Université d'Ottawa | University of Ottawa

CSI 3131 Operating Systems

uOttawa

www.uOttawa.ca

PROCESS SYNCHRONIZATION

uOttawa

Outline

- The critical section (CS) problem
- Critical Problem solutions
- Classic Thread Synchronization problems
- Java Threads Synchronization
- Pthread Synchronization

uOttawa

Issues related to concurrency

- Concurrent threads sometimes share data (files and common memory) and resources
- These are cooperative tasks
- If access is not controlled, the result of the program execution may depend on the order of interleaving of instruction execution (non-deterministic).
- A program can give different results from one execution to another and sometimes undesirable results.

Issues related to concurrency

- Very simplified queue:

```
 enqueue(data):
     qData[inPos] = data;
     inPos++;
dequeue():
     data = qData[outPos];
     outPos++;
```

- What happens if several processes/threads try to use the queue functions concurrently?

Issues related to concurrency

- The **enqueue & dequeue** operations are not Atomic, which means it may take several instructions to perform them
  - For example inPos++may be implemented as:
    - ✓Register1 = inPos
    - ✓Register1 = Register1 + 1
    - ✓inPos = Register1
- What will happen if two (or more) threads interleave the **enqueue or dequeue** operations?
  - The outcome is incorrect and undetermined

uOttawa

Issues related to concurrency

- In summary:
  - Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources
  - If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data
  - The results of actions performed by concurrent processes will then depend on the order in which their execution is interleaved – race condition
- Solution: abstract the danger of concurrent modification of shared variables into the critical-section problem

Critical-section problem

- The piece of code modifying the shared variables where a thread/process needs exclusive access to guarantee consistency is called critical section
- The general structure of each thread/process can be seen as follows:

```
while (true) {
  entry_section
  critical section (CS)
  exit_section
  remainder section (RS)
}
```

- CS & RS are well defined (code),

uOttawa

Critical-section problem

- We want to design entry & exit section satisfying the following requirements:
  - Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
  - Progress - If there exist some processes wishing to enter their CS and no process is in their CS, then one of them will eventually enter its critical section.
    - ✓ No deadlock
    - ✓ No interference – a process termination in the RS, does not restrict other processes access the CS
    - ✓ Assume that a thread/process always exits its CS.

Critical-section problem

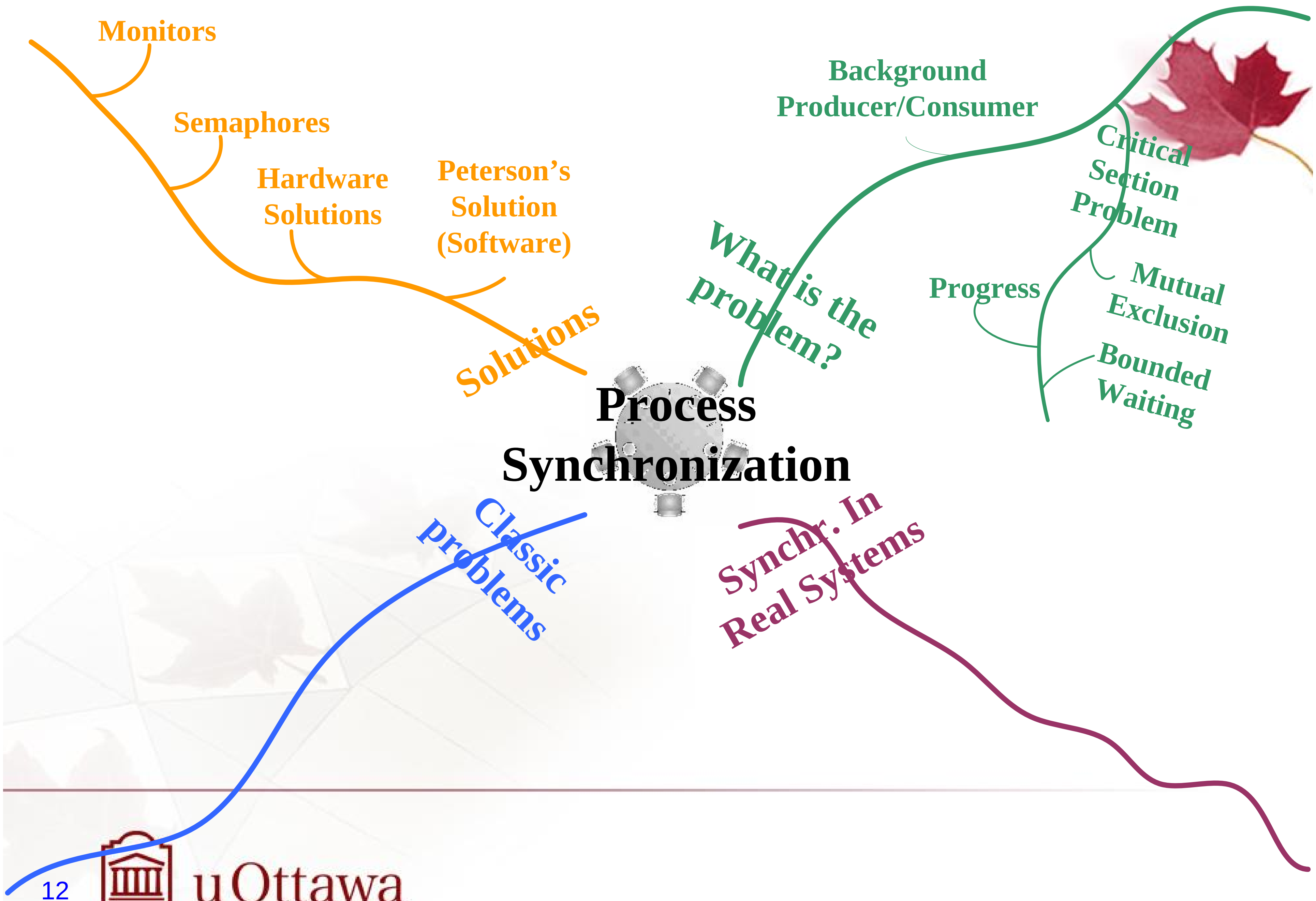
- We want to design entry & exit section satisfying the following requirements:
  - Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - No famine.

uOttawa

Critical-section problem

- Assumptions
  - Each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes
  - Many CPUs may be present but memory hardware prevents simultaneous access to the same memory location
  - No assumption about the order of interleaved execution

uOttawa


Process Synchronization
What is the problem?
Background Producer/Consumer
Critical Section Problem
Mutual Exclusion
Progress
Bounded Waiting
Solutions
Monitors
Semaphores
Hardware Solutions
Peterson's Solution (Software)
Classic problems
Synchr. In Real Systems

uOttawa

Critical Section Problem Solution

- Software solutions
  - algorithms who’s correctness does not rely on any other assumptions
  - Peterson’s algorithm
- Hardware solutions
  - rely on some special machine instructions
  - testAndSet, xchng
- OS provided solutions
  - higher level primitives (implemented usually using the hw solutions) provided for convenience by the OS/library/language
  - semaphores, monitors

Critical Section Problem Solution

- All solutions are based on atomic access to memory: memory at a specific address can only be affected by one instruction at a time, an thus one thread/process at a time.
- All solutions are based on the existence of atomic instructions, that operate as critical sections.

uOttawa

Software Solutions

- Lets start with a simpler problem – only two tasks
  - Two tasks, T0 and T1 (also Ti and Tj, where i != j)
  - We want to design the entry and exit sections

```
while (true)
{
   entry_section
   critical section (CS)
   exit section
   remainder section (RS)
}
```

Algorithm1

- Have a shared variable turn indicating whose turn it is now. It is initialized to 0 or 1, does not matter.
- Task T_i may enter the critical section if and only if turn = i
- After exiting the critical section, turn is set to the other value, to let the other task gain access to its critical section
- T_i can be in a busy wait if T_j is in its CS.

```
Task T0:
while(true)
{
   while(turn!=0){/*bw*/}
   Critical Section
   turn=1;
   Remainder Section
}
```

```
Task T1:
while(true)
{
   while(turn!=1){/*bw*/}
   Critical Section
   turn=0;
   Remainder Section
}
```

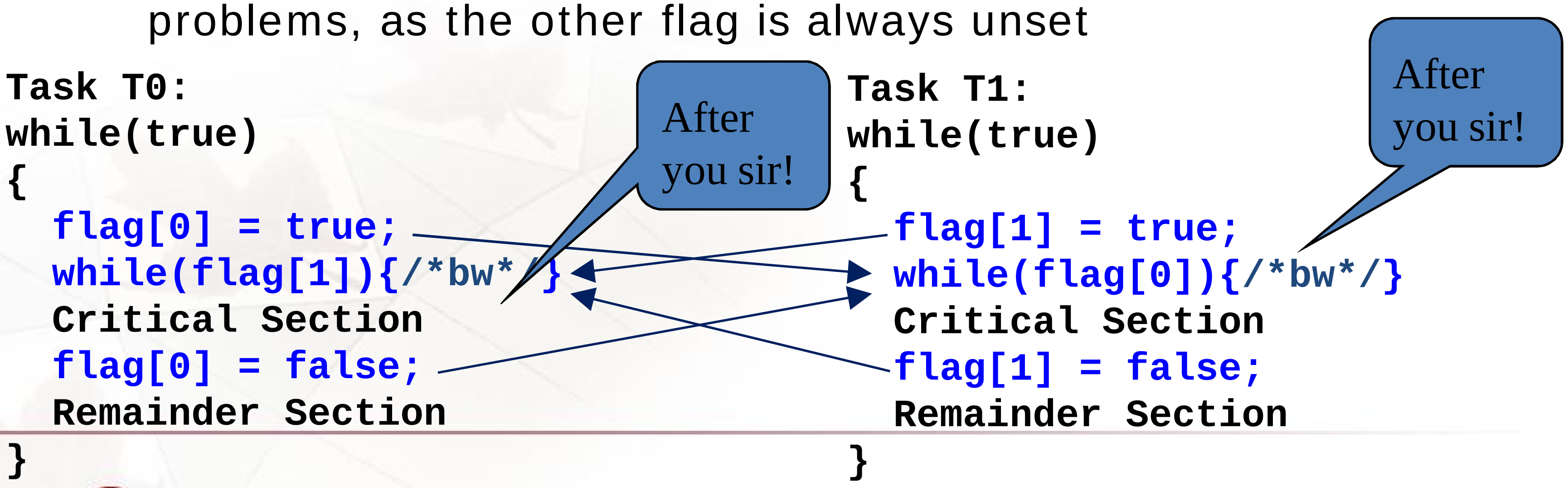
Algorithm1 - discussion

- Does it ensure mutual exclusion?
  - Yes, at any moment, turn has only one value, and if a task Ti is in its critical section, then turn = i
- Does it satisfy the bounded waiting requirement?
  - Yes, the tasks alternate
- Does it satisfy the progress requirement?
  - No, because it requires strict alternation of critical sections (CS), If a task requires its CS more often then the other, it cannot get it.

uOttawa

Algorithm2

- Lets introduce shared variables flag[0] and flag[1] to indicate whether task0 and task1 wants to enter the CS
- Task i sets to true the flag[i] before trying to enter the CS and set to false flag[i] after completing the CS.
- Ti does not enter the CS while flag[j] is true.
- If only one task wants to enter the CS several times, it has no problems, as the other flag is always unset



```
Task T0:
while(true)
{
   flag[0] = true;
   while(flag[1]){/*bw*/}
   Critical Section
   flag[0] = false;
   Remainder Section
}
```

```
Task T1:
while(true)
{
   flag[1] = true;
   while(flag[0]){/*bw*/}
   Critical Section
   flag[1] = false;
   Remainder Section
}
```

Algorithm2 - discussion

- Does it ensure mutual exclusion?
  - Yes, a process in CS has its flag set, but it does not enter CS if the flag of the other process is set
- Does it satisfy the bounded waiting requirement?
  - No, If the task Ti is scheduled after the Tj has set flag[j] to false but before it set again flag[j] to true, Ti will enter its CS.
- Does it satisfy the progress requirement?
  - What happens after the following sequence of instructions:
    - ✓T0: flag[0] = true;
    - ✓T1: flag[1] = true;
  - No progress, deadlock occurs.

