Concurrency

- Is a fundamental concept in operating system design
- Processes execute **interleaved** in time on a single processor
  - Creates the illusion of simultaneous execution
  - Benefits for processing efficiency
- Processes execute truly overlapping / parallel on multiprocessor systems
- Issues (mainly on resource allocation and process communication)
  - Communication among processes
  - Sharing of / competition for resources
  - Synchronization of activities of multiple processes
  - Allocation of processor time to processes
Difficulties of Concurrency

- **Sharing of global resources / Competition**
  - As processes operate in an interleaved fashion, the ordering of read / write operations (by different processes) on shared data objects is critical, danger of race conditions

- **Optimal allocation of resources**
  - Sub-optimal allocation: resources may be allocated to processes, process may be suspended before it can use / release the resource
  - May lead to deadlock situations

- **Non-deterministic behaviors**
  - Concurrent systems are difficult to debug, as it is hard to predict when which instruction of which process is scheduled for execution, the scheduling pattern may not be reproducible
Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - Beside shared data, this can also be I/O devices, memory access, among others.

- Basic requirement
  - Allow one process the execution of a course of actions, while excluding all other processes from performing this course of actions
  - If not, we will see the problem of … (next slide)
Race Conditions

- **Race conditions** occur when multiple processes / threads read and write shared data items
  - A race condition is the behavior of OS where the output is dependent on the sequence or timing of other uncontrollable processes.
- The processes *race* to perform their read / write actions
- The final result depends on the order of execution
  - The “loser” of the race is the process that performs the last update and determines the current value (which will be overwritten) of a shared data item
Race Conditions

- Scenario
  - Process 1 reads shared variable $x$ ($x = 10$)
  - Process 1 is preempted by scheduler
- Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

```c
process1() {
    read ( x )
    x = x - 5
    write ( x )
}
```

1. Read $x = 10$

```c
process2() {
    read ( x )
    x = x + 2
    write ( x )
}
```

$X = 10$
Race Conditions

• Scenario
  – Process 1 reads shared variable x (x = 10)
  – Process 1 is preempted by scheduler
  – Process 2 continues with its manipulations
  – Process 1 is scheduled and completes its actions

```java
process1()
{
    read ( x )
    x = x - 5
    write ( x )
}
```

1. Read x = 10

```java
X = 10
```

2. 

```java
process2() {
    read ( x )
    x = x + 2
    write ( x )
}
```
Example

Race Conditions

- Scenario
  - Process 1 reads shared variable `x` (`x = 10`)
  - Process 1 is preempted by scheduler
- Process 2 continues with its manipulations
- Process 1 is scheduled and completes its actions

```java
process1()
{
    read ( x )
    x = x - 5
}
```

Scheduler suspends process 1

1. Read `x = 10`
2.

```java
process2()
{
    read ( x )
    x = x + 2
    write ( x )
}
```

Scheduler schedules process 2
Race Conditions

- Scenario
  - Process 1 reads shared variable x (x = 10)
  - Process 1 is preempted by scheduler
  - Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

```text
process1()
{
  read ( x )
  x = x - 5

  write ( x )
}
```

Scheduler suspends process 1

1. Read x = 10

Scheduler schedules process 2

2. x = x - 5

3. Read x = 10

```text
process2()
{
  read ( x )
  x = x + 2
  write ( x )
}
```
Race Conditions

- Scenario
  - Process 1 reads shared variable x (x = 10)
  - Process 1 is preempted by scheduler
  - Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

```
process1() {
  read ( x )
  x = x - 5
  write ( x )
}
```

```
process2() {
  read ( x )
  x = x + 2
  write ( x )
}
```
Race Conditions

- Scenario
  - Process 1 reads shared variable x (x = 10)
  - Process 1 is preempted by scheduler

  ```
  process1()
  {
    read ( x )
    x = x - 5
  }
  write ( x )
  ```

  1. Read x = 10
  2. Scheduler suspends process 1

  - Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

  ```
  process2()
  {
    read ( x )
    x = x + 2
  }
  write ( x )
  ```

  3. Read x = 10
  4. Scheduler schedules process 2
  5. Write x = 12
Example

Race Conditions

- **Scenario**
  - Process 1 reads shared variable x (x = 10)
  - Process 1 is preempted by scheduler
  - Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

```plaintext
process1()
    read ( x )
    x = x - 5

Scheduler suspends process 1

1. Read x = 10

Scheduler resumes process 1

write ( x )

process2()
    read ( x )
    x = x + 2

Scheduler schedules process 2

3. Read x = 10

X = 10
X = 12

5. Write x = 12

write ( x )
```
Example

