors.

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- A monitor prohibits concurrent access to all procedures defined within the monitor.

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

Schematic View of a Monitor



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

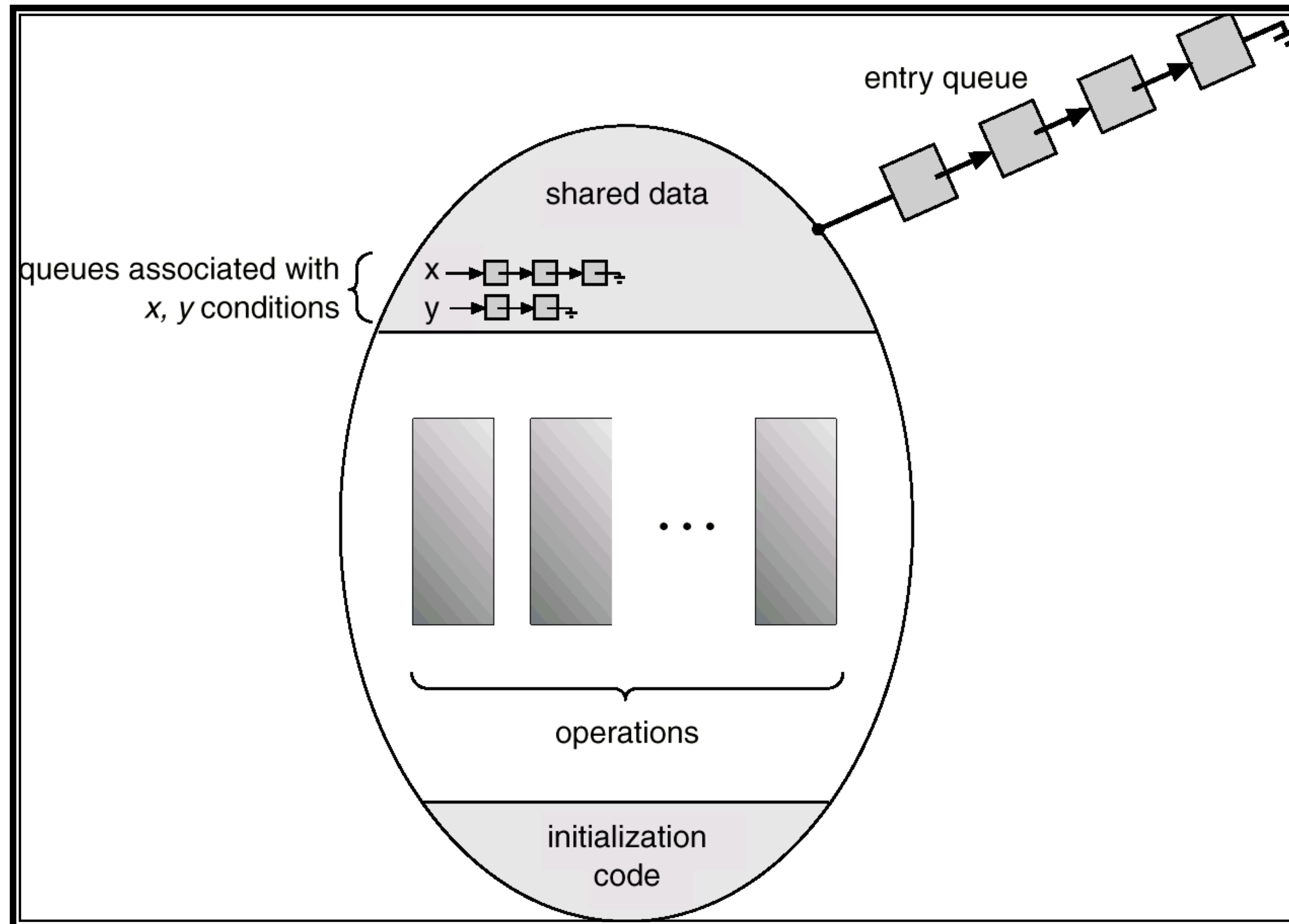
    x.wait();

    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.

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();
    }
}
```