Agorithm3

- Use the ideas from both Algorithm 1&2
  - Use the flag i to indicate willingness to enter CS
  - But use turn to let the other task enter the CS

```
Task T0:
while(true)
{
  flag[0] = true;  // T0 wants in
  turn = 1;// give a chance to T1
  while
   (flag[1]==true&&turn==1){}
  Critical Section
  flag[0] = false;   // T0 wants out
  Remainder Section
}
```

```
Task T1:
while(true)
{
  flag[1] = true;   // T1 wants in
  turn = 0;// give a chance to T0
  while
  (flag[0]==true&&turn==0){}
  Critical section
  flag[1] = false; // T1 wants out
  Remainder Section
}
```

Agorithm3 - discussion

- Does it ensure mutual exclusion?
  - Yes, a task enters CS only if it is its turn when both want in.
- Does it satisfy bounded waiting?
  - Yes, when a task is in its RS, the other task can enter its CS (flag of other task is false).
- Does it satisfy the progress requirement?
  - A task will enter CS if and only if it is its turn OR the other task does not want to enter CS
  - If the other task does not want to enter, I can go in, great. What if both want to enter?

uOttawa

Agorithm3 - discussion

- What happens if both tasks want to enter?
  - ✓ Both will set their respective flags to true
  - ✓ Both will set turn to the opposite value
  - ✓ Only one of them (the second one, for example T1) will really succeed in this, as turn can take only one value
- This means T0 will enter the CS!
- And after T0 exits it, it will set its flag to false and T1 will enter CS.
- The algorithm is called Peterson's algorithm

uOttawa

Agorithm3 - discussion

- A solution to the CSP is robust to tasks failing in the RS. If a task fails in the CS, the other task shall be blocked.
- The Peterson algorithm can be generalised to more than 2 tasks, but there exists other more elegant algorithms – the baker's algorithm
- For tasks with shared variable to work properly, all threads involved must used the same synchronization algorithm

uOttawa

Critique of software solutions

- Difficult to program! And to understand!
- The subsequent solutions we shall study are based on the existence of specialized instructions, which makes the work easier.
- Threads/processes that desire entry into their CS are busy waiting; which consumes CPU time.
  - For long critical sections, it is preferable to block threads/processes that must wait….

uOttawa

Hardware Solution1

- Disable interrupts - Simple solution
  - A process would not be pre-empted in its CS

Discussion

- Efficiency deteriorates: when a process is in its CS, it’s impossible to interlace the execution of other processes in their RS.
- Loss of interruptions
- On a multiprocessor system: mutual exclusion is not assured.
- A solution that is not generally acceptable.

```
Process Pi:
while(true)
{
  disable interrupts
  critical section
  enable interrupts
  remainder section
}
```

uOttawa

Hardware Solution2

- Specialized Instructions
  - Basic Idea: when a thread or process access an address of memory, no other can access the same memory address at the same time.
  - Extension: define machine instructions that perform multiple actions (ex: reading and writing) in the same memory location atomically (indivisible).
- An atomic instruction can only be executed by a single thread/process at a time (even in the presence of multiple processors).

The test-and-set instruction

- Use hardware enabled specialized instruction called test_and_set that is guaranteed by hardware to be Atomic instruction

```
bool testset(int *var)
{
  if (*var==0)
  {
    *var=1;
    return true;
  }
  else
  {
    return false;
  }
}
```

Atomic Instruction

```
Task Ti: // using test_and_set for
         // mutual execlusion
  while testset(&b)==false {};
  CS //enter when true
  b=0;
  RS
```

uOttawa

The test-and-set instruction

- Pros:
  - Mutual exclusion is assured:
    - ✓ if T_i enters the CS, the other T_j are busy waiting
- Cons:
  - still using busy waiting.
  - Can easily attain mutual exclusion, but needs more complex algorithms to satisfy the other requirements to the Critical Section Problem
  - When T_i leaves its CS, the selection of the next T_j is arbitrary: no bounded waiting: starvation is possible.

uOttawa

The xchg instruction

- Certain CPUs (ex: Pentium) offer the instruction xchg(a,b) that exchanges atomically two variables
- xchg(a,b) suffers from the same problems as the test-and-set

```
Task Ti:
while
{
  k = 1
  while(k!=0)xchg(k,b);
  Critical Section
  xchg(k,b);
  Remainder Section
}
```

uOttawa

The xchg instruction

- The variable b is initialized to 0
- Each T_i owns a local variable k
- The T_i that can enter its CS is the one that finds b=0
- The T_i excludes all others by assigning 1 to b
- When the CS is occupied, both k and b will have the value 1 in tasks trying to enter its CS.
- But k is 0 in the task that has entered its CS.
- xchg(a,b) suffers from the same problems as the test-and-set

uOttawa

Drawback of software & hardware solutions

- Processes that are requesting to enter in their critical section are busy waiting (consuming processor time needlessly)
- If Critical Sections are long, it would be more efficient to block processes that are waiting…
  - Can the OS help us? It can block processes.

Solutions based on system calls

- Solutions seen so far are difficult to program and easily leads to bad code
- Need to find methods to avoid common errors, such as deadlocks, starvation, etc.
- Need some higher level functions.
- The subsequent methods we shall study use more powerful functions that the OS can provide with system calls

Semaphores – Version1 – spinlocks

- The semaphore is a simple integer variable S on which three operations are allowed:
  - Initialization to a positive value (including zero).
  - wait(S) also called acquire
  - signal(S) also called release
- The first version of semaphores studied use busy wait

Spinlocks: Busy Wait Semaphores

```
wait(S)
{
  while(S<=0);//no-op
  S--;
}
```

The sequence `S<=0, S--` *must be atomic.*

```
signal(S)
{
   S++;
}
```

The signal call must be atomic.

uOttawa

Spinlocks: Busy Wait Semaphores

- Easiest way to implement semaphores.
- Used in situations where waiting is brief or with multi-processors.
- When S=n (n>0), up to n processes will not block when calling wait().
- When S becomes 0, processes block in the call wait() up until signal() is called.
- A call to signal() unblocks a blocked process or increments the semaphore value.

Using semaphores in Critical Section

- Can be applied with n processes
  - Initialize S to 1, means only 1 process can enter CS
  - To allow k processes to enter CS, initialize S to k

```
do
{
   wait(S);
   /*Critical Section*/
   signal(S);
   /*Remainder Section*/
} while(TRUE);
```

uOttawa

Using semaphores for synchronization

- Process P1 has task S1, process P2 has task S2
- Tasks must be implemented in this order S1, S2
- Set Semaphore initial value S=0

```
Process P1:
  S1
  signal(S);
```

```
Process P2:
 wait(S);
 S2
```

Semaphores: observations

- When S becomes >0, the process that unblocks is the first to test S (random)
  - Possibility of famine exists
- Still using busy waiting
- With version 1 (spinlocks): S is always >= 0
  - It is possible to limit the values of the semaphore to 0 and 1, called the binary semaphore and also called the mutex (for mutual exclusion). Easier to implement
  - The more general semaphore is called the counting semaphore

Semaphores – Version2 – no busy wait

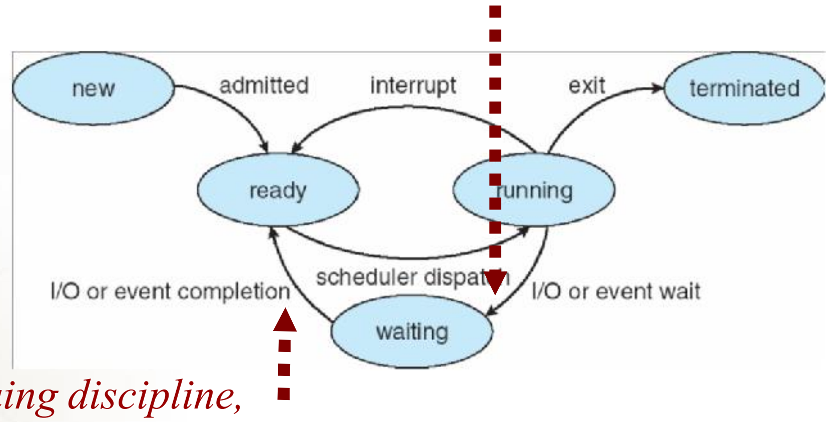
- When a process must wait for a semaphore to become greater than 0, place it in a queue of processes waiting on the same semaphore
- The queues can be FIFO, with priorities, etc. The OS controls the order in which processes are selected from the queue.
- Thus wait and signal become system calls similar to requests for I/O.
- There exists a queue for each semaphore similar to the queues defined for each I/O device.

Semaphore – Version 2

- The semaphore S now becomes a data structure:

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

A process waiting on a semaphore S, is blocked and placed in the queue S.list (its state changes to waiting).



Signal(S) removes (according to the queuing discipline, such as FIFO) a process from S.list and places it on the ready queue (process enters the ready state).

Implementation of wait()

```
wait(semaphore *S) {
  S->value--;
  if(S->value < 0) {
    add the process to S->list;
    block(); /*process goes to state waiting*/
  }
}
```

- This system call must be atomic
- When value becomes negative, its absolute value represents the number of blocked processes in list
- Recall that spinlock values never become negative (note that decrement and test operations are reversed)

uOttawa

Implementation of signal

```
signal(sempahore *S)  {
  S->value++;
  if(S->value <= 0) {
    /* processes are waiting */
    Remove process P from S->list;
    wakeup(P); /*process becomes ready*/
    }
 }
```

- This system call must be atomic
- The value of the semaphore can be initialised to a positive value (including 0).

Atomic execution of wait() and signal()

- wait() and signal() are in fact critical sections
- With single processor systems
  - Can disable interrupts, Operations are short (about 10 instructions)
- With SMP systems
  - Spinlocks
  - Other software and hardware CSP solutions.
- We have not eliminated busy waiting, but the busy waiting has been reduced considerably (to the wait() and signal() calls)

uOttawa

Atomic execution of wait() and signal()

- The semaphores (version 2), without busy wait, are used within applications that can spend long periods in their critical section (or blocked on a semaphore waiting for a signal) – many minutes or even hours
- Our synchronization solution is thus efficient

