Classic Sync Problems
Monitors

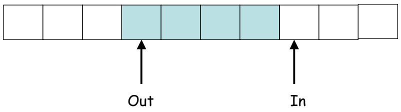
Announcements

Synchronization Problems

- Producer-Consumer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem

Producer-Consumer Problem

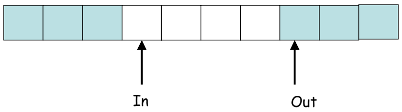
- Unbounded buffer
- Producer process writes data to buffer
  - Writes to *In* and moves rightwards
- Consumer process reads data from buffer
  - Reads from *Out* and moves rightwards
  - Should not try to consume if there is no data



Need an infinite buffer

Producer-Consumer Problem

- Bounded buffer: size 'N'
- Producer process writes data to buffer
  - Should not write more than 'N' items
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



Producer-Consumer Problem

- A number of applications:
  - Compiler's output consumed by assembler
  - Assembler's output consumed by loader
  - Web server produces data consumed by client's web browser
- Example: pipe (|) in Unix
  - > cat file | more
  - > prog | sort … what happens here?

Producer-Consumer Problem

First attempt to solve:

```
Shared: int counter;
        any_t buffer[N];

Init: counter = 0;
```

Producer

```
while (true) {
   /* produce an item in nextProduced*/
   while (counter == N)
      ; /* do nothing */
   buffer[in] = nextProduced;
   in = (in + 1) % N;
   counter++;
}
```

Consumer

```
while (true) {
   while (counter == 0)
      ; /* do nothing */
   nextConsumed = buffer[out];
   out = (out + 1) % N;
   counter--;
   /* consume an item in nextConsumed*/
}
```

Producer-Consumer Problem

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N;  /* number empty bufs */
      full = 0;   /* number full bufs */
```

Producer

```
do {
   . . .
   // produce an item in nextp
   . . .
   P(empty);
   P(mutex);
   . . .
   // add nextp to buffer
   . . .
   V(mutex);
   V(full);
} while (true);
```

Consumer

```
do {
   P(full);
   P(mutex);
   . . .
   // remove item to nextc
   . . .
   V(mutex);
   V(empty);
   . . .
   // consume item in nextc
   . . .
} while (true);
```

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
- Example: airline reservation

Readers-Writers Problem

- Many processes share a database
- Some processes write to the database
- Only one writer can be active at a time
- Any number of readers can be active simultaneously
- This problem is non-preemptive
  - Wait for process in critical section to exit
- First Readers-Writers Problem:
  - Readers get higher priority, and do not wait for a writer
- Second Readers-Writers Problem:
  - Writers get higher priority over Readers waiting to read
    - Courtois et al.

First Readers-Writers

```
Shared variables: Semaphore mutex, wrl;
                  integer rcount;

Init: mutex = 1, wrl = 1, rcount = 0;
```

Writer

```
do {

   P(wrl);
   . . .
   /*writing is performed*/
   . . .
   V(wrl);

}while(TRUE);
```

Reader

```
do {
   P(mutex);
   rcount++;
   if (rcount == 1)
      P(wrl);
   V(mutex);
   . . .
   /*reading is performed*/
   . . .
   P(mutex);
   rcount--;
   if (rcount == 0)
      V(wrl);
   V(mutex);
}while(TRUE);
```

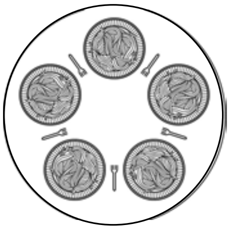
Readers-Writers Notes

- If there is a writer
  - First reader blocks on **wrl**
  - Other readers block on **mutex**
- Once a writer exists, all readers get to go through
  - Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers waiting on **wrl**, and writer exits
  - Who gets to go in first?
- Why doesn’t a writer need to use **mutex**?

Dining Philosopher's Problem

- Dijkstra

- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time
- How to avoid deadlock?



Example: multiple processes competing for limited resources

A non-solution

```
# define N      5
```

Philosopher i (0, 1, .. 4)

```
do {
    think();
    take_fork(i);
    take_fork((i+1)%N);
    eat(); /* yummy */
    put_fork(i);
    put_fork((i+1)%N);
} while (true);
```

Will this work?

