Process Synchronization

Reading:

Silberschatz
chapter 6

Additional Reading:

Stallings
chapter 5
Outline

- Concurrency
  - Competing and Cooperating Processes
- The Critical-Section Problem
  - Fundamental requirements, Attempts
  - Dekker’s algorithm
  - Peterson’s algorithm
  - Bakery algorithm
  - Hardware synchronization
- Semaphores
  - Classical Problems
- Monitors
Concurrency

**Motivation:** Overlap computation with I/O; simplify programming

- **Hardware parallelism:** CPU computing, one or more I/O devices are running at the same time

- **Pseudo parallelism:** rapid switching back and forth of the CPU among processes, pretending to run concurrently

- **Real parallelism:** can only be achieved by multiple CPUs

Real parallelism → not possible in single CPU systems
Concurrent Processes

In a multiprogramming environment, processes executing concurrently are either competing or cooperating.

Responsibilities of OS

Competing processes: Careful allocation of resources, proper isolation of processes from each other.

Cooperating processes: Protocols to share some resources, allow some processes to interact with each other; Sharing or Communication.
Competing Processes

Compete for devices and other resources

*Unaware of one another*

**Example:**
Independent processes running on a computer

**Properties:**
Deterministic - Start/Stop without side effects
Reproducible - Proceed at arbitrary rate
Cooperating Processes

Aware of each other, by communication or by sharing resources, may affect the execution of each other.

Example: Transaction processes in Railways/Airline/Stocks

Properties:
Shares Resources or Information
Non-deterministic
May be irreproducible
Race Condition
Why Cooperation?

- **Share Some Resources**
  - One checking accounts or res. files → Many tellers

- **Speed up**
  - Read next block while processing current one
  - Divide jobs into smaller pieces and execute them concurrently

- **Modularity**
  - Construct systems in modular fashion
Competition for Resources

- **Conflicting Demands**
  - I/O devices, memory, process time, ...
  - Blocked process $\rightarrow$ Slow or never gets access

- **Problems**
  - Mutual exclusion
  - Enforcement of mutual exclusion
    - Deadlock
    - Starvation
Process Cooperation

Cooperation by Sharing
- Multiple process → Shared file/database
- Control problems → Mutual exclusion, deadlock, starv
- Data items may be accessed in different modes
- Data Coherence or Racing

Cooperation by Communication
- Sync various activities
- No sharing, No mutual exclusion
- Starvation and Deadlock
The Producer/Consumer Problem

- Also called as bounded-buffer problem
- A producer produces data that is consumed by a consumer (e.g., spooler and printer)
- A buffer holds the data which is not yet consumed
- There exists several producers and consumers
- Code for the Producer/Consumer Process?

[Diagram of producer and consumer processes with buffer]
The Producer/Consumer Problem

- Two logical pointers; in and out
- in - next free position in the buffer
- in == out, Empty; ((in +1) % BUFFER_SIZE == out, Full