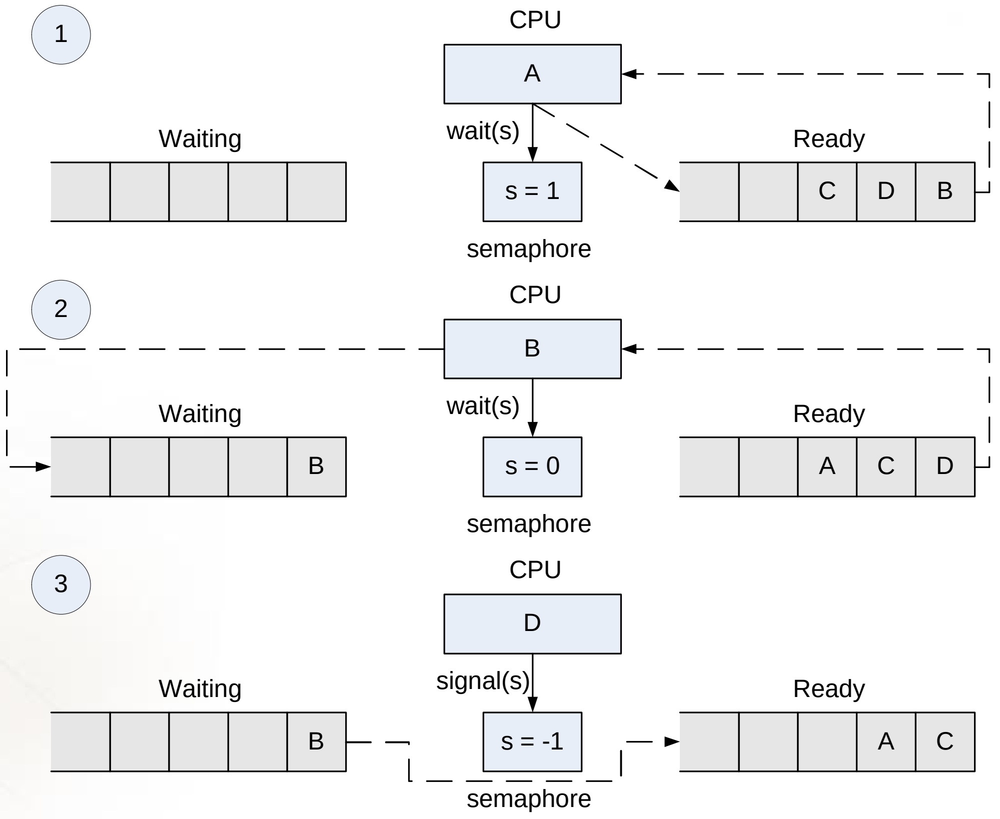
Example of semaphore operation

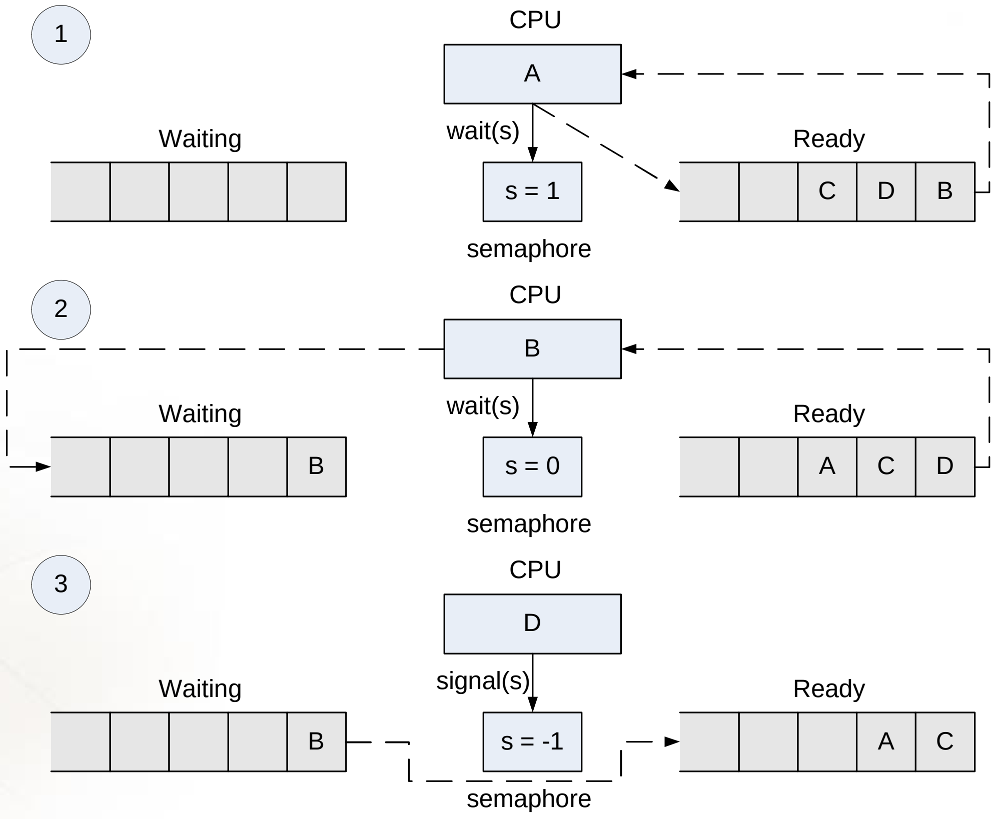
Process D produces some result and uses signal(s) to indicate the availability of results to other processes

Processes A, B, C: Consumes the results from D, and uses wait(s) to access them.

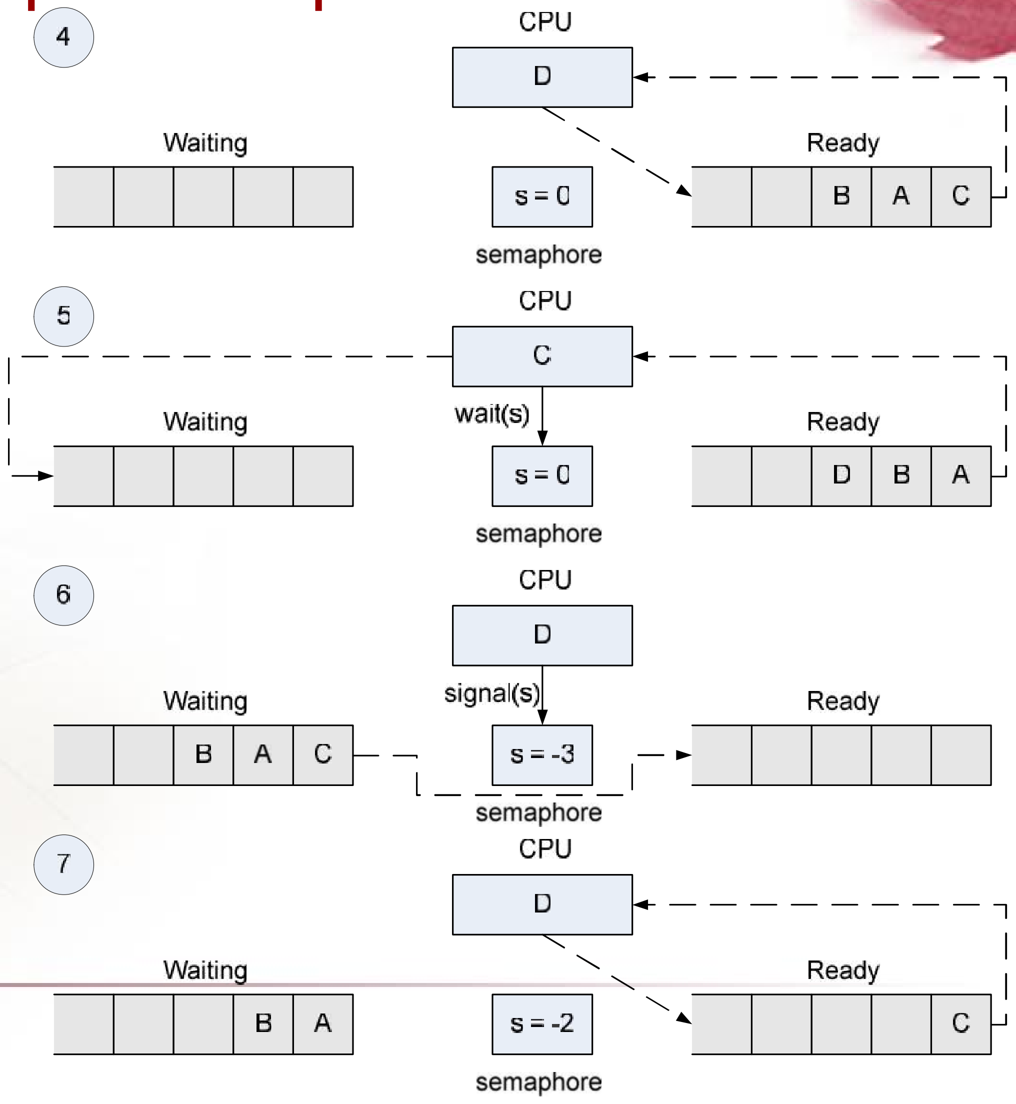
Scenario 1: D has already indicated that a result is ready and thus s=1

(Stallings)



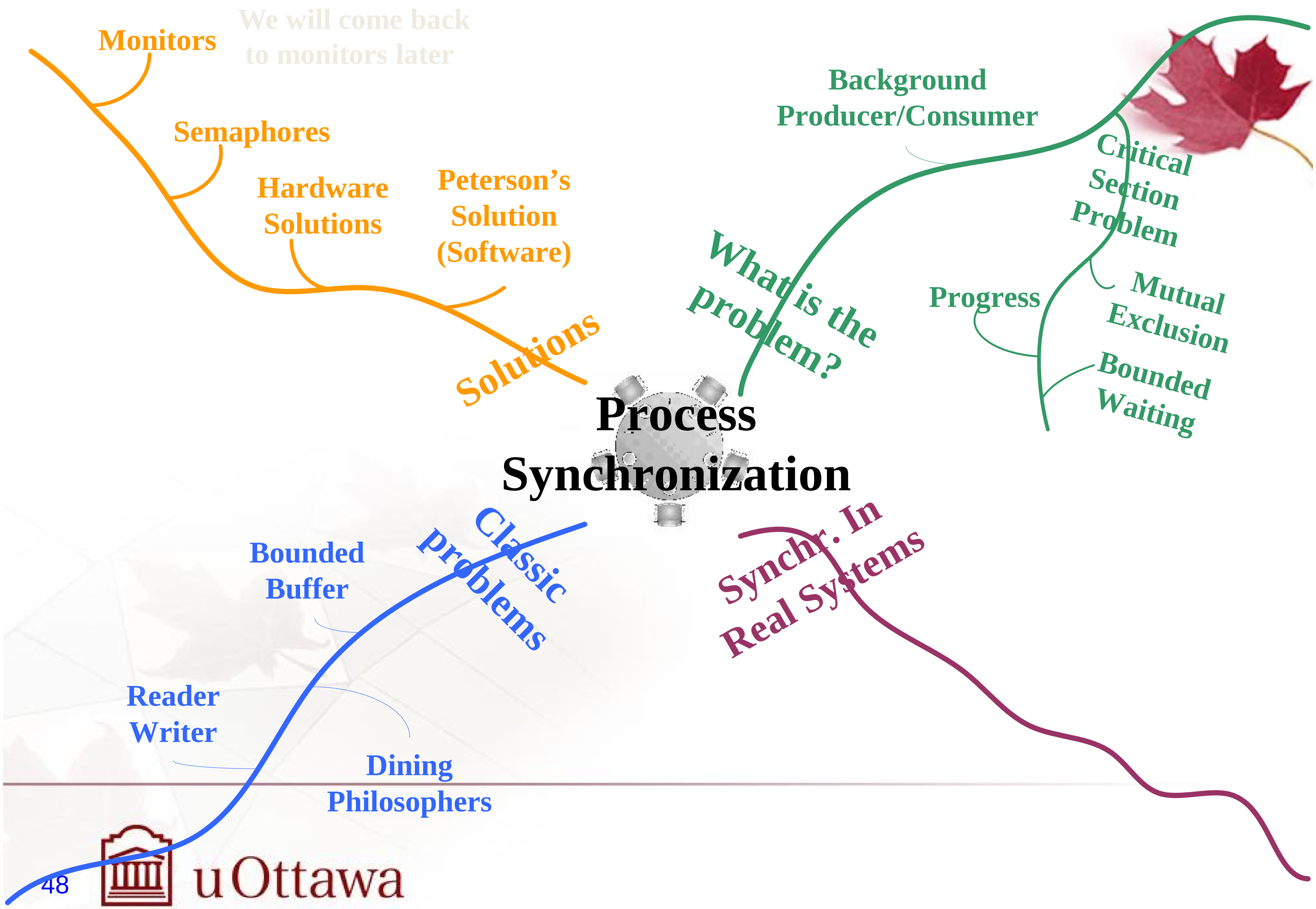


Example of semaphore operation



Issues associated with semaphores

- Starvation: consider what can happen if a LIFO queuing discipline were used in a semaphore.
- Deadlock: Given semaphores S and Q both initialized to 1, if 2 processes lock S & Q in reverse order a deadlock will occure