```
Shared: semaphore fork[5];
Init: fork[i] = 1 for all i=0 .. 4
```

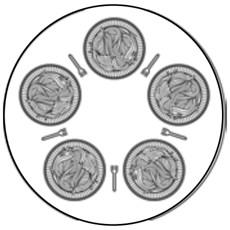
Philosopher i

```
do {
    P(fork[i]);
    P(fork[i+1]);

    /* eat */

    V(fork[i]);
    V(fork[i+1]);

    /* think */
} while(true);
```



Dining Philosophers Solutions

- Allow only 4 philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available

One possible solution

```
Shared: int state[5], semaphore s[5], semaphore mutex;
Init: mutex = 1; s[i] = 0 for all i=0 .. 4
```

Philosopher i

```
do {
    take_fork(i);
    /* eat */
    put_fork(i);
    /* think */
} while(true);
```

```
take_fork(i) {
    P(mutex);
    state[i] = hungry;
    test(i);
    V(mutex);
    P(s[i]);
}

put_fork(i) {
    P(mutex);
    state[i] = thinking;
    test((i+1)%N);
    test((i-1+N)%N);
    V(mutex);
}
```

```
test(i) {
if(state[i] == hungry
    && state[(i+1)%N] != eating
    && state[(i-1+N)%N != eating)
{
    state[i] = eating;
    V(s[i]);
}
```

Language Support for Concurrency

Common programming errors

Process i	**Process j**	**Process k**
P(S)	V(S)	P(S)
CS	CS	CS
P(S)	V(S)	

What's wrong?

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N; /* number empty bufs */
      full = 0;     /* number full bufs */
```

Producer

```
do {
   . . .
   // produce an item in nextp
   . . .
   P(mutex);
   P(empty);
   . . .
   // add nextp to buffer
   . . .
   V(mutex);
   V(full);
} while (true);
```

Consumer

```
do {
   P(full);
   P(mutex);
   . . .
   // remove item to nextc
   . . .
   V(mutex);
   V(empty);
    . . .
    // consume item in nextc
    . . .
} while (true);
```

What's wrong?

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N; /* number empty bufs */
      full = 0;     /* number full bufs */
```

Producer

```
do {
   . . .
   // produce an item in nextp
   . . .
   P(mutex);
   P(empty);
   . . .
   // add nextp to buffer
   . . .
   V(mutex);
   V(full);
} while (true);
```

What if buffer is full?

Consumer

```
do {
   P(full);
   P(mutex);
   . . .
   // remove item to nextc
   . . .
   V(mutex);
   V(empty);
    . . .
    // consume item in nextc
    . . .
} while (true);
```

Revisiting semaphores!

- Semaphores are still low-level
  - Users could easily make small errors
  - Similar to programming in assembly language
    - Small error brings system to grinding halt
  - Very difficult to debug
- Simplification: Provide concurrency support in compiler
  - Monitors

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - "in the monitor"
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
      - ⇒ Need for a wait queue
  - If thread within a monitor blocks, another can enter
- Effect on parallelism?

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . . .) {
        . . . .
    }

    procedure P2(. . . .) {
        . . . .
    }
    .
    .
    procedure PN(. . . .) {
        . . . .
    }

    initialization_code(. . . .) {
        . . . .
    }
}
```

For example:

```
Monitor stack
{
    int top;
    void push(any_t *) {
        . . . .
    }

    any_t * pop() {
        . . . .
    }

    initialization_code() {
        . . . .
    }
}
```

only one instance of stack can be modified at a time

Synchronization Using Monitors

- Defines Condition Variables:
  - condition x;
  - Provides a mechanism to wait for events
    - Resources available, any writers
- 3 atomic operations on *Condition Variables*
  - x.wait(): release monitor lock, sleep until woken up
    - ⇒ condition variables have waiting queues too
  - x.notify(): wake one process waiting on condition (if there is one)
    - No history associated with signal
  - x.broadcast(): wake all processes waiting on condition
    - Useful for resource manager
- Condition variables are not Boolean
  - If(x) then { } does not make sense

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
    any_t buf[N];
    int n = 0, tail = 0, head = 0;
    condition not_empty, not_full;
    void put(char ch) {
            if(n == N)
                wait(not_full);
            buf[head%N] = ch;
            head++;
            n++;
            signal(not_empty);
    }
    char get()  {
            if(n == 0)
                wait(not_empty);
            ch = buf[tail%N];
            tail++;
            n--;
            signal(not_full);
            return ch;
    }
}
```

What if no thread is waiting when signal is called?

Compare with Semaphore Solution

```
Monitor Producer_Consumer {
    any_t buf[N];
    int n = 0, tail = 0, head = 0;
    condition not_empty, not_full;
    void put(char ch) {
            if(n == N)
                wait(not_full);
            buf[head%N] = ch;
            head++;
            n++;
            signal(not_empty);
    }
    char get()  {
            if(n == 0)
                wait(not_empty);
            ch = buf[tail%N];
            tail++;
            n--;
            signal(not_full);
            return ch;
    }
}
```

