

1 – Synchronization Mechanisms

List of Slides

- 1 Synchronization Mechanisms
- 4 Administrative Notes
- 5 Semaphores - The Data Structure
- 6 Semaphores - Operations
- 7 Semaphores - Usage
- 8 Synchronization Using Semaphores
- 9 Semaphores — Mailbox Problem
- 10 Semaphores — Mailbox Solution
- 11 Producer Consumer Problem — Busy Waiting
- 12 Bounded Buffer Problem — Introduced
- 13 Bounded Buffer Problem — Semantics
- 14 Bounded Buffer Problem — Semantics

- 15 Bounded Buffer Problem — Counting Semaphore Approach
- 16 Bounded Buffer Problem — Counting Semaphore Solution
- 17 Dining Philosopher's Problem
- 19 Dining Philosopher's Psuedocode
- 20 A Semaphore Based Approach — 1st try
- 21 A Semaphore Based Approach — 2nd try
- 22 A Semaphore Based Solution
- 23 A Solution for the Dining Philosopher's Problem
- 24 Readers Writers Problem
- 25 A Weak Reader's Preference Solution
- 26 Some Useful Psuedocode Notation
- 27 Steps to Deriving Mutual Exclusion Solutions
- 28 Reader Writer Outline
- 29 Passing the Baton

- 30 Passing the Baton — SIGNAL Defined
- 31 Reader Writer Solution (Passing the Baton)
- 32 Reader Writer Solution (Passing the Baton)
- 33 Reader Writer Solution (Passing the Baton)
- 34 Reader Writer Problem Notes
- 35 Reader Writer Problem — Writer's Preference
- 37 General Notes on Semaphores
- 38 Monitors
- 39 Monitors

2 – Administrative Notes

Exams are graded.

3 – Semaphores - The Data Structure

Dijkstra's semaphores are a sort of synchronized counter. They are the most frequently available/used synchronization mechanism. Let S denote a semaphore, S contains:

1. *count* — How many processes can call wait before being blocked on S .
2. *queue* — The list of processes waiting on S . It can be shown that if *queue* is FIFO then the semaphores are *fair* (usually assumed), otherwise they are called *weak*. We restrict our consideration to fair semaphores for the remainder of this course.

For *counting semaphores*, *count* is allowed to have non negative integer values.

For *binary semaphores*, $count \in \{0, 1\}$.

4 – Semaphores - Operations

The following operations are available:

1. *Wait(s)* — Sometimes called $P(S)$.

```
atomic void wait(S){
    if (s == 0){ /* Should I Block? */
        Insert this process in s.queue
        block this process
    }
    --s.count;
}
```

2. *signal(s)* — Also called $V(S)$ or *post(S)*.

```
atomic void signal(S){
    s.count++;
    if (s.count <= 0){ /* Any blocked ? */
        remove process p from s.queue
        place p in the ready to run list
    }
}
```

5 – Semaphores - Usage

Typically the solution consists of bracketing the critical sections with wait and signal calls.

Suppose there are n processes with critical sections, and $0 \leq i < n$, then for process i :

```
semaphore mutex = {k, empty};
void process_i(){
    while (not done) do {
        wait(mutex);
        <Critical Section>
        signal(mutex);
        <Non Critical Section>
    }
}
```

6 – Synchronization Using Semaphores

The following techniques are used to semaphore based solutions:

1. Use a separate semaphore for each condition requiring synchronization.
2. When a condition is true, the process should *signal* the corresponding semaphore.
3. When a process blocks pending a condition becoming true, the process should *wait* on the corresponding semaphore.
4. Semaphores are suited for resource allocation problems:
 - (a) Initialize the Semaphore to the number of available resource units.
 - (b) *Wait* requests a unit if none are available.
 - (c) *Signal* releases a unit (or indicates creation).

7 – Semaphores — Mailbox Problem

Consider a system with a single mailbox which has room for just 1 message, with the processes,

- *sender* generating messages and putting them in the mailbox
- *receiver* removes messages from the mailbox and process them

Can you derive a solution for this problem?