Race Conditions

- **Scenario**
  - Process 1 reads shared variable \(x\) \((x = 10)\)
  - Process 1 is preempted by scheduler
- **Scenario**
  - Process 2 continues with its manipulations
  - Process 1 is scheduled and completes its actions

```plaintext
process1()

1. Read \(x = 10\)
2. \(x = x - 5\)

Scheduler suspends process 1

WRONG!

Scheduler resumes process 1

write (x)

process2()

3. Read \(x = 10\)
4. \(x = x + 2\)

Scheduler schedules process 2

4. Write \(x = 12\)

Lost Update!

write (x)

6. Write \(x = 5\)
Race Conditions

- Why do race conditions occur
  - Scheduling: Context switches at arbitrary times during execution
  - Outdated Information: Processes / Threads operate with “out-of-date” copies of memory values in registers
    - Other processes may already have changed the original value in the shared memory location

- Critical Section (CS)
  - Part of the program code that accesses shared resource

```plaintext
process1() { 
  read ( x )
  x = x - 5
  write ( x )
}

process2() { 
  read ( x )
  x = x + 2
  write ( x )
}
```
Race Conditions

- Mutual Exclusion
  - Solutions to race conditions: At any time, at most one process can be in its critical section, which is called mutual exclusion
  - The challenge now is to design a mutual exclusion protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors)
  - In order to guarantee mutual exclusion, we have to control the execution of program code that accesses shared resources
Control Resource Competition

- A race condition can be avoided, if we arrange the execution of processes such that
  - Only one of them is executing its critical section (mutual exclusion)
  - This one process can finish the execution of its critical section, even if it is temporarily suspended
  - Any other process sharing the resource is blocked, in the meantime, from accessing the critical section
Control Resource Competition

Mutual exclusion using critical sections
Requirements for Mutual Exclusion

- Serialization of access:
  - Only one process at a time is allowed in the critical section for a resource
  - A process that halts in its non-critical section must remain so without interfering with other processes halting on a critical section
  - A process must not be delayed in accessing a critical section when available
  - A process remains inside its critical section for a finite time only, and leaves the critical section asap when finish
  - No assumptions are made about relative process speeds or number of processes
  - No deadlock or starvation (next slide)
Mutual Exclusion

- Enforcing mutual exclusion (with a poor implementation) may create two problems
  - Deadlocks: Processes wait forever for each other to free resources
  - Starvation (a bit different from Chap 9): A process waits forever to be allowed to enter its critical section

- To achieve mutual exclusion, we need to enable two patterns of execution
  - Normally: processes are scheduled for execution in an interleaved fashion
  - Critical section: Enforce serial execution, i.e., only one process is allowed to enter its critical section

We have to temporarily limit / completely avoid interleaved execution of processes. Then How?
Critical Section & Control Entry

- Critical Section (CS) needs a defined entry point
  - Has to be a kind of “gate” that checks whether a process may proceed execution
  - Has to communicate to other processes that this process has entered critical section

- Critical section needs defined exit point
  - Process communicates exit of critical section to other processes

```c
process ()
{
    cs_entry()
    action1()
    action2()
    ...
    actionN()
    cs_exit()
}
```
Solutions to Mutual Exclusion

Multiple hardware and software supports are typically provided to enable mutual exclusion and address race conditions:

- **Hardware**
  - Disable interrupts
  - Processor provides special instructions

- **Software**
  - Use lock variables to control access to critical section

- **Higher operating system constructs**
  - Semaphores, monitor, among others
Mutual Exclusion: Disable Interrupts

- **Interrupt Disabling**
  - A process cannot be interrupted until it enables interrupts again
  - Therefore, guarantees mutual exclusion on a uniprocessor system

- **Disadvantages**
  - Does not work on a multiprocessor architecture
  - The efficiency of execution may degrade as concurrency of processes on one processor is reduced
Mutual Exclusion: Entry Control

Spin Lock:  
- "Busy Wait"
Test a condition until it becomes true:
while(! condition())
    /* do nothing */;

Busy Wait (shared spin locks):
- If a process wants to enter its critical section, it checks to see whether the entry is allowed

Sleep and Wakeup:
- Suspend process until shared resource becomes available and currently accessing process has left critical section

Sleep and Wakeup:
- Use system calls sleep() and wakeup() to suspend and reschedule processes
  sleep();
  ...
  wakeup(process);
Shared Lock

- We could introduce a shared variable “lock” (also loosely called “mutex” or a “mutex lock”)
  - Used to indicate whether one of the competing processes has entered critical section
    - If lock==0, the lock is not set (resource available)
    - If lock==1, then lock is set (resource not available)
  - All processes that compete for a shared resource, also share this lock variable (i.e., lock is also shared)

- Procedures of using a shared lock:
  - A process checks/tests the lock
    - If lock is not set, process sets lock and enters critical section
    - If lock is set, process waits

- Problem?

As the “lock” is a shared resource, race conditions can occur when it requires both check and set operations
Mutual Exclusion: Hardware Support

- Special machine instructions:
  - Processor has instructions that perform two actions in sequence, which cannot be interrupted
  - Help to implement a Busy Wait (and others)

- **Test and Set** Instruction
  - Reads a memory location into a register and writes a value > 0 into this memory location in one atomic action (then tests whether register holds a value > 0)

- **Exchange** Instruction
  - Exchanges the content of two memory locations in one atomic action, usually a register and a memory word
Mutual Exclusion: Hardware Support

- The TSL Instruction (Test and Set Lock)
  - Does a Read + Write in one go
  - In an atomic operation to:
    - Read content of a shared memory location, the variable "lock", into register and overwrites "lock" with a non-zero value
    - Tests whether a shared variable, used as a "lock", is set
      - If value == 0, lock is not set; If a value > 0, lock is set