Producer process

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

Consumer process

```c
item nextConsumed;
while (1) {
    while (in == out) ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```
The Potential Problem

Last solution allows BUFFER_SIZE – 1
Remedy → use integer variable, counter = 0

Shared data

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

- **Consumer process**

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

- **Producer process**

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

- The statements `counter++;` and `counter--;` must be performed *atomically*.

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

- **Race condition** → Several processes access and manipulate shared data concurrently.
  Final value of the shared data → Process that finishes last

- To prevent race conditions, concurrent processes must be synchronized.
## An Example

<table>
<thead>
<tr>
<th>time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:00</td>
<td>Look in fridge. <em>Out of milk</em></td>
<td></td>
</tr>
<tr>
<td>8:05</td>
<td>Leave for store.</td>
<td></td>
</tr>
<tr>
<td>8:10</td>
<td>Arrive at store.</td>
<td>Look in fridge. <em>Out of milk</em></td>
</tr>
<tr>
<td>8:15</td>
<td>Buy milk.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>8:20</td>
<td>Leave the store.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>8:25</td>
<td>Arrive home, put milk away.</td>
<td>Buy milk.</td>
</tr>
<tr>
<td>8:30</td>
<td></td>
<td>Leave the store.</td>
</tr>
<tr>
<td>8:35</td>
<td></td>
<td>Arrive home, <em>OH! OH!</em></td>
</tr>
</tbody>
</table>

Someone gets milk, but NOT everyone *(too much milk!)*
Mutual Exclusion

- If cooperating processes are not synchronized, they may face unexpected **timing** errors → *too-much-milk-problem*

- **Mutual exclusion** is a mechanism to avoid data inconsistency. It ensure that only one process (or person) is doing certain things at one time.

Example: Only one person *buys milk* at a time.
Critical Section

- A section of code or collection of operations in which only one process may be executing at a given time, which we want to make atomic

  Atomic operations are used to ensure that cooperating processes execute correctly

- Mutual exclusion mechanisms are used to solve CS problems
Critical Section

Requirements for the solution to CS problem

- **Mutual exclusion** – no two processes will simultaneously be inside the same CS

- **Progress** – processes wishing to enter critical section will eventually do so in finite time

- **Bounded waiting** – processes will remain inside its CS for a short time only, without blocking
Critical Section Problem - Attempts

- General structure of process
  do {
    Initialization
    entry protocol
    critical section
    exit protocol
    reminder section
  } while (1);

- Only two processes (\( P_i \) and \( P_j \))

- Process may share some common variables \( \rightarrow \) Sync their actions
Attempt 1: Taking Turns

- **Approach** → keep a track of CS usage with a shared variable `turn`

- **Initialization:**
  
  ```c
  shared int turn;
  ...
  turn = i;
  ```

- **Entry protocol:** (for process `i`)
  
  ```c
  /* wait until it's our turn */
  while (turn != i) {
  }
  ```

- **Exit protocol:** (for process `i`)
  
  ```c
  /* pass the turn on */
  turn = j;
  ```

**Problem?**
Attempt 2: Using Status Flags

- **Approach**: Usage of a shared boolean array named as `flags` for each process; flag values – BUSY when in CS or FREE otherwise.

- **Initialization**:
  ```c
  typedef char boolean;
  ...
  shared boolean flags[n - 1];
  ...
  flags[i] = FREE;
  ...
  flags[j] = FREE;
  ```

- **Entry protocol**: (for process \( i \))
  ```c
  /* wait while the other process is in its CS */
  while (flags[j] == BUSY) {
  }
  -->
  /* claim the resource */
  flags[i] = BUSY;
  ```

- **Exit protocol**: (for process \( i \))
  ```c
  /* release the resource */
  flags[i] = FREE;
  ```
Attempt 3: Using Status Flags Again

- **Approach** → same as attempt 2, but now each process sets its own flag *before* testing others flag to avoid violating mutual exclusion.

- **Initialization:**
  ```
  typedef char boolean;
  ...
  shared boolean flags[n - 1];
  ...
  flags[i] = FREE;
  ...
  flags[j] = FREE;
  ```

- **Entry protocol**: (for process $i$)
  ```
  /* claim the resource */
  flags[i] = BUSY;

  /* wait if the other process is using the resource */
  while (flags[j] == BUSY) {
  }
  ```

- **Exit protocol**: (for process $i$)
  ```
  /* release the resource */
  flags[i] = FREE;
  ```

**Problem?**
Attempt 4: Last Try!

- **Approach** → same as attempt 3, but now we periodically clear and reset our own flag while waiting for other one, to avoid deadlock.

- **Initialization:**
  
  ```
  typedef char boolean;
  shared boolean flags[n - 1];
  ...
  flags[i] = FREE;
  ...
  flags[j] = FREE;
  ```

- **Entry protocol:** (for process $i$)

  ```
  /* claim the resource */
  flags[i] = BUSY;
  /* wait if the other process is using the resource */
  while (flags[j] == BUSY) {
    flags[i] = FREE;
    delay a while ;
    flags[i] = BUSY;
  }
  ```

- **Exit protocol:** (for process $i$)