8 – Semaphores — Mailbox Solution

The conditions causing blocking are the mailbox being full or empty:

- semaphore empty — Initial value = 1
- semaphore full — Initial value = 0

```
void sender(){
loop
    wait(empty)
    put mesg in mbox
    signal(full)
end loop
}
```

```
void receiver{
texttloop
    wait(full)
    get mesg from mbox
    signal(empty)
end loop
}
```

9 – Producer Consumer Problem — Busy Waiting

Let's consider the following producer consumer problem with $n = \infty$ buffers:

producer	consumer
repeat produce item x $b[in] := x$ $in ++$ forever	repeat while $in < out$ do nothing $y = b[out]$ $out ++$ consume y forever

Table 1: Producer Consumer Busy Waiting Solution, $n = \infty$

10 – Bounded Buffer Problem — Introduced

The bounded buffer problem is an instance of the producer consumer problem using a finite number of buffers.

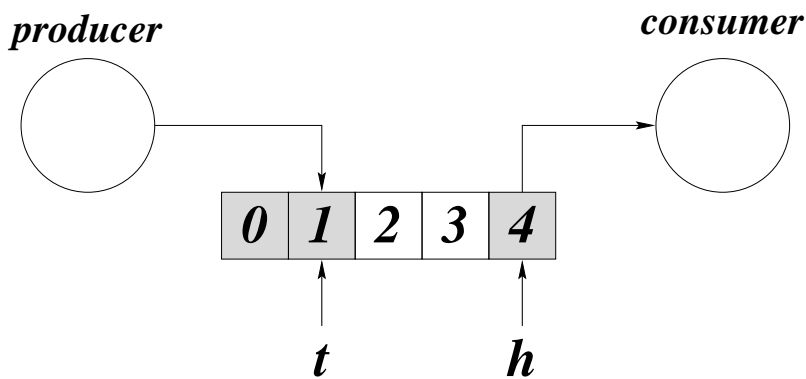


Figure 1: A Bounded Buffer Example, $n = 5$, $c = 3$, $h = 1$

11 – Bounded Buffer Problem — Semantics

Assume that we use a circular buffer array, b , containing n buffers. Outside of the critical section, the following invariants will be satisfied:

- Let h be the head of the queue, $0 \leq h < n$.
- Let c be the count (number) of items in the queue, so: $0 \leq c \leq n$.
- Let t be the tail of the queue, so: $0 \leq t < n$ and $t = (h + c - 1) \bmod n$.

12 – Bounded Buffer Problem — Semantics

Operations on b include:

- $empty(b)$ — returns $c == 0$
- $full(b)$ — returns $c == n$
- $insert(b, i)$ — inserts i into b :
 - Precondition : $\neg full(b)$.
 - action : $t := t + 1 \bmod n; b[t] = i; c ++;$
 - Postcondition : $\neg empty(b)$.
- $remove(b, i)$ — removes i from b , fails if
 - Precondition : $\neg empty(b)$.
 - action : $i = b[h]; h = (h + 1) \bmod n; c --;$
 - Postcondition : $\neg full(b)$.

13 – Bounded Buffer Problem — Counting Semaphores

We introduce the following counting semaphores:

- *nempty* — number of empty buffers, initially $nempty.count = n$.
- *nfull* — number of full buffers, initially $nfull.count = 0$.
- *mutex* — number of processes allowed to concurrently access the queue internal structures, initially $mutex.count = 1$. Needed to ensure atomic queue manipulations.

14 – Bounded Buffer Problem — Counting Semaphores

producer	consumer
repeat loop <i>Produce(i);</i> <i>wait(nempty);</i> <i>wait(mutex);</i> <i>insert(b, i)</i> <i>signal(mutex)</i> <i>signal(nfull)</i> forever	repeat loop <i>wait(nfull);</i> <i>wait(mutex);</i> <i>remove(b, i)</i> <i>signal(mutex);</i> <i>signal(nempty);</i> <i>Consume(i);</i> forever

Table 2: Bounded Buffers using Counting Semaphores

Try to solve it using only binary semaphores.

15 – Dining Philosopher's Problem

Dijkstra's dining philosopher's problem is the following resource allocation problem:

1. Suppose there are n philosophers sitting around a circular table.
2. Between each pair of adjacent philosophers is one chopstick.
3. All philosophers alternates between eating rice and thinking at their own pace.
4. The philosophers require two chopsticks to eat rice.