```
enter_section
  TSL REGISTER, LOCK
  CMP REGISTER, #0
  JNE enter_section
  RET

exit_section
  MOVE LOCK, #0
  RET
```

```c
  // copy LOCK to register and set lock to 1 if lock == 0?
  // if it was > 0, LOCK was set, loop again return to caller, critical section free to enter
```
Mutual Exclusion: Hardware Support

- The TSL Instruction Implements a “busy wait”: process will do this check until the test indicates a value == 0
  - Process will enter critical section, if test is successful
  - Shared memory “lock” is already set to a value > 0
  - When process exits critical section, it will write a value == 0 into the shared lock variable

```c
process()
{
    while ( testset(LOCK) ) /* do nothing */ ;
    do_critical_section() ;
    LOCK = 0 ;
}
```
Mutual Exclusion: Hardware Support

- The Exchange Instruction
  - Swaps content of a register and a shared lock variable in one atomic action
  - Provides the same service as TSL

```assembly
enter_section
  MOVE REGISTER, #1           put 1 into the register
  XCHG REGISTER, LOCK        swap content of register with content of LOCK
  CMP REGISTER, #0           was lock == 0 ?
  JNE enter_section         if it was > 0, LOCK was set, loop again
  RET                       return to caller, critical section free to enter

exit_section
  MOVE LOCK, #0
  RET
```
Pros and Cons of Using Locks

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and easy to verify
  - It can be used to support multiple critical sections, each critical section has its own lock variable

- **Disadvantages**
  - Implementation of a “Busy Waiting” continuously consumes processor instruction cycles
  - Starvation may occur, when the process leaves its critical section and more than one process is waiting
  - Deadlock may occur
OS Solutions for Mutual Exclusion

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.</td>
</tr>
<tr>
<td>Binary Semaphore</td>
<td>A semaphore that takes on only the values 0 and 1.</td>
</tr>
<tr>
<td>Mutex</td>
<td>Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).</td>
</tr>
<tr>
<td>Condition Variable</td>
<td>A data type that is used to block a process or thread until a particular condition is true.</td>
</tr>
<tr>
<td>Monitor</td>
<td>A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor’s variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.</td>
</tr>
</tbody>
</table>

- These mechanisms are used to implement a “sleep and wakeup” based on process synchronization.
Semaphores

- Synchronization mechanism, no need for Busy Wait
- Two or more processes can communicate by means of simple signals
  - One process can be forced to wait for a signal from other processes
- Semaphore can act as a lock, making processes wait (but semaphore is also more than a lock, and can be used to model # resources)
Semaphores

- A semaphore is a data structure that contains:
  - A counter (e.g., value)
  - A waiting queue (e.g., plist)
- Semaphores are based on a decrement-increment mechanism:
  - If semaphore $S > 0$, then process can continue execution
  - If semaphore $S \leq 0$, then process has to wait for signal

