LECTURE 14

INTRODUCTION TO CONCURRENCY

SUBJECTS

Concepts of concurrency

Subprogram-level concurrency

Semaphores

Deadlocks

CONCEPT OF CONCURRENCY

Concurrency can be achieved at different levels:

- Instruction level
- Unit level
- Program level

Concurrent execution of program units can be performed:

- Physically on separate processors,
- or Logically in some time-sliced fashion on a single processor computer system

WHY STUDY CONCURRENCY?

uOttawa
L'Université canadienne
Canada's university

Computing systems solve real world problems, where things happen at the same time:

- Railway Networks (lots of trains running, but that have to synchronize on shared sections of track)
- Operating System, with lots of processes running concurrently
- Web servers

Multiple processor or multi-core computers are now being widely used

- This creates the need for software to make effective use of that hardware capability

SUBPROGRAM-LEVEL CONCURRENCY

A task is a unit of program that can be executed concurrently with other units of the same program

- Each task in a program can provide one thread of control

A task can communicate with other tasks through:

- Shared nonlocal variables
- Message passing
- Parameters

SYNCHRONIZATION

A mechanism to control the order in which tasks execute

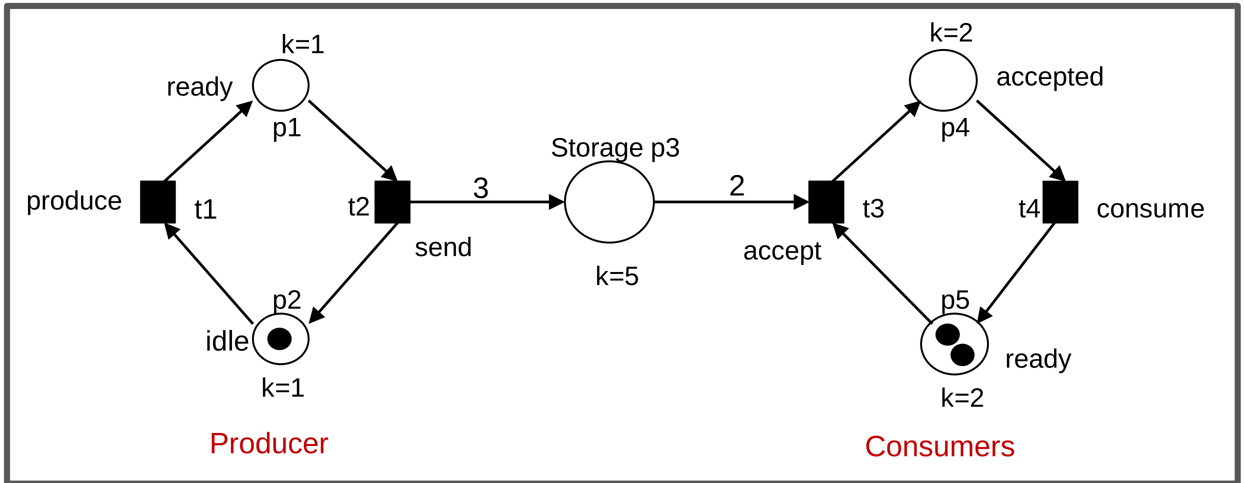
Cooperation synchronization is required between task A and task B when:

- **Task A** must wait for **task B** to complete some specific activity before **task A** can continue execution
- **Recall** the producer-consumer petri net problem