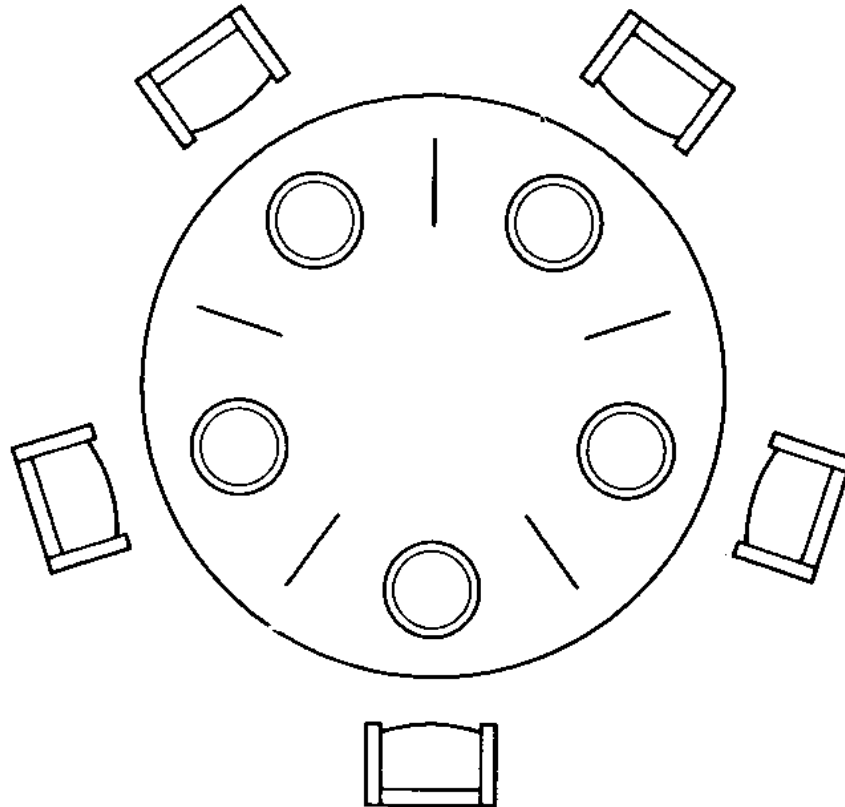


Figure 2: The Dining Philosophers for $n = 5$

16 – Dining Philosopher's Pseudocode

The i th philosopher can be modeled as a process i :

```
void philosopher(){
    think;
    acquire chopsticks;
    eat;
    release chopsticks;
}
```

Can you solve it?

17 – A Semaphore Based Approach — 1st try

The chopsticks in the dining philosophers problem can be represented as an array of n semaphores, indexed by 0 to $n - 1$, the left chopstick being at position i , and the right one being at $(i + 1) \bmod n$.

Consider the solution where the philosopher picks up the left chopstick first and then the right:

```
void philosopher(){
    think;
    wait(chopstick[i]);
    wait(chopstick[(i+1) mod n]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) mod n]);
}
```

Does it work?

18 – A Semaphore Based Approach — 2nd try

No! It deadlocks if all philosophers pickup their right chopstick.

Now consider a solution where each philosopher:

1. Picks up their left chopstick,
2. Checks to see if the right chopstick is in use.
3. If so, the philosopher puts down their left chopstick, and starts over at Step 1.
4. Otherwise the philosopher eats.

Could this approach work?

19 – A Semaphore Based Solution

No it livelocks (all could pickup their left chopstick and then put it down and repeat indefinitely). It is necessary to either:

1. introduce asymmetry or
2. limit the number of concurrently dining philosophers to $n - 1$.

20 – A Solution for the Dining Philosopher's Problem

Having odd numbered philosophers pick up their right chopstick first gives:

```
void philosopher(){
    think;
    if (odd(i)){
        wait(chopstick[(i+1) mod n]);
        wait(chopstick[i]);
    }else{
        wait(chopstick[i]);
        wait(chopstick[(i+1) mod n]);
    }
    eat;
    if (odd(i)){
        signal(chopstick[(i+1) mod n]);
        signal(chopstick[i]);
    }else{
        signal(chopstick[i]);
        signal(chopstick[(i+1) mod n]);
    }
}
```

21 – Readers Writers Problem

Consider a system containing:

1. Some shared data, for now suppose it is a database,
2. Many processes which are either:
 - (a) readers — Examine the database.
 - (b) writers — Examine and update the database.

We want to ensure that exactly one of the following is true:

1. Many readers are accessing the file concurrently but no writers are currently accessing the database.
2. At most one writer (and no readers) is accessing the database.

22 – A Weak Reader's Preference Solution

```
void reader(){
    loop
        wait(mutex);
        nr++; // Num. Readers
        if (nr = 1){ wait(wsem); }
        signal(mutex);
        READ
        wait(mutex);
        --nr;
        if (nr = 0){ signal(wsem); }
        signal(mutex);
    forever
}

void writer(){
    loop
        wait(wsem);
        WRITE
        signal(wsem);
    forever
}
```

23 – Some Useful Psuedocode Notation

Let B be a boolean condition and S be a (possibly compound) statement.