Process Synchronization
Solutions
Monitors
We will come back to monitors later
Semaphores
Hardware Solutions
Peterson's Solution (Software)
What is the problem?
Background Producer/Consumer
Critical Section Problem
Progress
Mutual Exclusion
Bounded Waiting
Classic problems
Bounded Buffer
Reader Writer
Dining Philosophers
Synchr. In Real Systems

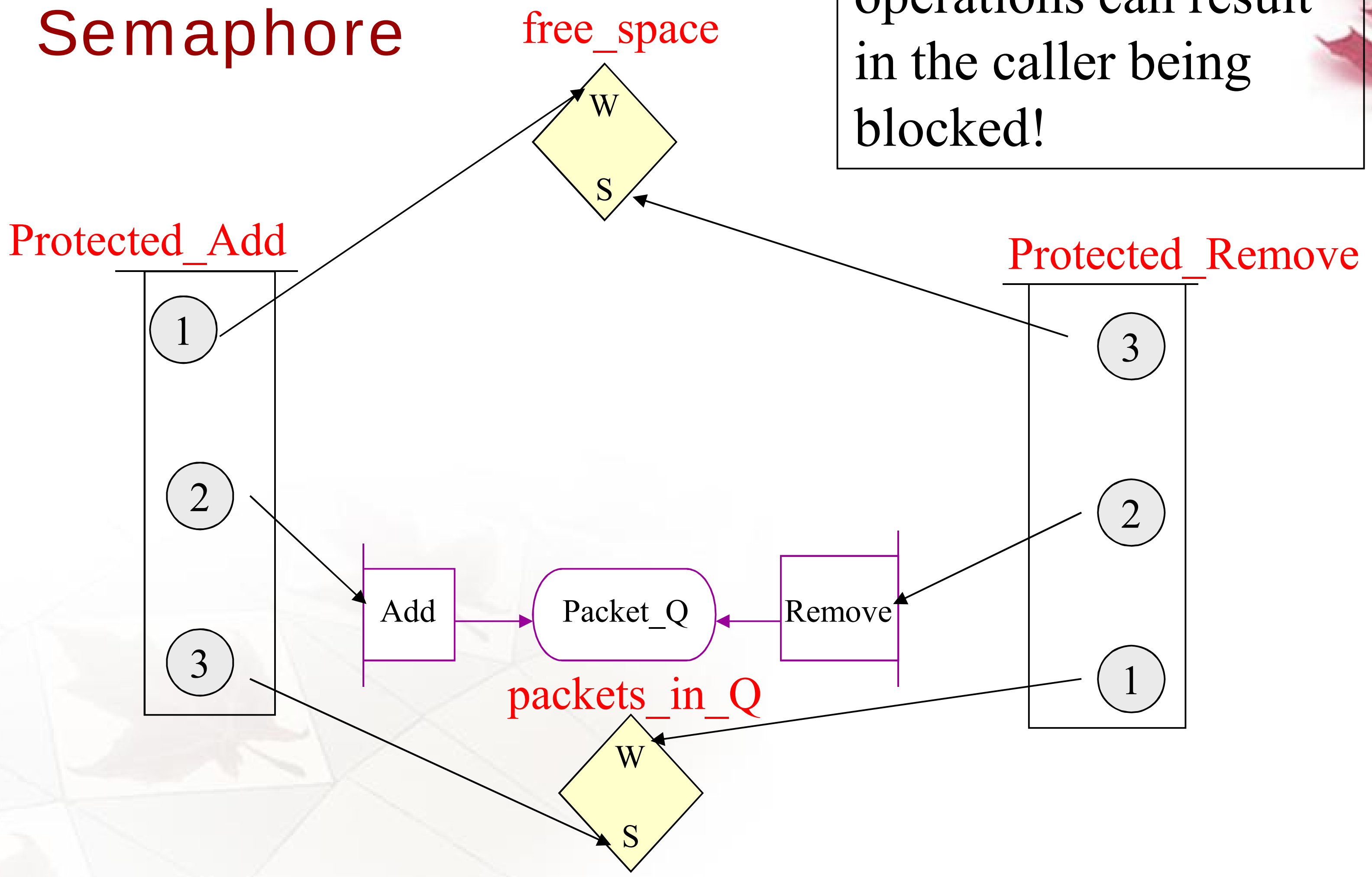
uOttawa

Classical Problems of Synchronization

- Producer/Consumer problem
- Reader/Writer Problem
- Dinning Philosopher Problem

Semaphore

calling the Protected operations can result in the caller being blocked!