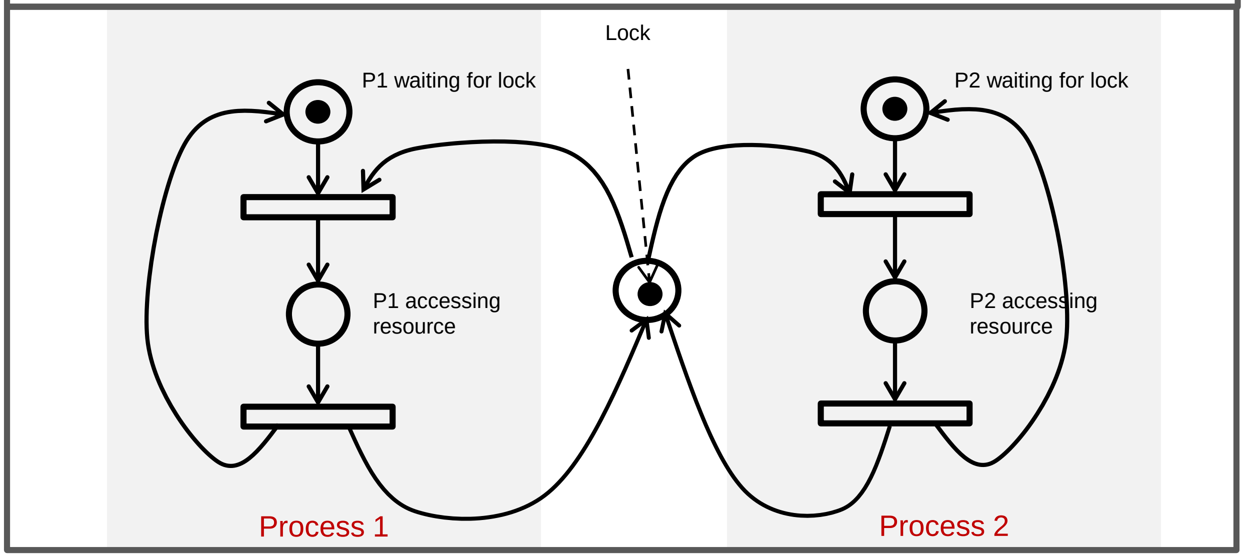
Competition synchronization is required between two tasks when:

- Both require the use of some resource that cannot be simultaneously used

SYNCHRONIZATION (PETRI NETS)

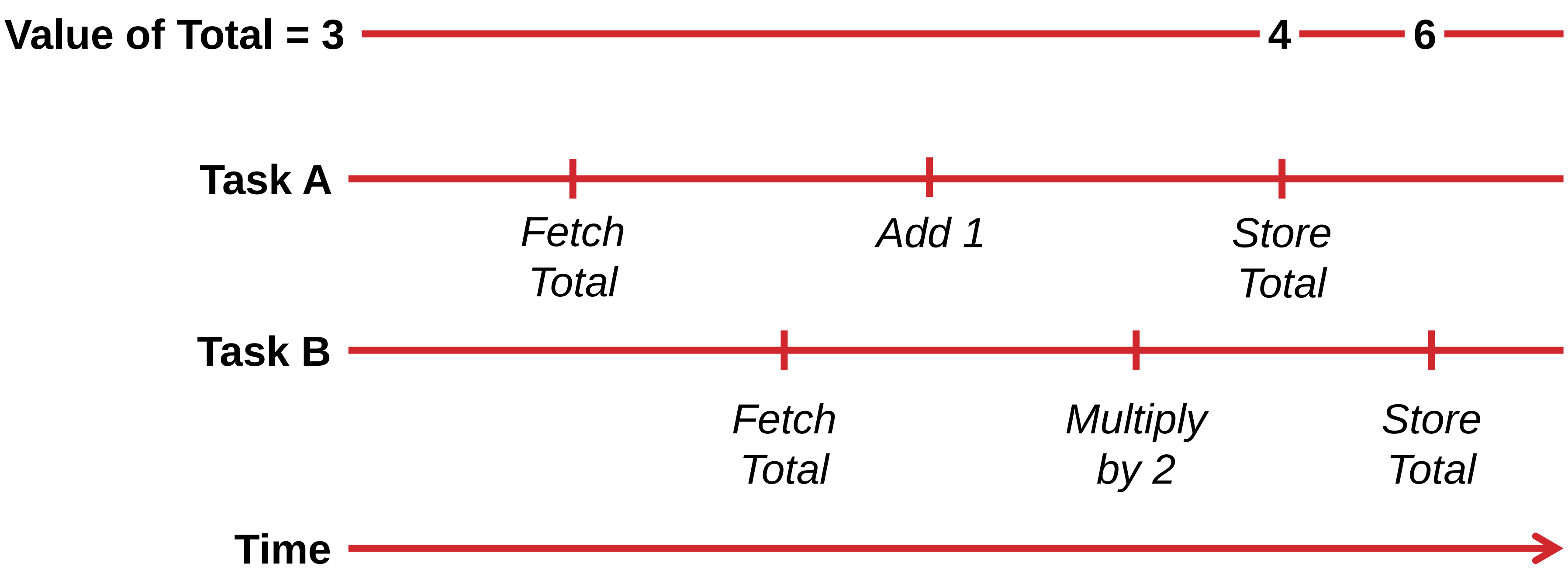


Cooperation Synchronization



Competition Synchronization

THE NEED FOR COMPETITION SYNCHRONIZATION



CRITICAL SECTION

A segment of code, in which the thread may be:

- Changing common variables,
- Updating a table,
- Writing to a file,
- Or updating any shared resource

The execution of critical sections by the threads is mutually exclusive in time

TASK (THREAD) STATES

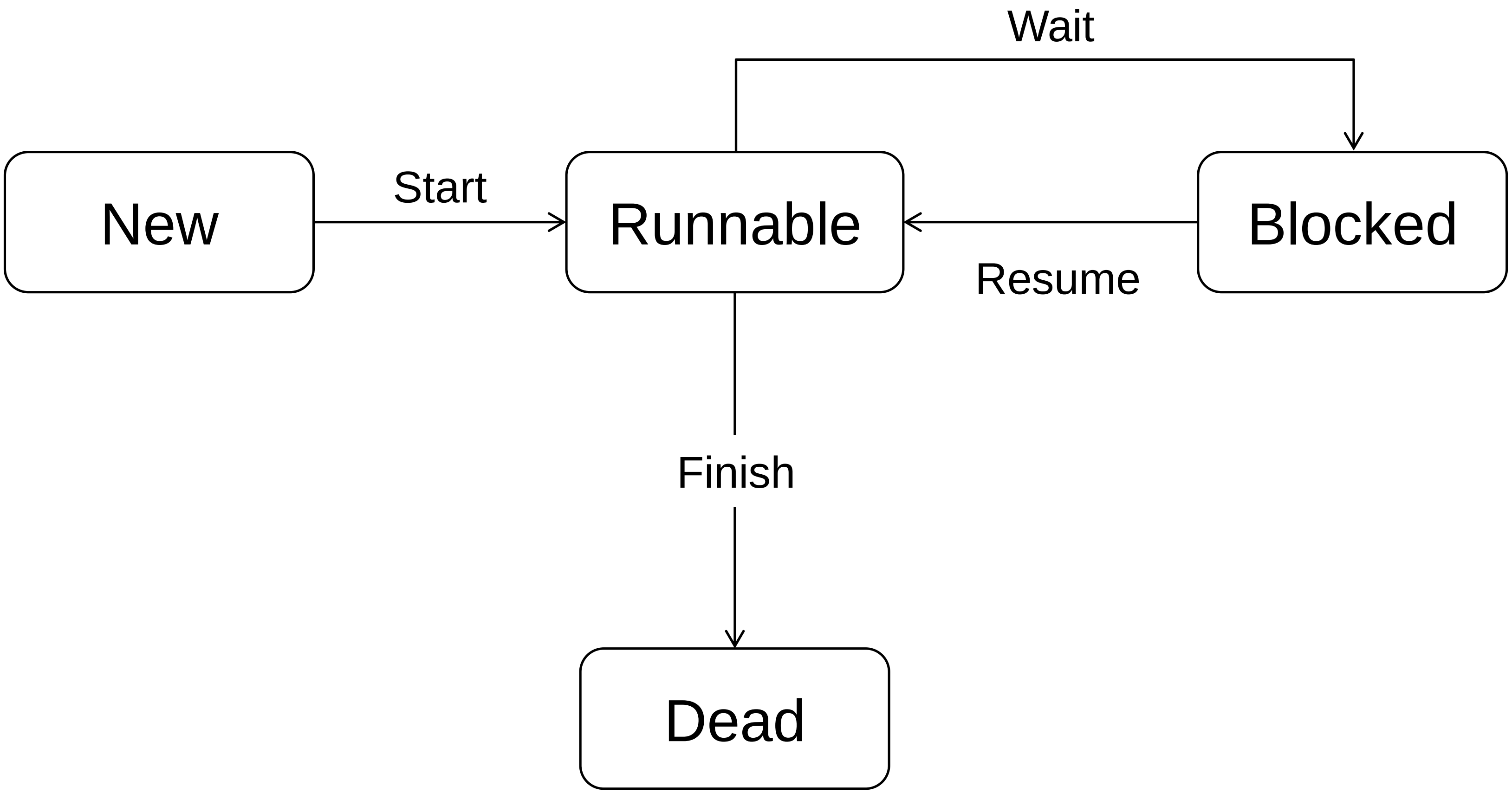
New: it has been created, but has not yet begun its execution