Andrews [1] uses the notation:

1. $\langle S \rangle$ — Execute statement S atomically.
We will call \langle and \rangle *angle brackets* when used in this context.
2. $\langle \textit{await} B \rightarrow S \rangle$ — Wait until condition B is met, and then execute S atomically. It is atomic since it is enclosed in angle brackets.
3. Global Invariant — A predicate which is true during process execution, with the possible exception of of during execution of atomic statements.

24 – Steps to Deriving Mutual Exclusion Solution

Andrews [1] suggests the following series of steps:

1. *Define the problem precisely* — Identify processes, specify the synchronization problem and introduce necessary variables and predicates.
2. *Outline a Solution* — Annotate the solution with assignments to variables, and identify regions requiring atomic or mutually exclusive access.
3. *Ensure the Invariant* — Check to make sure that you have sufficient synchronization to ensure the invariant.
4. *Implement the Atomic Actions* — Express the atomic actions and await statements using the available synchronization primitives.

25 – Reader Writer Outline

Letting nw be the number of writers, the invariant is:

$$INV = (nr = 0 \text{ or } nw = 0) \text{ and } (nw \leq 1) \quad (1)$$

```
int nr := 0, nw := 0;
```

```
void reader(){
    <await nw = 0 -> nr++; >
    READ
    <nr--; >
}
```

```
void writer(){
    <await nr = 0 and nw = 0 -> nw++; >
    WRITE
    <nw--; >
}
```

26 – Passing the Baton

Let e be a semaphore used to ensure atomic access and $SIGNAL$ be an operation to be defined later. Andrews [1] synchronization statements are of the form:

1. $F_1 : \langle S_i \rangle$

wait(e);

S $_j$;

SIGNAL();

2. $F_2 : \langle await B_j \rightarrow S_i \rangle$ — This solution uses:

(a) d_j — The number of processes delayed on condition B_j .

(b) b_j — A semaphore for processes waiting for condition B_j to become true.

wait(e); if (not B_j) d_j++ ; signal(e); wait(b_j);
S $_i$; SIGNAL();

27 – Passing the Baton — SIGNAL Defined

The signal statement looks complex, but it is just a list of conditions, and a check to see if any processes are blocked on that condition. Note that it is critical to reduce the appropriate count of delayed processes when a process becomes unblocked.

```
void SIGNAL(){
    if (B1 and d1 > 0){
        d1--; signal(b1);
    } else if (B2 and d2 > 0){
        d2--; signal(b2);
    } ... {
    } else if (Bn and dn > 0){
        dn--; signal(bn);
    } else { // Not waiting on any condition
        signal(e);
    }
}
```

28 – Reader Writer Solution (Passing the Baton)

```
int nr := 0; nw := 0;
// x(c) initializes x.count to c
semaphore e(1), w(0), r(0);

void reader(){
    wait(e);
    if (nw > 0){
        ++dr;
        signal(e);
        wait(r);
    }
    ++nr;
    SIGNAL(); // case 1
    READ;
    wait(e);
    --nr;
    SIGNAL(); // case 2
}
```

29 – Reader Writer Solution (Passing the Baton)

```
void writer(){
    wait(e);
    if ((nr > 0) or (nw > 0)) {
        ++dw;
        signal(e);
        wait(w);
    }
    ++nw;
    SIGNAL(); // case 3
    WRITE;
    wait(e);
    --nw;
    SIGNAL(); // case 4
}
```

30 – Reader Writer Solution (Passing the Baton)

This corresponds to a readers preference solution

```
void SIGNAL(){
    if ((dr > 0) and (nw == 0)){
        --dr;
        signal(r);
    } else if ((nr == 0) and
        (nw == 0) and (dw > 0)){
        --dw;
        signal(w);
    } else if (((dr == 0) or (nw > 0)) and
        ((nr > 0) or (nw > 0) or (dw == 0))){
        signal(e);
    }
}
```

31 – Reader Writer Problem Notes

The solution given is unfair, i.e. it imposes an unfair scheduling policy.

Try to derive the following scheduling policies:

1. a weak writer's preference solution
2. a fair solution

32 – Reader Writer Problem — Writer's Preference

To prefer writes the solution needs to adjust:

1. The condition where readers are allowed to enter the critical section. The line:

```
if (nw > 0){
```

should instead read:

```
if ((nw > 0) or (dw > 0)){
```

to force readers to block if any writers are blocked.