  ```
  /* release the resource */
  flags[i] = FREE;
  ```

**Problem?**
Dekker’s Algorithm

- **Approach** → same attempt 4, but now we judiciously combine the turn variable (attempt 1) and the status flags.

- **Initialization:**
  
  ```
  typedef char boolean;
  shared boolean flags[n - 1];
  shared int turn;
  ...
  turn = i;
  ...
  flags[i] = FREE;
  ...
  flags[j] = FREE;
  ```

- **Entry protocol:** (for process $i$)
Dekker’s Algorithm

◆ **Entry protocol:** (for process $i$)

/* claim the resource */
flags[$i$] = BUSY;
/* wait if the other process is using the resource */
while (flags[$j$] == BUSY) {

/* if waiting for the resource, also wait our turn */
  if (turn != $i$) {
    /* but release the resource while waiting */
    flags[$i$] = FREE;
    while (turn != $i$) {
      }
    flags[$i$] = BUSY;
  }
}

◆ **Exit protocol:** (for process $i$)

/* pass the turn on, and release the resource */
turn = $j$;
flags[$i$] = FREE;
Peterson’s Algorithm

- **Approach** → similar to Dekker’s algorithm; after setting our flag we immediately give away the turn; By waiting on the **and** of two conditions, we avoid the need to clear and reset the flags.

- **Initialization:**

  ```c
  typedef char boolean;
  shared boolean flags[n - 1];
  shared int turn;
  ...
  turn = i;
  ...
  flags[i] = FREE;
  ...
  flags[j] = FREE;
  ```

- **Entry protocol:** (for process $i$)
Peterson’s Algorithm

◈ Entry protocol: (for process \(i\))

/* claim the resource */
flags\([i]\) = BUSY;

/* give away the turn */
turn = \(j\);
/* wait while the other process is using the resource *and* has the turn */
while ((flags\([j]\) == BUSY) && (turn != \(i\))) {} 

◈ Exit protocol: (for process \(i\))

/* release the resource */
flags\([i]\) = FREE;
Multi-Process Solutions

Dekker’s and Peterson’s algorithms → can be generalized for N processes, however:

- N must be fixed and known in advance
- Again, the algorithms become too much complicated and expensive

*Implementing a mutual exclusion mechanism is difficult!*

Bakery Algorithm

- **Goal** – Solve the CS problem for n processes
- **Approach** – Customers take numbers → lowest number gets service next (*here service means entry to the CS*)
Bakery Algorithm

Approach → The entering process checks all other processes sequentially, and waits for each one which has a lower number. Ties are possible; these are resolved using process IDs.

Initialization:

typedef char boolean;
...

shared boolean choosing[n]
shared int num[n];
...

for (j=0; j < n; j++) {
    num[j] = 0;
}
...

Bakery Algorithm

◈ Entry protocol: (for process i)

/* choose a number */
choosing[i] = TRUE;
num[i] = max(num[0], ..., num[n-1]) + 1;
choosing[i] = FALSE;

/* for all other processes */
for (j=0; j < n; j++) {
  /* wait if the process is currently choosing */
  while (choosing[j]) {}
  /* wait if the process has a number and comes ahead of us */
  if ((num[j] > 0) &&
      ((num[j] < num[i]) ||
       (num[j] == num[i]) && (j < i))) {
    while (num[j] > 0) {}
  }
}

◈ Exit protocol: (for process i)

/* clear our number */
num[i] = 0;
Bakery Algorithm – Why choosing[i]?

- choosing[i] → What happens if you leave it out

- Mutual Exclusion → Violation

*Let's comment choosing[i] to examine the algorithm!*
Bakery Algorithm – Why choosing[i]?

- **Entry protocol:** (for process \(i\))

  ```c
  /* choose a number */
  choosing[i] = TRUE;
  num[i] = max(num[0], ..., num[n-1]) + 1;
  choosing[i] = FALSE;

  /* for all other processes */
  for (j=0; j < n; j++) {

    /* wait if the process is currently choosing */
    while (choosing[j]) {}

    /* wait if the process has a number and comes ahead of us */
    if ((num[j] > 0) &&
        ((num[j] < num[i]) ||
          (num[j] == num[i]) && (j < i))) {
      while (num[j] > 0) {}
    }
  }

- **Exit protocol:** (for process \(i\))