Runnable or **ready**: it is currently running or is ready to run

Blocked: it has been running, but its execution was interrupted by one of several different events

Dead: no longer active in any sense

TASK (THREAD) STATES



SEMAPHORES

Semaphore is a technique used to control access to a common resource for multiple tasks

- It is an object consisting of an integer (*counter*) and a queue that stores task descriptors

A task that requires access to a critical section needs to "acquire" the semaphore

Classically: a system of sending messages by holding the arms of two flags in certain positions according to an alphabetic code

SEMAPHORES

Two operations are always associated with Semaphores

Operation "P" is used to get the semaphore, and "V" to release it

- P (proberen, meaning to test and decrement the integer)
- V (verhogen, meaning to increment) operation

Alternatively, these operations are called: wait and release

SEMAPHORES

Operating systems often distinguish between counting and binary semaphores

The value of the counter of a counting semaphore can range over an unrestricted domain.

The value the counter of a binary semaphore can range only between 0 and 1

BINARY SEMAPHORES

The general strategy for using a binary semaphore to control access to a critical section is as follows:

```
Semaphore aSemaphore;

wait(aSemaphore);
Critical section();
release(aSemaphore);
```

BINARY SEMAPHORE - WAIT

```
wait(binarySemaphore):



  if binarySemaphore’s counter == 1 then
    set binarySemaphore’s counter = 0
  else
    Set the task’s state to blocked
    Put the caller in binarySemaphore’s queue
  end
```

BINARY SEMAPHORE - RELEASE

```
release(binarySemaphore):

    if binarySemaphore's queue is empty then
      set binarySemaphore's counter = 1
    else
      set the state of the task at the queue head to runnable
      Perform a dequeue operation
    end
```

COUNTING SEMAPHORE - WAIT

```
wait(countingSemaphore):



  if countingSemaphore's counter > 0 then
    Decrement countingSemaphore's counter
  else
    Set the task's state to blocked
    Put the caller in countingSemaphore's queue
  end
```

COUNTING SEMAPHORE - RELEASE

```
release(countingSemaphore):

  if countingSemaphore's queue is empty then
    increment countingSemaphore's counter
  else
    set the state of the task at the queue head to ready
    Perform a dequeue operation
  end
```

PRODUCER AND CONSUMER

uOttawa
L'Université canadienne
Canada's university

```
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;
```

```
wait(countingSemaphore):

 if countingSemaphore's counter > 0 then
  Decrement countingSemaphore's counter
 else
  Set the task's state to blocked
  Put caller in countingSemaphore's queue
 end
```

```
release(countingSemaphore):

 if countingSemaphore's queue is empty then
  increment countingSemaphore's counter
 else
  set state of task at queue head to ready
  Perform a dequeue operation
end
```

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;
```

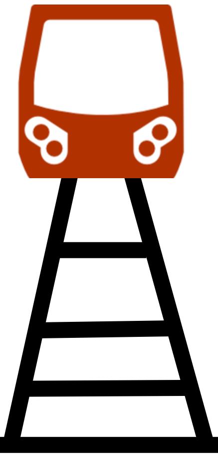
COMPETITION SYNCHRONIZATION (II)