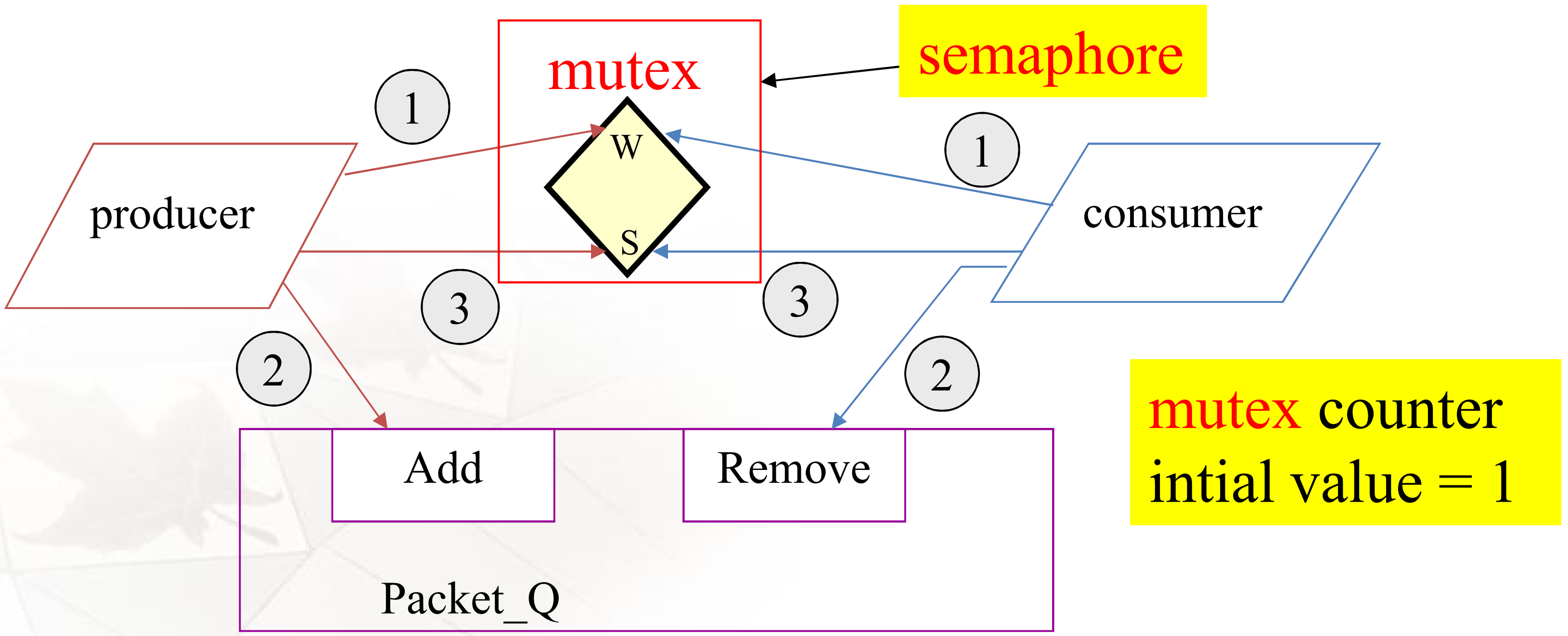


Producer and Consumer

```
semaphore fullspots, emptyspots;
fullspots.count := 0;
emptyspots.count := BUFLEN;
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);     { wait for a space }
  DEPOSIT(VALUE);
  release(fullspots);   { increase filled spaces }
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);      { make sure it is not empty }
  FETCH(VALUE);
  release(emptyspots); { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

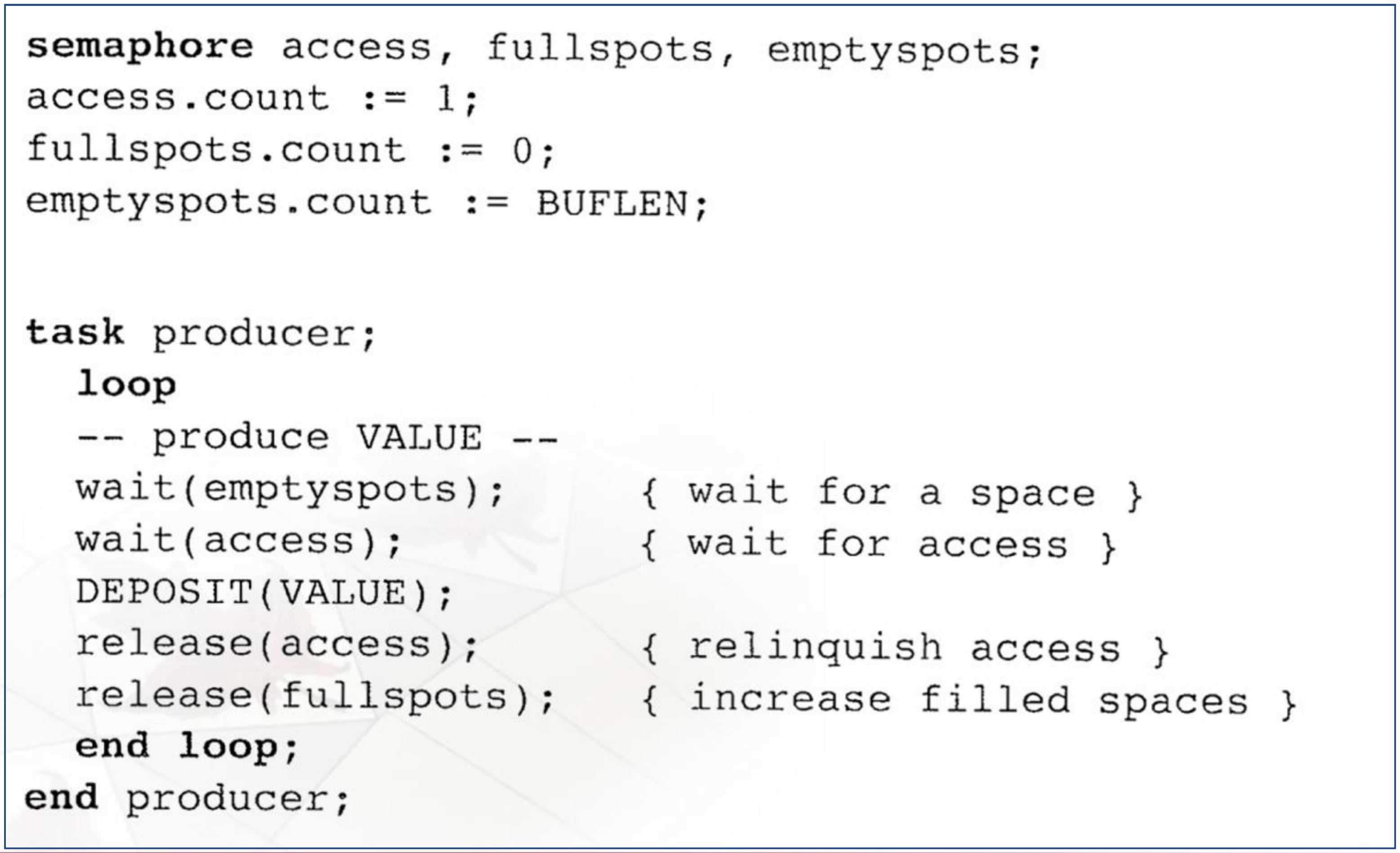
Mutual Exclusion (mutex)

- recall Stream-2-Pipe example: want mutually exclusive access to packet_Q



uOttawa

Adding Competition Synchronization

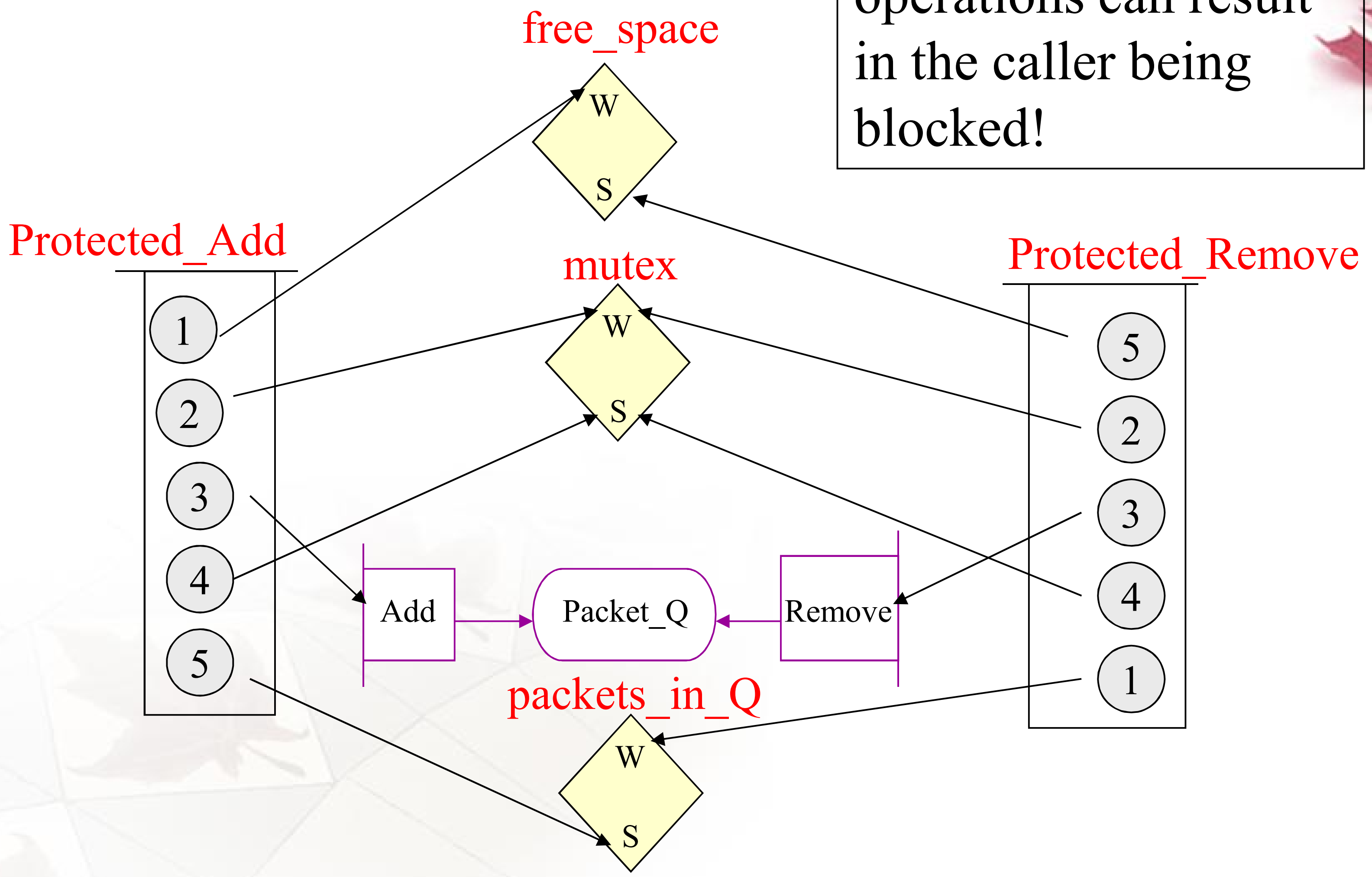


```
semaphore access, fullspots, emptyspots;
access.count := 1;
fullspots.count := 0;
emptyspots.count := BUFLEN;


task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);       { wait for a space }
  wait(access);           { wait for access }
  DEPOSIT(VALUE);
  release(access);        { relinquish access }
  release(fullspots);     { increase filled spaces }
  end loop;
end producer;
```

Adding Competition Synchronization

```
task consumer;
  loop
  wait(fullspots);       { make sure it is not empty }
  wait(access);          { wait for access }
  FETCH(VALUE);
  release(access);       { relinquish access }
  release(emptyspots);   { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

calling the Protected operations can result in the caller being blocked!



Readers-Writers Problem

- Two types of processes accessing a shared database
  - readers only reads the data, does not modify them
  - writers want to modify the data
- In order to maintain consistency, as well as efficiency, the following rules are used:
  - Several readers can access the database simultaneously
  - A writer needs to have an exclusive access to the database (has a critical section)

Readers-Writers Problem

- What to do if there are several readers in the system and a writer arrives? Many options:
  - While there is a reader active, do not have new readers wait (first readers-writers problem).
  - No new reader is admitted if there is a writer waiting (second readers-writers problem).
  - Both might lead to starvation

uOttawa

Readers-Writers Problem

- Have a semaphore wrt for allowing only one write process to modify the database
  - The writer process can simply use it, i.e.
  - wait(wrt), followed by CS, and then signal(wrt)
- How to prevent a writer entering when there is a reader?
  - The first reader grabs the semaphore wrt, i.e. executes wait(wrt) to prevent the writer from entering its CS
  - The last reader releases semaphore wrt (i.e. executed signal(wrt)
  - Now the writer can enter

Readers-Writers Problem

- We need a counter readers to know how many readers are in the system, to know when to grab and release the wrt semaphore
  - Only the first one and the last one should do that
- But then we need a second semaphore rmutex to guard access to readers

Readers-Writers Problem

Initialization:

```
wrt.value = 1;
rmutex.value = 1;
readers = 0;
```

Writer:

```
while(true)
{
   wait(wrt);          ⎫
   writer's CS;        ⎬ Mutual exclusion
   signal(wrt);        ⎭ for writer
}
```

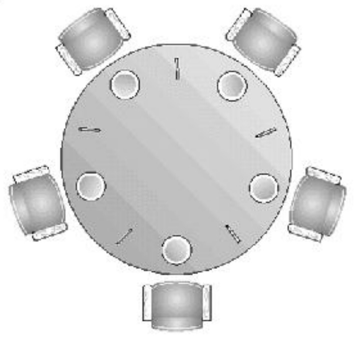
Reader:

```
while
{
                       ⎧ wait(rmutex);
Mutual exclusion       ⎪ readers++;
for readers            ⎨ if (readers == 1)
access                 ⎪   wait(wrt);
                       ⎩ signal(rmutex);
  reading database;
                       ⎧ wait(rmutex);
                       ⎪ readers--;
                       ⎨ if (readers == 0)
                       ⎪   signal(wrt);
                       ⎩ signal(rmutex);
}
```

uOttawa

The Dining Philosophers Problem

- 5 philosophers think, eat, think, eat, think ..
- In the center a bowl of rice.
- Only 5 chopsticks available.
- Require 2 chopsticks for eating.
- A classical synchronization problem
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



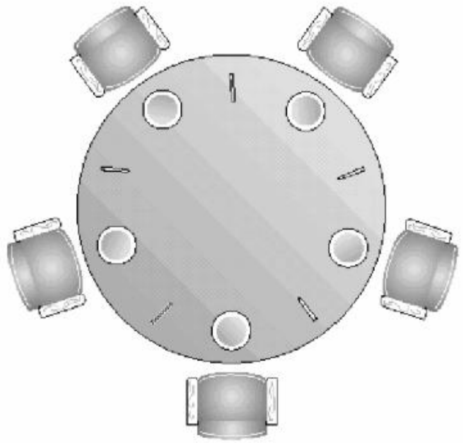
The Dining Philosophers Problem

- One semaphore per chopstick:
- So that two philosophers cannot grab the same chopstick simultaneously

```
semaphore chopst[5];
Initialization: chopst[i].value=1 for i = 0 to 4
```

```
Process Pi:
while(true)
{
 think();
 wait(chopst[i]);
 wait(chopst[(i+1)%5]);
 eat();
 signal(chopst[i+1)%5]);
 signal(shopst[i]);
}
```

What happens if each philosopher starts by picking his left chopstick? Deadlock!



uOttawa

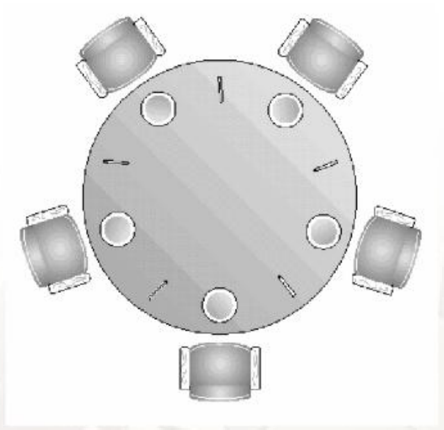
The Dining Philosophers Problem

- Solution: allow only 4 philosophers at a time to try to eat
  - Then 1 philosopher can always eat when the other 3 are holding 1 chopstick
  - use another semaphore T that would limit at 4 the number of philosophers trying to take a chopstick
    - ✓ Initialize T.value = 4
- free of deadlock and starvation

The Dining Philosophers Problem

```
Initialization:
Chopst[i].value = 1;
T.value = 4;
```

```
Process Pi:
while(true)
{
 think();
 wait(T);
 wait(chopst[i]);
 wait(chopst[(i+1)%5]);
 eat();
 signal(chopst[(i+1)%5]);
 signal(chopst[i]);
 signal(T);
}
```