2. In the SIGNAL call, the condition needs to be adjusted to read:

```
void SIGNAL(){
    if ((dr > 0) and (nw == 0) and (dw == 0)){
        --dr;
        signal(r);
    } else if ((nr == 0) and
        (nw == 0) and (dw > 0)){
        --dw;
        signal(w);
    } else if
        (((dr == 0) or (nw > 0) or (dw > 0))
        and ((nr > 0) or (nw > 0) or
        (dw == 0))){
        signal(e);
    }
}
```

33 – General Notes on Semaphores

In practice we notice the following issues concerning semaphores:

1. Semaphores can express a wide variety of synchronization constructs,
2. There is a dangerous tendency to incorrectly pair wait and signal calls.
3. Construction of correct solutions becomes nontrivial.
4. Fairness of semaphores is (nearly) always enforced, unfair semaphores are notoriously difficult to deal with.
5. Passing the Baton is a powerful technique for deriving correct semaphore based solutions.

34 – Monitors

A *monitor* is a programming language construct which:

1. which encapsulates a critical section
2. guarantees no more than one process will concurrently execute within the monitor.
3. processes may block on a *condition*, and are considered outside the monitor during the blocking.
4. monitors are not necessarily fair.

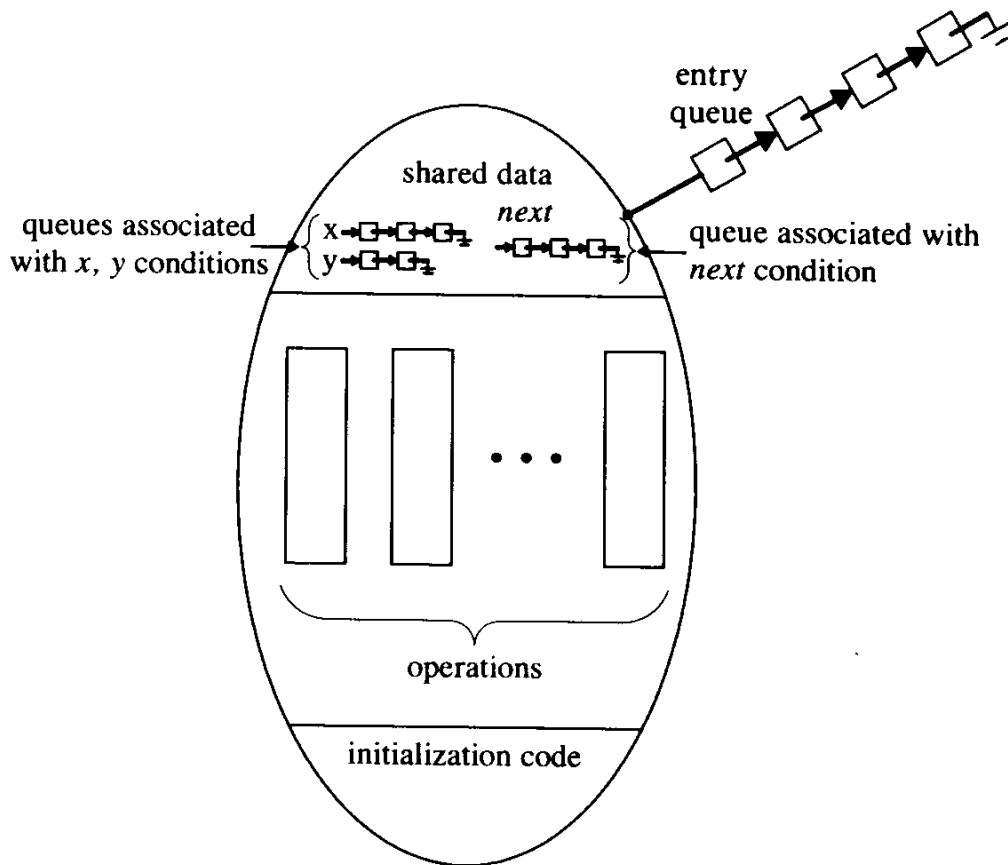
Monitors combine programming language and systems programming.

Monitors (properly speaking) are features of programming languages, and are not inherent in operating systems.

Can you name a programming language which uses monitors?

35 – Monitors

One can consider a monitor as having the appearance of:



References

- [1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.