```c
typedef struct {
    int value;
    queue plist;
} semaphore;
```
Semaphores

- Semaphore can only be accessed by two atomic functions

  - **wait** (**S**): decrements semaphore counter
    - If **S** <= 0 then processes calling **wait**(**S**) are suspended and added to the semaphore waiting queue
    - Also called **P operation** (atomic): derived from a Dutch word "Proberen", which means "to test".

  - **signal**(**S**): increments semaphore counter
    - processes calling **signal**(**S**) wake up other processes immediately (bring one process to ready queue)
    - Also called **V operation** (atomic): derived from the word "Verhogen", which means "to increment".
Semaphores

- Operating systems distinguish between counting and binary semaphores
  - Value of a counting semaphore can range over an unrestricted domain
    - Initial value determines how many resources are available in the system to be used for processes
  - Value of a binary semaphore toggles between 0 and 1 (with a similar functionality as a lock)
- Counting semaphores can be used to control access to a finite set of resources
- Binary semaphores can be used as “mutex locks” to deal with the critical section problem
Counting Semaphores

- Counting semaphores can be used to control access to a resource with finite capacity
  - Counts how many units of a resource are still available after fulfilling a request
- Example:
  - A room booking system with $n$ rooms
  - A buffer with $n$ places
- Procedures of using a semaphore
  - The semaphore is initialized to the number of units available
  - Each process that wishes to use a resource, calls `wait()` (i.e., `P`), which decrements the semaphore counter
  - When the count goes to 0, the capacity of a resource is exhausted and any new process will have to wait
  - When a process releases its resource, the counter will be increased by calling `signal()` (i.e., `V`)
Counting Semaphores

```c
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Note: `SemWait()` and `SemiSignal()` (i.e., P and V) are atomic operations meaning there are at least one blocked process on the semaphore.
Counting Semaphores

For example: We have four process ABDC using a shared buffer; ABC are consumers, and D is the producer.
Binary Semaphores

```c
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s) {
    if (s.value == one) {
        s.value = zero;
    } else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s) {
    if (s.queue is empty()) {
        s.value = one;
    } else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Note: SemWait() and SemiSignal() (i.e., P and V) are atomic operations
Binary Semaphores

- Binary semaphores can be used to implement mutex locks
  - Many processes share a semaphore “mutex”
  - Note: a mutex can be considered a binary semaphore, but the call to wait(mutex) and signal(mutex) has to be by the same process.

```c
process ()
{
    wait ( mutex ) ;
    critical_section () ;
    signal ( mutex ) ;
    remainder_section () ;
}
```
Strong / Weak Semaphores

- The Semaphore waiting queue holds all processes waiting on the semaphore
- **Strong semaphores**
  - Use FIFO queue: the process that has been blocked the longest is released from the queue first
- **Weak semaphores**
  - The order in which processes are removed from the queue is not specified (well, not FIFO)
Deadlock and Starvation

- **Deadlock**
  - Two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes.

    ![Diagram showing deadlock example]

- **Starvation**
  - A process waits indefinitely: may occur if waiting list of semaphore is served in LIFO (last-in-first-out) order.
Deadline

- Essential aspect of real-time system
- Tasks perform their function within prescribed deadline
  - Failure may result in severe consequences
- Deadlines can be
  - Hard
  - Soft
Classic Problems of Synchronization

- Several processes running in an OS share resources due to which problems like data inconsistency may arise
  - For example, one process changing the data in a memory location while another process is trying to read the data from the same memory location
  - It is possible that the data read by the second process will be erroneous
- Three classic synchronization problems:
  - Producer-Consumer Problem
  - Reader-Writer Problem
  - Dining Philosophers Problem
Producer-Consumer Problems

A first Producer-Consumer problem

- We assume an *unbounded/infinite* buffer consisting of a linear array of elements
- *in* points to the next item to be produced
- *out* points to the next item to be consumed

**Infinite Buffer** for the Producer/Consumer Problem; the shaded area indicates portion of buffer that is occupied
A Solution to the Infinite-Buffer Producer-Consumer Problem Using Binary Semaphores

In your first impression based on previous (non-OS) programming experience, do you notice any problems?

```c
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}

void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}

void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

"delay" encodes whether the buffer is empty: delay==0 → empty delay==1 → not empty
Possible scenario for the program

**Note:** White areas represent the critical section controlled by the `s` semaphore.
A Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

This is an **incorrect** solution

Two variables \( n \) and \( delay \) are used to refer to the same information (e.g., whether the buffer is empty, so inconsistency can exit between them.)

```c
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

"delay" encodes whether the buffer is empty:

delay==0 \( \rightarrow \) empty
delay==1 \( \rightarrow \) not empty
consumer() can be preempted during data consumption (thus \( n \) can be changed), before it can use the correct \( n \) to update “delay”

<table>
<thead>
<tr>
<th></th>
<th>Producer</th>
<th>Consumer</th>
<th>s</th>
<th>n</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>
To summarize:

The problem is caused by the facts that

1. Two variables ($n$ and `delay`) are used to refer to the same information (e.g., whether the buffer is empty, so inconsistency can exit between them.

2. `consumer()` can be preempted during data consumption (thus $n$ can be changed), before it can use the correct $n$ to update “delay”