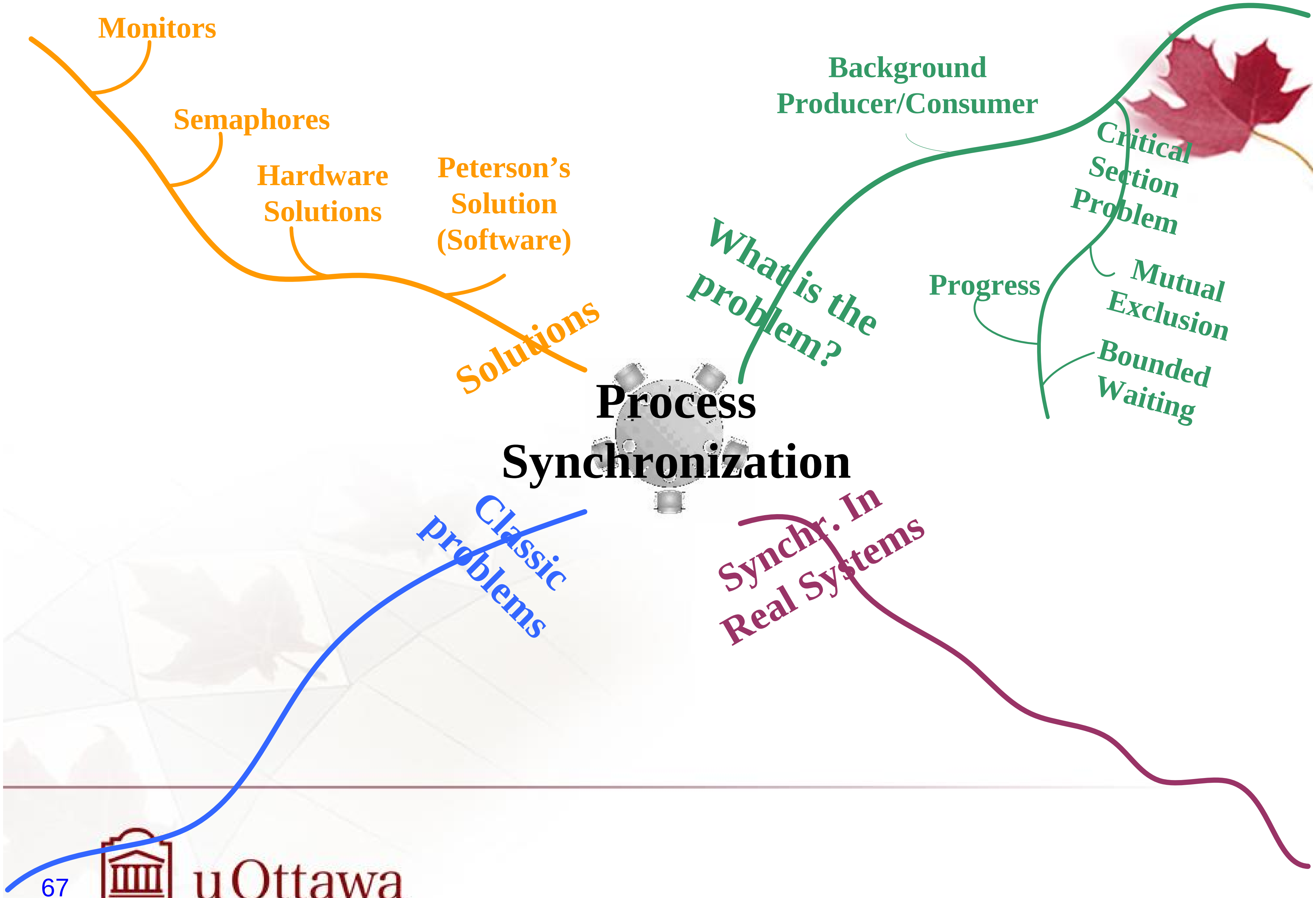


Advantages of semaphores

- Single variable (data structure) per critical section
- Two operations: wait, signal
- Can be applied to more than 2 processes
- Can have more than a single process enter the CS
- Can be used for general synchronization
- Service offered by OS, including blocking and queue management

Problems with semaphores

- programming difficulty
  - Wait and signal are scattered across different programs and running threads/processes
  - Not always easy to understand the logic
  - Usage must be correct in all threads/processes
  - One « bad » thread/process can make a collection of threads/processes fail (e.g. forget a signal)



Process Synchronization
Monitors
Semaphores
Hardware Solutions
Peterson's Solution (Software)
Solutions
What is the problem?
Background Producer/Consumer
Critical Section Problem
Progress
Mutual Exclusion
Bounded Waiting
Classic problems
Synchr. In Real Systems

Monitors

- Are high-level language constructs that provide equivalent functionality to that of semaphores but are easier to control
- Found in many concurrent programming languages
  - Concurrent Pascal, Modula-3, C#, Java...
  - Can be implemented with semaphores...
  - See section 6.7.3 of the textbook

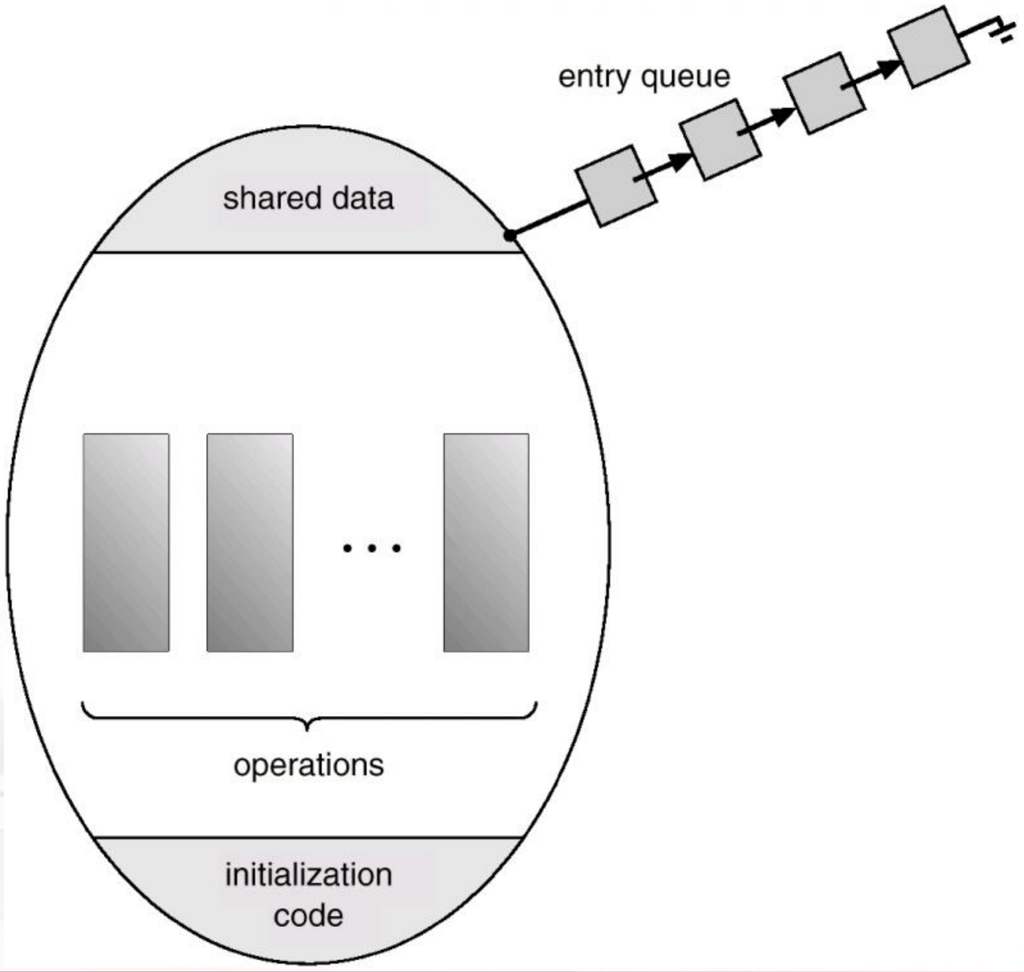
Monitors

- Monitor is a software module Abstract Data Type(ADT) containing:
  - one or more procedures
  - an initialization sequence
  - local data variables
- Monitors Characteristics:
  - local variables accessible only by monitor’s procedures
  - a process enters the monitor by invoking one of it’s procedures
  - only one process can be executing in the monitor at any one time (but a number of threads/processes can be waiting in the monitor).

Monitors

- The monitor ensures mutual exclusion: no need to program this constraint explicitly
- Shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry
- Process synchronization is done by the programmer by using condition variables that represent conditions a process may need to wait for before executing in the monitor

General Structure of the Monitor

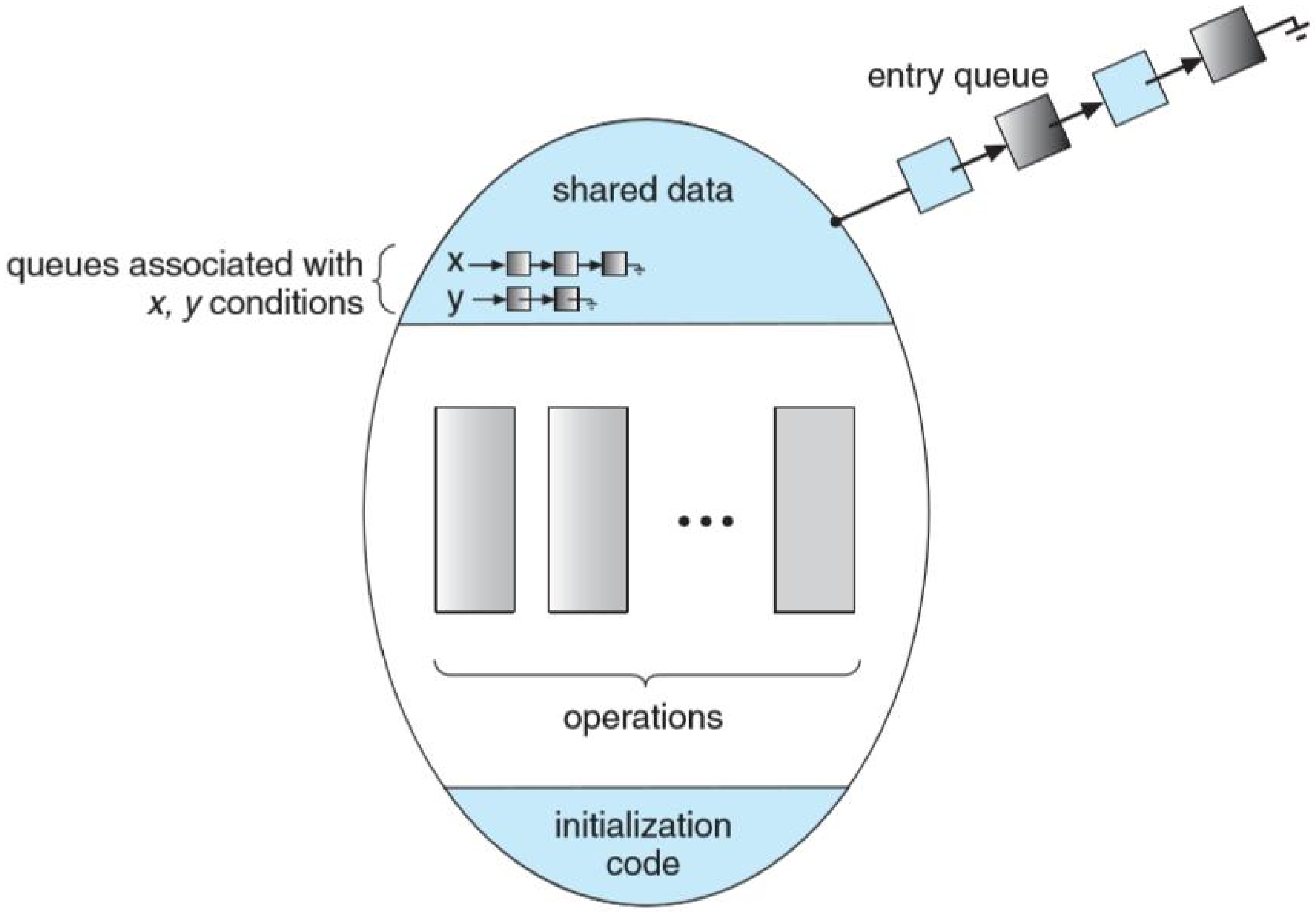


Condition variables

- Condition Variables are local to the monitor (accessible only within the monitor)
- Condition Variables support two functions:
  - wait() blocks execution of the calling process/thread on condition (variable)
  - signal() resume execution of some process/thread blocked on condition variable
    - ✓ If several such process exists: choose one based on queuing scheme (e.g. FCFS, priority - see section 6.7.4 of the textbook)
    - ✓ If no such process exists: do nothing

uOttawa

Monitor with Conditional Variables



Dinning Philosophers: Monitor

- Define the monitor « dp » (dining philosophers)
- Shared variables: each philosopher has its own state that can be: (thinking, hungry, eating)
  - philosopher i can only have state[i] = eating iif its neighbors are not eating
- Conditional variables: each philosopher has a condition self
- the philosopher i can wait on self [i] if it wants to eat but cannot obtain 2 chopsticks

Dinning Philosophers: Monitor

- Four functions in the monitor
  - pickup(i) – philo. i wants to pick up chopsticks
  - putdown(i) – philo. i puts down his/her chopsticks
  - test(i) – check if philo. i can start to eat
  - Initialization_code() - initialization

Philosopher i executes the loop

```
while(true)
{
   think
   dp.pickup(i)
   eat
   dp.putdown(i)
}
```

Check Possibility to eat

```
void test(int i)
{
  if ( (state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) )
  {
    state[i] = EATING;
    self[i].signal();
  }
}
```

A philosopher eats if his/her neighbors do not eat and he/she is hungry.