  /* clear our number */
  num[i] = 0;
```

- Consider 2 Process → \(P_0\) & \(P_1\)
- Same token \(P_1\) goes to CS after 2 iterations
- Later \(P_0\) blocked-unblocked → After 2 iterations enters CS!!
Hardware Solutions

- Use of hardware instructions to mask interrupts. The solution for \( N \) processes would be as simple as below:

  For Process \( i \)
  
  ```
  while (TRUE) {
    disableInterrupts();
    <Critical Section i>
    enableInterrupts();
    ...
  }
  ```

- Problems
  - Only one system-wide CS active at a time
  - No OS allows user access to privileged instructions
  - Not correct solution for multiprocessor machine
Hardware Solutions

- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions

- Test and Set Instruction

```java
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```
Hardware Solutions

Exchange Instruction

```c
void exchange(int register, int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```
Sample Program

```c
const int n = /* number of processes */;
int bolt;
void P (int i)
{
    while (true);
    {
        while (!testset (bolt))
            /* do nothing */
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}

void main()
{
    bolt = 0;
    parbegin (P(1), P(2), … , P(n));
}
```

Test and Set Instruction

```c
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```
Hardware Solutions

Advantages
- Applicable to any # processes, single/multiple processors sharing main memory
- Verification is simple/easy
- Can be used to support multiple CS

Disadvantages
- Busy waiting → Consumes processors time
- Starvation is possible → Selection of waiting process is arbitrary
- Deadlock is possible → The flag can only be reset by low priority process but has been preempted by high priority process
Semaphores

- $S$, Semaphore (an integer variable) → *Operation* $P$ and $V$

  - When a process executes $P(S)$, $S$ is decremented by one
    - $S \geq 0 \rightarrow$ Process continues execution; or
    - $S < 0 \rightarrow$ Process is stopped and put on a *waiting queue* associated with $S$.

  - When a process executes $V(S)$, $S$ is incremented by one
    - $S > 0 \rightarrow$ Process continues execution; or
    - $S \leq 0 \rightarrow$ Process is removed from the *waiting queue* and is permitted to continue execution; *process which evoked* $V(S)$ *can also continue execution*.

- $P$ and $V$ are indivisible/atomic → Cannot be interrupted in between
- Only one process can execute $P$ or $V$ at a time on given Semaphore
Implementation

- Busy Waiting
  - Two process solutions
  - Loop continuously in entry code
  - Problem → Multiprogramming systems
  - **Spinlock** → Spins while waiting for Lock
  - Useful
    - Multiprocessor System, No context switch time
    - Locks are expected to be held for short time

- Semaphore Solution
  - **P**, wait → block itself into a *waiting queue*
  - **V**, signal → *waiting queue* to *ready queue*
Implementation

```c
struct semaphore {
    int count;
    queue Type queue
};

void wait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        place this process in the s.queue;
        block this process
    }
}

void signal(semaphore s) {
    s.count++;    
    if (s.count <= 0) {
        remove a process \( p \) from the s.queue;
        place process \( p \) on the \textit{ready queue}
    }
```
Mutual Exclusion

➢ Sample Program

```c
const int n = /* number of processes */
semaphore s = 1;
void P (int i)
{
    while (true);
    {    
        wait(s);
         /* critical section */;
         signal(s);
        /* remainder */
    }
}
void main()
{
    parbegin (P(1), P(2), … , P(n));
}
```

Above program can also handle the requirement that more than one process be allowed inside CS at a time, How?
Mutual Exclusion

- Example - Three Process Accessing Shared Data using Semaphore

Note that normal execution can proceed in parallel but that critical regions are serialized.
Semaphore Types

- Integer/Counting/General Semaphore
- Binary Semaphore
- Fairest Policy $\rightarrow$ FIFO
- Order of removing process from waiting queue
  - Strong Semaphore $\rightarrow$ Includes policy definition
    - Guarantees freedom from Starvation
    - Typically provided by most OS
  - Weak Semaphore $\rightarrow$ Does not specify the order
Example
Possible Implementations

- No existing hardware implements P and V operations directly.
- Semaphores → Build up using hardware sync primitives.
- Uniprocessor Solution
  - Usually → disable interrupts.
- Multiprocessor Solution
  - Use hardware support for atomic operations.