```
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;
```

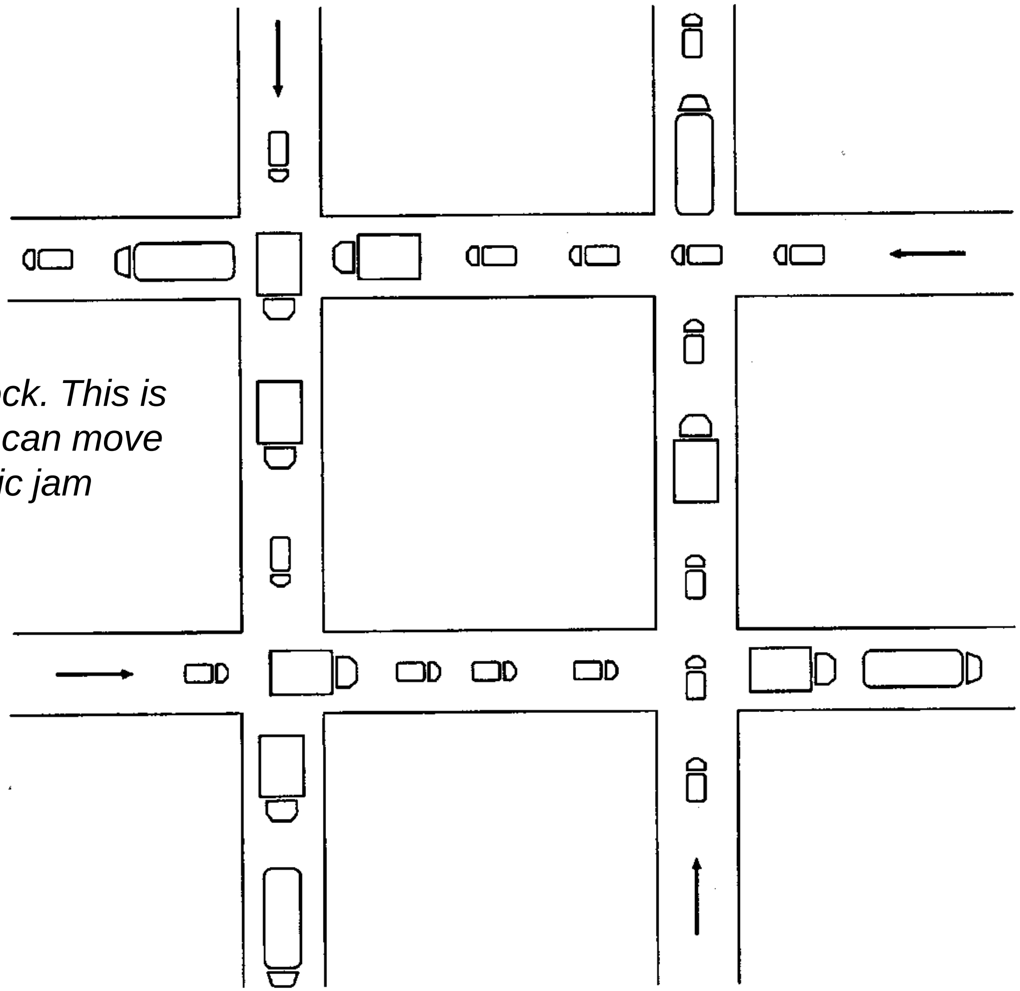
DEADLOCKS

A law passed by the Kansas legislature early in 20th century:

"…… When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. ……"



DEADLOCK



Classic case of traffic deadlock. This is a gridlock where not vehicle can move forward to clear the traffic jam

CONDITIONS FOR DEADLOCK

Mutual exclusion: the act of allowing only one process to have access to a dedicated resource

Hold-and-wait: there must be a process holding at least one resource and waiting to acquire additional ones that are currently being held by other processes

No preemption: the lack of temporary reallocation of resources. Ressources can be released only voluntarily

Circular waiting: each process involved in the impasse is waiting for another to voluntarily release the resource

DEADLOCK EXAMPLE

```
BinarySemaphore s1, s2;
```

```
Task A:
   wait(s1)
      // access resource 1
      wait (s2)
         // access resource 2
      release(s2)
   release(s1)
end task
```

```
Task B:
   wait(s2)
      // access resource 2
      wait(s1)
         // access resource 1
      release(s1)
   release(s2)
end task
```

STRATEGY FOR HANDLING DEADLOCKS

Prevention: eliminate one of the necessary conditions

Avoidance: avoid if the system knows ahead of time the sequence of resource requests associated with each active processes

Detection: detect by building directed resource graphs and looking for circles

Recovery: once detected, it must be untangled and the system returned to normal as quickly as possible

- Process termination
- Resource preemption

THANK YOU!

QUESTIONS?