When the philosopher allowed to eat, a signal allows the philosopher to stop waiting and start eating

No change in state occurs if tests fail.

uOttawa

Pickup Chopsticks

```
void pickUp(int i)
{
  state[i] = HUNGRY;
  test(i); // check if can eat
  if (state[i] != EATING)
    self[i].wait();
}
```

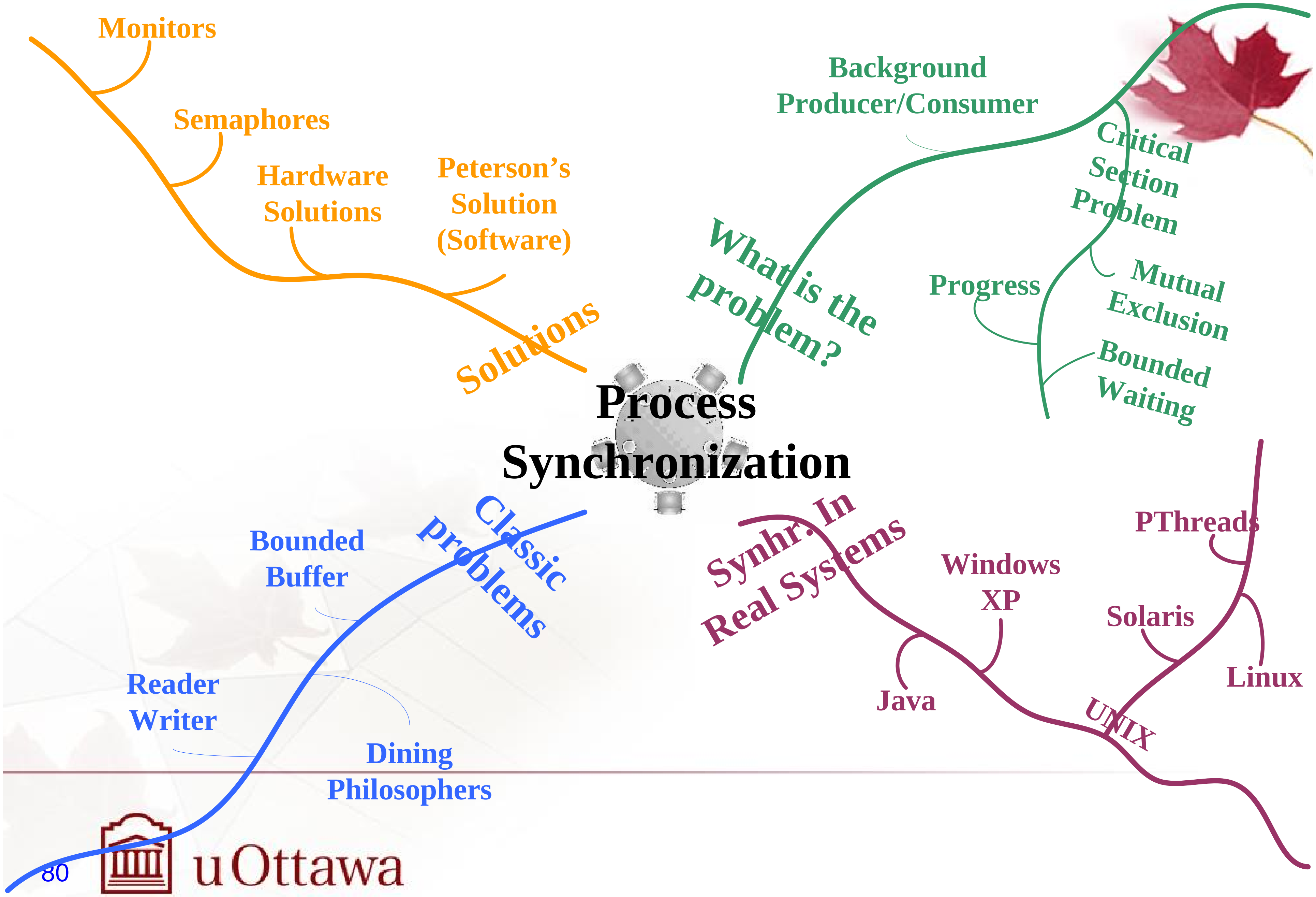
Try to start eating by doing a test. If comes out of test not eating, must wait – blocs until the execution of self[i] signal() in a future test

uOttawa

Put down Chopsticks

```
void putDown(int i)
{
  state[i] = THINKING;
 // test the two neighbors if they're waiting
  test((i + 4) % 5);
  test((i + 1) % 5);
}
```

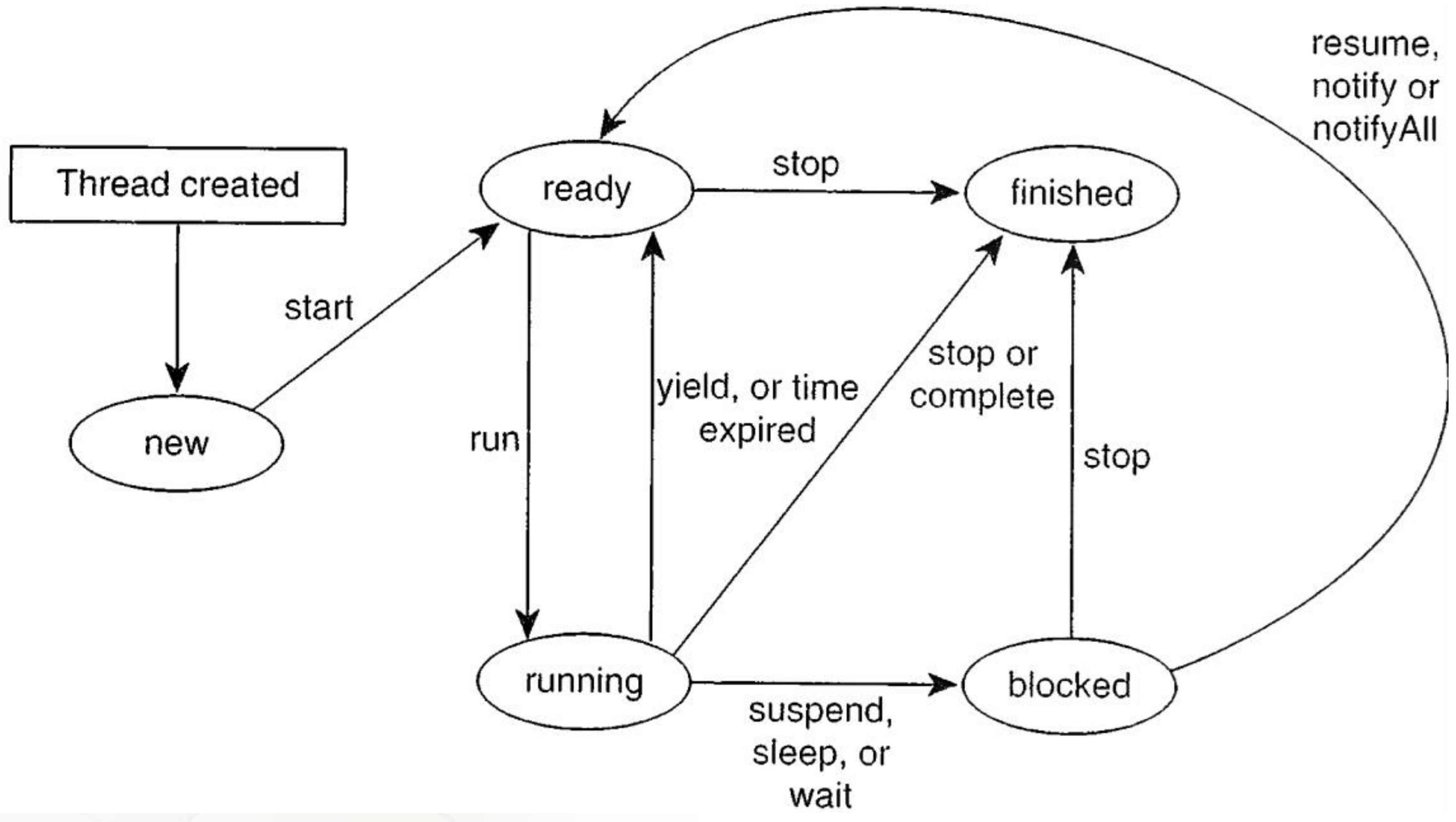
Once done eating, try to get his/her neighbors eating using test.



Process Synchronization
What is the problem?
Background
Producer/Consumer
Critical Section Problem
Mutual Exclusion
Progress
Bounded Waiting
Solutions
Monitors
Semaphores
Hardware Solutions
Peterson's Solution (Software)
Classic problems
Bounded Buffer
Reader Writer
Dining Philosophers
Synhr. In Real Systems
Java
Windows XP
UNIX
Solaris
Linux
PThreads