Possible Usage

- Mutual Exclusion → Initialize semaphore to one.
- Synchronization → Initialize semaphore to zero.
- Multiple instances → Initialize semaphore to # of instances.
Two Possible Implementations

```c
wait(semaphore s)
{
    disable interrupts

    s.count--;
    if (s.count < 0)
    {
        place this process in the s.queue;
        block this process and enable interrupts
    }
    else
    {
        enable interrupts
    }
}

signal(semaphore s)
{
    disable interrupts

    s.count++;  
    if (s.count <= 0)
    {
        remove a process p from the s.queue;
        place process p on the ready queue
    }
    enable interrupts
}
```
The Producer/Consumer Problem

Semaphore `freeSpace`, initially `n`
Semaphore `availItems`, initially `0`

© Producer process
item `nextProduced`;

while (1) {
    wait(`freeSpace`);
    `buffer[in]` = `nextProduced`;
    `in` = (`in`+1) mod `n`;
    signal(`availItems`);
}

© Consumer process
item `nextConsumed`;

while (1) {
    wait(`availItems`);
    `nextConsumed` = `buffer[out]`;
    `out` = (`out`+1) mod `n`;
    signal(`freeSpace`);
}
Deadlock and Starvation

- **Deadlock**

- Let \( S \) and \( Q \) be two semaphores initialized to 1

\[
\begin{align*}
P_0 & \quad P_1 \\
\text{wait}(S); & \quad \text{wait}(Q); \\
\text{wait}(Q); & \quad \text{wait}(S); \\
\vdots & \quad \vdots \\
\text{signal}(S); & \quad \text{signal}(Q); \\
\text{signal}(Q) & \quad \text{signal}(S);
\end{align*}
\]

- **Starvation** – indefinite blocking; A process may never be removed from the semaphore queue in which it is suspended.
Implementing $S$ as a Binary Semaphore

- Data structures:
  
  ```
  binary-semaphore S1, S2;
  int C:
  ```

- Initialization:
  
  ```
  S1 = 1
  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);
  ```
Problems with Semaphores

- The \( P(S) \) and \( V(S) \) signals are scattered among several processes. Therefore it's difficult to understand their effects.

- Incorrect usage → timing errors (difficult to detect; only with some particular execution sequence which are rare)

- One bad process or programming error can kill the whole system or put the system in deadlock

**Solution?**

**High-level language constructs**

- Critical Regions, Eventcounts, Sequencers, Path Expressions, Serializers, Monitors, ...

A fundamental high-level synchronization construct → *Monitor* type
Monitor

- A monitor type presents a set of *programmer defined operations* which can provide *mutual exclusion within the monitor*
  - Procedures
  - Initialization code
  - Shared data

- **Monitor Properties**
  - Shared data can only be accessed by monitors procedures
  - Only one process at a time can execute in the monitor (executing a monitor procedure)

- Shared data may contain condition variables
A monitor type presents a set of programmer defined operations which can provide mutual exclusion within the monitor.

**Procedures**

- **Initialization code**
- **Shared data**

- Monitor Properties
  - Shared data can only be accessed by monitors
  - Only one process at a time can execute in the monitor (executing a monitor procedure)

**Example Monitor Code**

```plaintext
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }  
    procedure body P2 (...) {
        ...
    }  
    procedure body Pn (...) {
        ...
    }
    {  
        initialization code  
    }
}
```
**Condition Variables**

- Condition variables → To allow a process to wait in a monitor
- Condition variables can only be used with following operations
  - \( \text{Condition} : x, y \)
    - Declaring a condition variable
  - \( x.\text{wait} \)
    - Process invoking \( x.\text{wait} \) is suspended until another process invokes \( x.\text{signal} \)
  - \( x.\text{signal} \)
    - Resumes exactly one suspended process. If no process is suspended this operation has no effect
- If \( x.\text{signal} \) is evoked by a process \( P \), after \( Q \) → suspended
  - Signal and Wait
  - Signal and Continue
- Resuming processes within monitor; \( x.\text{wait}(c) \) → conditional-wait
Monitor Architecture

- Shared data
- Queues associated with $x$, $y$ conditions
  - $x$ queue
  - $y$ queue
- Operations
- Initialization code
- Entry queue
Classical Synchronization Problems

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

- Example of large class of concurrent-control problems
- Provide deadlock-free and starvation-free solution
- Chopstick $\rightarrow$ Semaphore
  - `semaphore chopstick[5];`
  - Initially `chopstick \rightarrow 1`
Dining-Philosophers Problem