```c
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```
A **correct** solution to the Infinite-Buffer Producer-Consumer Problem Using Binary Semaphores

```c
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
This correct solution here is still an awkward solution, because of the global and local variables of $m$ and $n$. 
Since \( n \) is used to track the number of available data elements, an elegant solution is to define \( n \) as a semaphore.

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

“\( n \)” is a counting semaphore, encoding the number of data pieces in the buffer.

“\( s \)” is a binary semaphore, working as a mutex lock.

Now, the previous “int \( n \)” and “binary semaphore delay” are encoded using a single counting semaphore, thus we remove the possible inconsistency between the previous \( n \) and delay.
In consumer(), if we switch the ordering of `semWait(n)` and `semWait(s)` as shown on the right, does the resulted implementation work? Why?

```c
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(s);
        semWait(n);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```
In `consumer()`, if we switch the ordering of `semSignal(s)` and `semSignal(n)` as shown on the right, does the resulted implementation work? Why?
Now consider a 2nd producer-consumer problem. We assume a **bounded buffer** consisting of a circular array of elements.

**Finite/bounded Buffer** for the Producer-Consumer Problem; the shaded area indicates portion of buffer that is occupied.
Does the previous solution for infinite-buffer producer-consumer problem work for the bounded-buffer problem? Why?

```c
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```
The problem is the buffer now can be full (which is different from infinite-buffer problems)

In this case, the producer should not insert any data into the buffer

We thus need a new semaphore to encode the available space in the buffer

```c
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```
Similar questions:

Can we switch the ordering of `semWait(e)` and `semWait(s)`?

Can we switch the ordering of `semSignal(s)` and `semSignal(e)`?
A monitor is a software construct that serves two purposes:

- **enforce mutual exclusion** of concurrent access to shared data objects
  - Processes have to acquire a lock to access such a shared resource
- **support synchronization** between processes accessing shared data
  - Multiple processes may use monitor-specific wait()/signal() mechanisms to wait for particular conditions to hold

Programs using monitors are supposed to allow easier implementation of mutual exclusion and synchronization
Monitor: Mutual Exclusion

- A monitor is a software construct consisting of
  - One or more procedures
  - Some local data that can only be accessed via these procedures
- When a process calls one of these procedures, it "enters" the monitor
  - The process has to acquire a monitor lock
- The monitor guarantees that
  - Only one process at a time may call one of these procedures and "enter" the monitor
  - All other processes have to wait
Monitor: Synchronization

- A monitor also supports process synchronization with condition variables
  - Only accessible by the process within the monitor
  - Two functions
    - `wait(c)` : a process calling this function is suspended, releases monitor lock, waits until a signal based on condition c is received, reacquires lock
    - `signal(c)` : resume one of the processes waiting
- A monitor may maintain a set of these condition variables
  - Processes may wait for any of these condition variables
    - A suspended process releases the monitor lock
  - For each condition variable, the monitor maintains a waiting queue
Monitor: Synchronization

- Process “enters” a monitor
  - It calls one of the procedures
    - Process acquires lock, or may have to wait
- One process currently in the monitor
  - Process may call `wait(condition)`
    - made to wait for a condition in a condition queue
  - Process may call `signal(condition)`
    - This resumes one of the processes waiting for this conditional signal
Reader-Writer Problem (Brief)

- Processes that share a resource, e.g. a database, may perform read and write operations.
- Write operations are critical (pattern 1)
  - As it is a change to the shared data object, only one writer at a time may access the data object.
  - All other processes, readers and writers, must be excluded from access.
- Read operations are critical (pattern 2)
  - Many readers at the same time may read a shared data object.
  - During reading, writers should not change the data.
- Writers must have exclusive access to shared data.
- Readers can access shared data simultaneously.
Dining Philosophers Problem (Brief)

- Five philosophers sit around a table for dinner
- There are only 5 forks on the table, two neighboring philosophers share one fork
- Each philosopher needs two forks to eat
### OS Solutions for Mutual Exclusion

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a <strong>counting semaphore</strong> or a <strong>general semaphore</strong>.</td>
</tr>
<tr>
<td>Binary Semaphore</td>
<td>A semaphore that takes on only the values 0 and 1.</td>
</tr>
<tr>
<td>Mutex</td>
<td>Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).</td>
</tr>
<tr>
<td>Condition Variable</td>
<td>A data type that is used to block a process or thread until a particular condition is true.</td>
</tr>
<tr>
<td>Monitor</td>
<td>A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <strong>critical sections</strong>. A monitor may have a queue of processes that are waiting to access it.</td>
</tr>
</tbody>
</table>