```
Init: mutex = 1;  empty = N; full = 0;
```

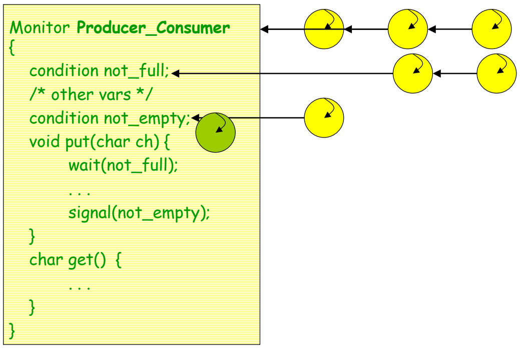
Producer

```
do {
    // produce an item in nextp
    P(empty);
    P(mutex);
      // add nextp to buffer
    V(mutex);
    V(full);
} while (true);
```

Consumer

```
do {
    P(full);
    P(mutex);
    // remove item to nextc
    V(mutex);
    V(empty);
     // consume item in nextc
} while (true);
```

Producer Consumer using Monitors



```
Monitor Producer_Consumer
{
    condition not_full;
    /* other vars */
    condition not_empty;
    void put(char ch) {
        wait(not_full);
        . . .
        signal(not_empty);
    }
    char get() {
        . . .
    }
}
```

Types of Monitors

What happens on notify():

- Hoare: signaler immediately gives lock to waiter (theory)
  - Condition definitely holds when waiter returns
  - Easy to reason about the program
- Mesa: signaler keeps lock and processor (practice)
  - Condition might not hold when waiter returns
  - Fewer context switches, easy to support broadcast
- Brinch Hansen: signaller must immediately exit monitor
  - So, notify should be last statement of monitor procedure

Mesa-style monitor subtleties

```
char buf[N];                              // producer/consumer with monitors
int n = 0, tail = 0, head = 0;
condition not_empty, not_full;
void put(char ch)
        if(n == N)
                wait(not_full);
        buf[head%N] = ch;
        head++;
        n++;
     signal(not_empty);
char get()
        if(n == 0)
                wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
```

Consider the following time line:

0. initial condition: n = 0
1. c0 tries to take char, blocks on not_empty (releasing monitor lock)
2. p0 puts a char (n = 1), signals not_empty
3. c0 is put on run queue
4. Before c0 runs, another consumer thread c1 enters and takes character (n = 0)
5. c0 runs.

Possible fixes?

Mesa-style subtleties

```
char buf[N];                              // producer/consumer with monitors
int n = 0, tail = 0, head = 0;
condition not_empty, not_full;
void put(char ch)
        while(n == N)
            wait(not_full);
        buf[head] = ch;
        head = (head+1)%N;
        n++;
        signal(not_full);
char get()
        while(n == 0)
            wait(not_empty);
        ch = buf[tail];
        tail = (tail+1) % N;
        n--;
        signal(not_full);
        return ch;
```

When can we replace “while” with “if”?

Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by a lock
  - Wait: blocks on thread and gives up the lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if value less than 0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - V() increments value, so future threads need not wait on P()
    - Condition variables have no history
- However they can be used to implement each other

Hoare Monitors using Semaphores

For each procedure F:

```
P(mutex);

/* body of F */

if(next_count > 0)
   V(next);
else
   V(mutex);
```

Condition Var Wait: x.wait:

```
x_count++;
if(next_count > 0)
   V(next);
else
   V(mutex);
P(x_sem);
x.count--;
```

Condition Var Notify: x.notify:

```
If(x_count > 0) {
   next_count++;
   V(x_sem);
   P(next);
   next_count--;
}
```

Language Support

- Can be embedded in programming language:
  - Synchronization code added by compiler, enforced at runtime
  - Mesa/Cedar from Xerox PARC
  - Java: **synchronized, wait, notify, notifyall**
  - C#: **lock, wait (with timeouts) , pulse, pulseall**
- Monitors easier and safer than semaphores
  - Compiler can check, lock implicit (cannot be forgotten)
- Why not put everything in the monitor?

Eliminating Locking Overhead

- Remove locks by duplicating state
  - Each instance only has one writer
  - Assumption: assignment is atomic
- Non-blocking/Wait free Synchronization
  - Do not use locks
  - Optimistically do the transaction
  - If commit fails, then retry

Optimistic Concurrency Control

- Example: hits = hits + 1;
  - A) Read hits into register R1
  - B) Add 1 to R1 and store it in R2
  - C) Atomically store R2 in hits only if hits==R1 (i.e. CAS)
    - If store didn’t write goto A
- Can be extended to any data structure:
  - A) Make copy of data structure, modify copy.
  - B) Use atomic word compare-and-swap to update pointer.
  - C) Goto A if some other thread beat you to the update.
- 👍 Less overhead, deals with failures better
- 👎 Lots of retrying under heavy load