Philosopher $i$:

```c
   do {
      wait(chopstick[i])
      wait(chopstick[(i+1) mod 5])
      ...  
      eat
      ...  
      signal(chopstick[i]);
      signal(chopstick[(i+1) mod 5]);
      ...  
      think
      ...  
   } while (1);
```

Problem $\rightarrow$ Deadlock
Dining-Philosophers Problem

- Possible solutions against deadlock
  - Allow at most 4 philosophers to sit simultaneously
  - Allow a philosopher to pick chopstick only if both chopsticks are available,
  - Odd philosopher → first left then right chopstick

- Satisfactory solution must guard against Starvation

  Deadlock-free solution does not eliminate possible starvation
Dining Philosophers Example

- Deadlock-free solution using monitor
- Chopsticks pick up → Only if both of them are available
  - Distinguish among 3 states of a philosopher

```c
monitor dp {
enum {thinking, hungry, eating} state[5];
condition self[5]; /* delay yourself when hungry but unable to obtain chopsticks */
void pickup(int i) /* Next Slide */
void putdown(int i) /* Next Slide */
void test(int i) /* Next Slide */
void init() {
    for (int i = 0; i < 5; i++)
        state[i] = thinking;
        state [(i+4) mod 5] != eating &&
        state [(i+1) mod 5] != eating
```

EEL 602
Dining Philosophers Example

monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    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) mod 5);
        test((i+1) mod 5);
    }

    void test(int i) {
        if ((state[(i + 4) mod 5] != eating) &&
            (state[i] == hungry) &&
            (state[(i + 1) mod 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }

    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}

Problem?
/* program diningphilosophers */
semaphore fork [5] = {1};
int i:
void philosopher (int i)
{
    while (true)
    {
        think ();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat ();
        signal (fork[i]);
        signal (fork [(i+1) mod 5]);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2), philosopher (3), philosopher (4));
}

First Solution - Dining Philosophers
Second Solution - Dining Philosophers

```c
/* program diningphilosophers */
semaphore fork [5] = {1};
semaphore room = {4};
int i:
void philosopher (int i)
{
    while (true)
    {
        think ();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat ();
        signal (fork[i]);
        signal (fork [(i+1) mod 5]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2), philosopher (3), philosopher (4));
}
```

Readers-Writers Problem

- File/Record is to be shared among several concurrent processes
- Many readers, Exclusively one writer at a time

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readers</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Writers</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- Several variations
  - No reader should wait for other readers to finish simply because a writer is waiting
  - Once a writer is ready, writer performs its write ASAP
- Possible starvation
- Solution → First variation
  ```
  int readcount = 0;
  semaphore mutex,
     initially 1
  semaphore wrt,
     initially 1
  ```
Readers-Writers Problem

**Writer:**
- `wait(wrt)`
- ... writing is performed
- `signal(wrt)`

**Reader:**
- `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 only in the system:
- \textit{wsem} set
- no queues

Writers only in the system:
- \textit{wsem} and \textit{rsem} set
- Writers queues on \textit{wsem}

Last Solution, Writers $\rightarrow$ Starvation

No new readers are allowed to access the data once at least one writer has declared a desire to write

/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1)
            semWait (wsem);
        semSignal (z);
        semSignal (rsem);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0)
            semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true)
    {
        semWait (y);
        writecount++;
        if (writecount == 1)
            semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0)
            semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
Readers-Writers Problem

Readers only in the system:
- \textit{wsem} set
- no queues

Writers only in the system:
- \textit{wsem} and \textit{rsem} set
- Writers queues on \textit{wsem}

Both Readers and Writers with Read First:
- \textit{wsem} set by reader
- \textit{rsem} set by writer
- all writers queues on \textit{wsem}
- one reader queues on \textit{rsem}
- other readers queues on \textit{z}

Both Readers and Writers with write First
- \textit{wsem} set by writer
- \textit{rsem} set by writer
- writers queues on \textit{wsem}
- one reader queues on \textit{rsem}
- other readers queues on \textit{z}

/*program readersandwriters*/
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
        semWait (z);
        semWait (rsem);
        readcount++;
        if (readcount == 1)
            semWait (wsem);
        semSignal (z);
        semSignal (rsem);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0)
            semSignal (wsem);
        semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
        semWait (y);
        writecount++;
        if (writecount == 1)
            semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0)
            semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
Synchronization in Pthreads