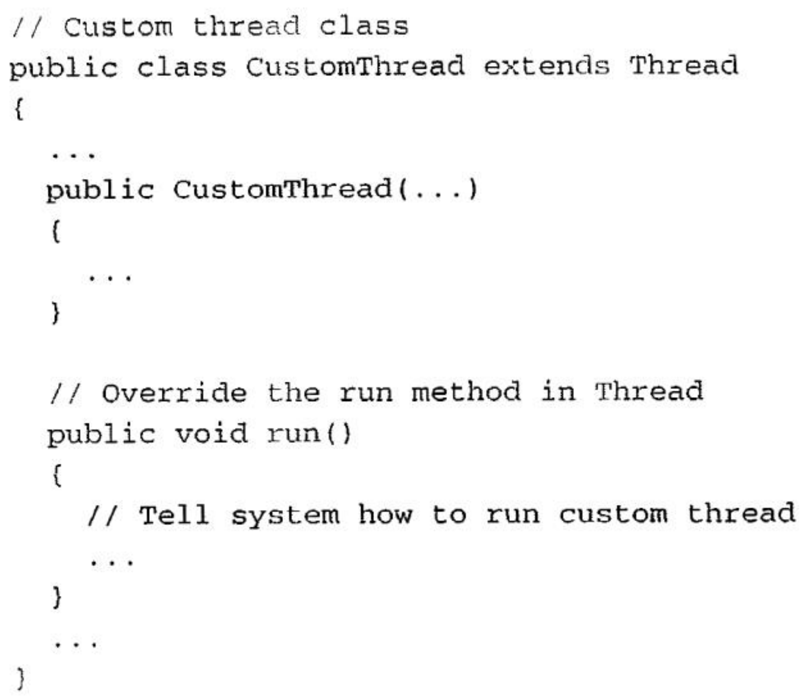
uOttawa

Java Thread States

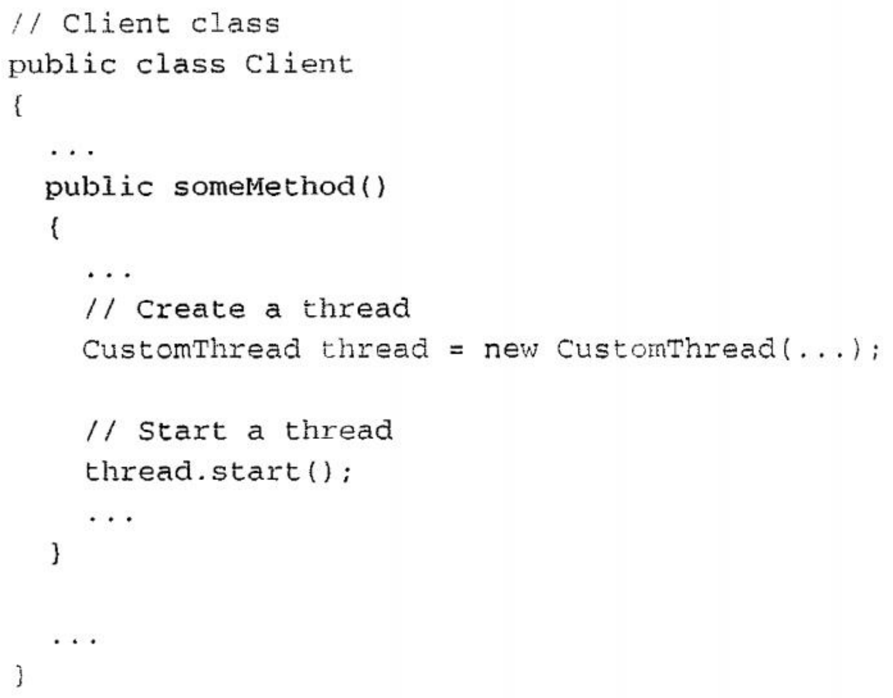


Creating a Thread

- By extending the Thread class



```
// Custom thread class
public class CustomThread extends Thread
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Override the run method in Thread
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

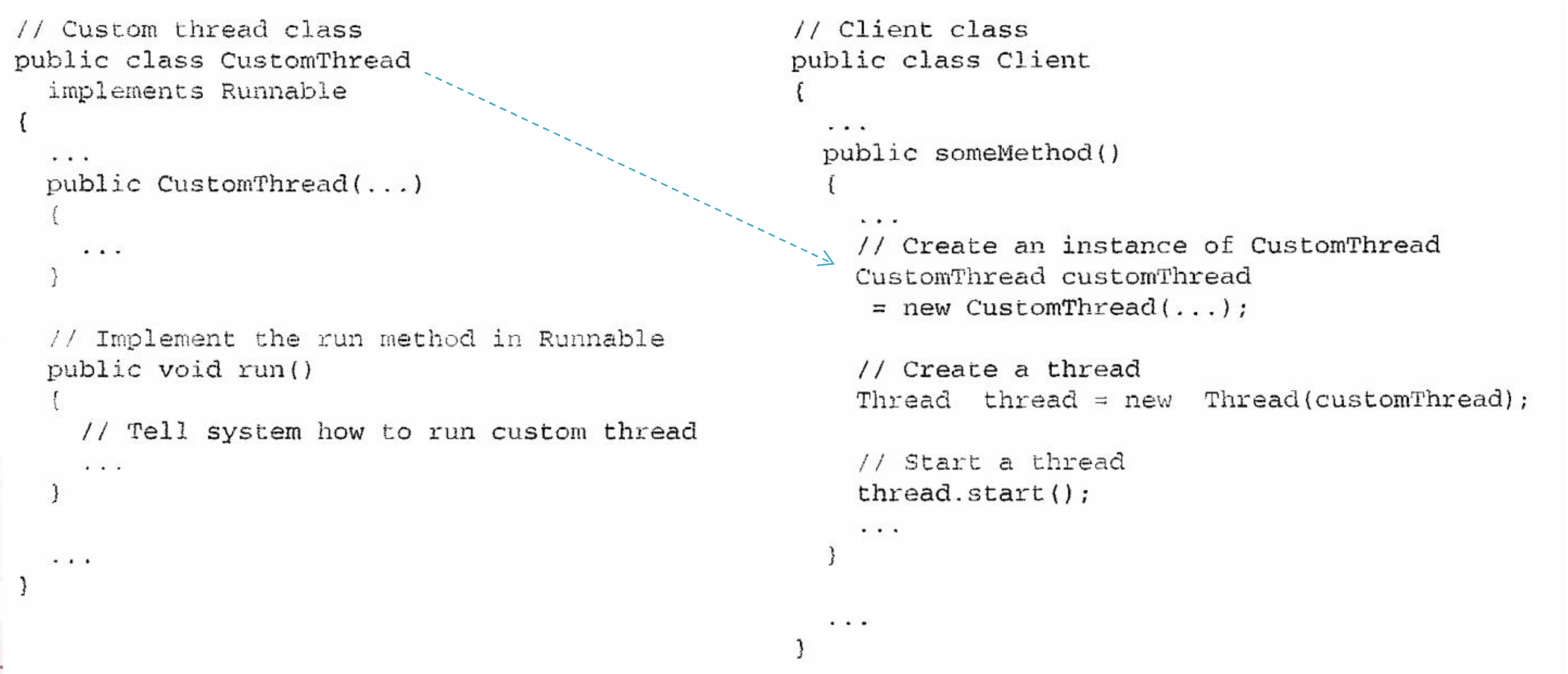


```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create a thread
    CustomThread thread = new CustomThread(...);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

uOttawa

Creating a Thread

- By implementing the Runnable interface



```
// Custom thread class
public class CustomThread
  implements Runnable
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Implement the run method in Runnable
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }

  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create an instance of CustomThread
    CustomThread customThread
     = new CustomThread(...);

    // Create a thread
    Thread  thread = new  Thread(customThread);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

uOttawa

Thread Sleep

- You can make a thread sleep (block) for a number of milliseconds
- Notice that these methods are static
- They thrown an InterruptedException

```
public static void sleep(long ms) throws InterruptedException
```

```
…
  try{
    sleep(1000); // sleep 1 sec
  } catch (InterruptedException e) {}
…
```

uOttawa

Thread Join

- If you create several threads you can wait for the threads to complete before putting together the results from these threads

```
public final void join() throws InterruptedException
public final void join(long millis) throws InterruptedException
public final void join(long millis, int nanos) throws InterruptedException
```

```
Thread t1 = new Thread(new Task1());
Thread t2 = new Thread(new Task2());
Thread t3 = new Thread(new Task3());

// this will call run() function
t1.start(); t2.start(); t3.1start();
/
/ waits for this thread to die
 t1.join();  t2.join();  t3.join();
```

Interrupting Threads

- In Java, you have no way to force a Thread to stop
- But you can interrupt a Thread with the `interrupt()` method (software interrupt)
  - If the thread is sleeping or joining another Thread, an `InterruptedException` is thrown and the interrupted status of the thread is cleared

```
public void interrupt()
```

Intrinsic Locks – synchronized methods

- Any piece of code that can be simultaneously modified by several threads must be made Thread Safe
- Consider the following simple piece of code:

```
public int getNextCount(){
  return ++counter;
}
```

- An increment like this, is not an atomic action, it involves:
  - Reading the current value of counter
  - Adding one to its current value
  - Storing the result back to memory

Intrinsic Locks – synchronized methods

- We must use a lock on access to the “value”
- You can add such lock to a method by simply using the keyword: synchronized

```
public synchronized int getNextCount(){
   return ++counter;
}
```

- This guarantees that only one thread executes the method
- If you have several methods with the synchronized keyword, only one method can be executed at a time
  - This is called an Intrinsic Lock

Intrinsic Locks – synchronized block

Each Java object has an intrinsic lock associated with it (monitor)

- we can use that lock to synch access to a method
- instead, we can elect to synch access to a block of code

```
public int getNextValue() {
        synchronized (this) {return value++;}
 }
```

- alternatively, use the lock of another object, which is useful since it allows you to use several locks for thread safety in a single class

```
public int getNextValue() {
        synchronized (lock) {return value++;}
}
```

Intrinsic Locks - synchronized static methods

- So we mentioned that each Java object has an intrinsic lock associated with it, e.g. static methods that are not associated with a particular object?
  - There is also an intrinsic lock associated with the class
  - Only used for synchronized class (static) methods

Java Monitors

- There are several mechanisms to create monitors
  - The simplest one, uses the knowledge that we have already gathered regarding the intrinsic locks
- The intrinsic locks can be effectively used for mutual exclusion (competition synchronization)
- We need a mechanism to implement cooperation synchronization
  - we need to allow threads to suspend themselves if a condition prevents their execution in a monitor
  - This is handled by the `wait()` and `notify()` methods

Wait Operations

`wait()`

Tells the calling thread to give up the monitor and wait until some other thread enters the same monitor and calls **`notify()or notifyAll()`**

`wait(long timeout)`

Causes the current thread to wait until another thread invokes the **`notify() or notifyAll()`** method, or the specified amount of time elapses

Notify Operations

`notify()`

Wakes up a single thread that is waiting on this object's monitor (intrinsic lock).

If more than a single thread is waiting, the choice is arbitrary (*is this fair?*)

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

`notifyAll()`

Wakes up all threads that are waiting on this object's monitor.

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

The next thread to lock this monitor is also randomly chosen

Wait / Notify Example

```
public class BufferMonitor{
    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    public synchronized void deposit (int item ) throws InterruptedException{
        while (buffer.length == filled){
            wait(); // blocks thread
        }

        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;

        notify(); // free a task that has been waiting on a condition
    }
```

Wait / Notify Example

```
    public synchronized int fetch() throws InterruptedException{
        while (filled == 0){
            wait(); // block thread
        }

        int item = buffer[next_out];
        next_out = (next_out + 1) % buffer.length;
        filled--;

        notify(); // free a task that has been waiting on a condition

        return item;
    }
}
```

Java Lock interface

- You can create a monitor using the Java Lock interface
  - **`ReentrantLock`** is the most popular implementation of **`Lock`**
- `ReentrantLock` defines two constructors:
  - Default constructor
  - Constructor that takes a Boolean (is_fair)
- Fairness is a slightly heavy (in terms of processing), and therefore should be used only when needed
- To acquire the lock, you just have to use the method **`lock`**, and to release it, call **`unlock`**

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

Lock Example

```
public class SimpleMonitor {
   private final Lock lock = new ReentrantLock();
   public void testA() {
      lock.lock();
      try {//Some code}
      finally {lock.unlock();}
   }
   public int testB() {
      lock.lock();
      try {return 1;}
      finally {lock.unlock();}
   }
}
```

Adding a condition to a lock

- Without being able to wait on a condition, monitors are useless...
  - Cooperation is not possible
- There is a specific class that has been developed just to this end: `Condition` class
  - You create a **`Condition`** instance using the **`newCondition()`** method defined in the **`Lock`** interface

Adding a condition to a lock

```
public class BufferMonitor {

    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    private final Lock lock = new ReentrantLock(true);
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
```

Adding a condition to a lock

```
public void deposit (int item ) throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (buffer.length == filled){
            notFull.await(); // blocks thread (wait on condition)
        }
        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;
        notEmpty.signal(); // signal thread waiting on the empty condition
    }
    finally{
        lock.unlock();// Whenever you lock, you must unlock
    }
}
```

uOttawa

Lock vs Synchronized

- Monitors implemented with `Lock` and `Condition` classes have some advantages over the intrinsic lock based implementation:
  1. Ability to have more than one condition variable per monitor (see previous example)
  2. Ability to make the lock fair (remember, synchronized blocks or methods do not guarantee fairness)
  3. Ability to check if the lock is currently being held (by calling the **`isLocked()`** method)
     - ✓ Alternatively, you can call **`tryLock()`** which acquires the lock only if it is not held by another thread
  4. Ability to get the list of threads waiting on the lock (by calling the method **`getQueuedThreads()`**)

Lock vs Synchronized

- Disadvantages of `Lock` and `Condition` :
  1. Need to add lock acquisition and release code
  2. Need to add try-finally block

uOttawa

Java Semaphores

- Java defines a semaphore class: `java.util.concurrent.Semaphore`
- Creating a counting semaphore:

```
Semaphore available = new Semaphore(100);
```

- Creating a binary semaphore:

```
Semaphore available = new Semaphore(1);
```

Semaphore Example

```
public class Example {
    private int counter= 0;

    private final Semaphore mutex = new Semaphore(1)

    public int getNextCount() throws InterruptedException {
        mutex.acquire();
        try {
            return ++counter;
        } finally {
            mutex.release();
        }
    }
}
```