- **Pthread API**
  - Mutex locks, condition variables, read-write locks for thread synchronization

- **Pthreads Mutex Locks**

  ```c
  # include <pthread.h>
  pthread_mutex_t mutex;

  /* create the mutex lock */
  pthread_mutex_init(&mutex, NULL);

  /* acquire the mutex lock */
  pthread_mutex_lock(&mutex);

  /**** Critical Section ****/

  /* release the mutex lock */
  pthread_mutex_unlock(&mutex);
  ```
Synchronization in Pthreads

- **Pthread Semaphores**
  - Include `<semaphore.h>`
  ```c
  #include <semaphore.h>
  
  sem_t sem;
  
  /* create the semaphore and initialize to 8 */
  sem_init(&sem, 0, 8)
  
  wait() → sem_wait()
  signal() → sem_post()
  ```

- **Protecting CS using semaphore**
  ```c
  #include <semaphore.h>
  sem_t mutex;
  
  /* create the semaphore */
  sem_init(&mutex, 0, 1);
  
  /* acquire the semaphore */
  sem_wait(&mutex);
  
  /* Critical Section */

  /* release the semaphore */
  sem_post(&mutex);
  ```
Synchronization using Win32 API

- **Win 32 mutex Locks**

```c
#include <windows.h>
HANDLE Mutex;

/* create a mutex lock*/
Mutex = CreateMutex(NULL, FALSE, NULL);

/* Acquiring a mutex lock created above */
WaitForSingleObject(Mutex, INFINITE);

/* Release the acquired lock */
ReleaseMutex(Mutex);
```

- **Win 32 Semaphores**

```c
#include <windows.h>
HANDLE Sem;

/* create a semaphore*/
Sem = CreateSemaphore(NULL, 1, 5, NULL);

/* Acquiring the semaphore */
WaitForSingleObject(Semaphore, INFINITE);

/* Release the semaphore, signal() */
ReleaseSemaphore(Sem, 1, NULL);
```
Synchronization in Linux

- Current versions → processes running in kernel mode can also be preempted, when higher priority process available

- Linux Kernel → Spinlocks and Semaphores for locking in kernel

- Locking mechanisms
  - Uniprocessor → Enabling and disabling kernel preemption
    - preempt_disable(), preempt_enable()
  - Multiprocessor → Spinlocks
    - Kernel is designed such that spinlocks are held only for short duration
Synchronization in Linux

- **Atomic Operations → Special data type, atomic_t**
  - ATOMIC_INT (int i), int atomic_read(atomic_t *v)
  - void atomic_add(int i, atomic_t *v)
  - void atomic_sub(int i, atomic_t *v)

- **Spinlocks → Only one thread at a time can acquire spinlock**
  - void spin_lock(spinlock_t *t)
  - void spin_unlock(spinlock_t *lock)

- **Reader-Writer Spinlock → Exclusive access to spinlock that intends to update the data structure, favors readers**

- **Semaphores → Binary, Counting, Reader-Writer**
  - void sema_init(struct semaphore *sem, int count)
  - void init_MUTEX(struct semaphore *sem)
  - void init_MUTEX_locked(struct semaphore *sem)
  - Void init_rwsem(struct rw_semaphore *sem)
Synchronization in Windows XP

- Kernel access global resources
  - Uniprocessor → Temporarily *masks interrupts* for all interrupt handlers
  - Multiprocessor
    - Uses spinlocks to protect access to global resources
    - Spinlocks → only to protect short code segment
    - A thread will never be preempted while holding a spinlock

- Thread synchronization outside kernel → *dispatcher objects*
  - Using dispatcher objects, threads synchronize using different mechanisms (*mutexes, semaphores, events, timers*)
  - Singled state, Nonsingled state

- Dispatcher objects may also provide *events* → much like a condition variable
Minor II

- Syllabus → Scheduling, Synchronization, Deadlocks

- Open Book/Notes
  - Can bring your own notes
  - Can also bring class lecture slides
  - Exchange of notes/materials → Strictly prohibited
  - No textbook is allowed
  - No xerox of book(s) is allowed

- Type of questions → Remains Open!

Good Luck!