COMP750

Concurrent Programming

Patterns

Concurrent Programming

- Programming generally involves applying the appropriate known algorithms to your problem.
- There are just a few standard constructs used in concurrent programming.
- Once you know the standard constructs, you can apply them to solve almost all concurrent programs.

Concurrency Patterns

- **Mutual Exclusion** – Used to ensure that only one task executes a particular segment of code. Can be used for resource allocation.
- **Synchronization** – Coordinates the action of different tasks.
- **Barrier Synchronization** – Waits until all tasks are complete.

Mutual Exclusion with semaphores

```
semaphore s = 1;

p(s);
  // Critical section;
v(s);
```

Windows Mutual Exclusion

```
#include <process.h>

HANDLE          mutex;
mutex = CreateMutex(NULL, false, NULL);

WaitForSingleObject(mutex, INFINITE);
        // Critical Section
ReleaseMutex(mutex);
```

Mutual Exclusion with synchronized methods

```
public class Example {
  public synchronized object getThing() {
     Critical section; // nothing to it
  }
}
```

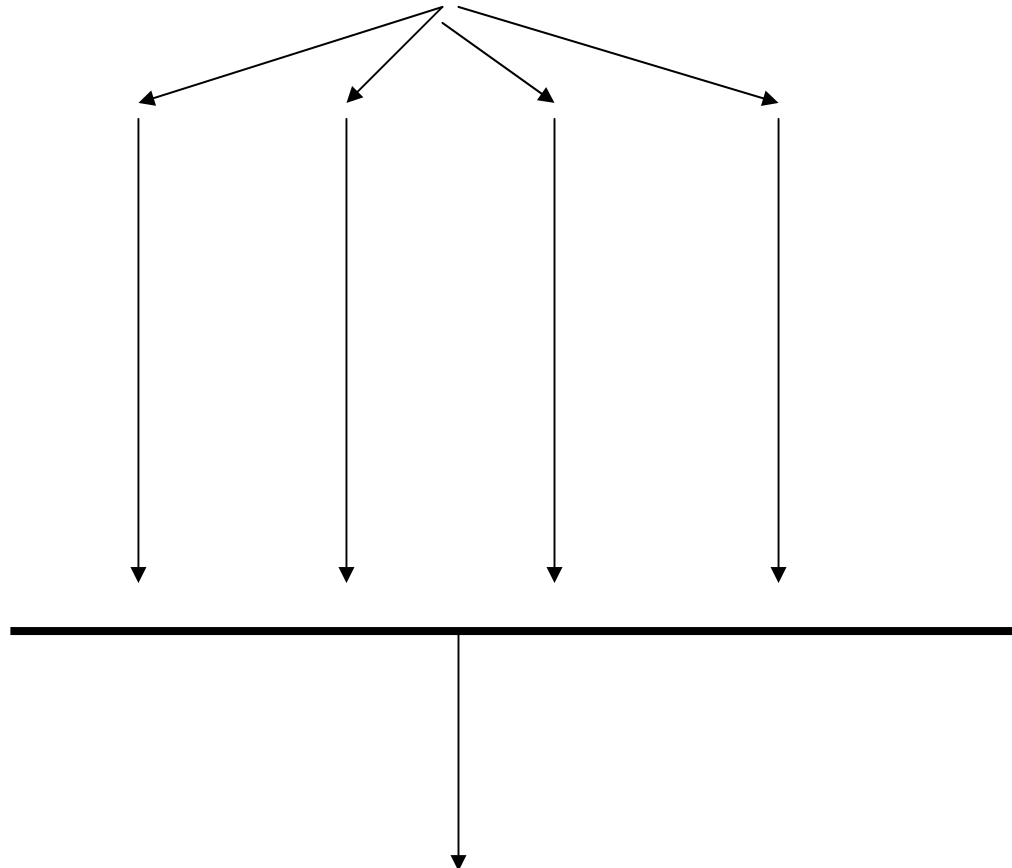
Synchronization with semaphores

```
semaphore numReady = 0;
void makeIt() {
  v(numReady);
}
void useIt() {
  p(numReady);
}
```

Synchronization with Synchronized Methods

```
public class numReady {
int available = 0;
public synchronized void makeIt(){
  available++;
  signal();
}
public synchronized void useIt() {
  if (available == 0) wait();
  available--;
}
```

Barrier Synchronization



One task continues after all tasks complete.

Barrier Synchronization with Semaphores

```
semaphore mutex = 1;
int numRunning = N;
// start threads and run the parallel stuff here
p(mutex);
numRunning--;
If (numRunning != 1) {
  v(mutex);
  exit(); // all but last task terminates
}
v(mutex);
```

Barrier Synchronization with Synchronized Methods

```
int numRunning = N;
public void run() {
  // run the parallel stuff here
  synchronized (this) {
     numRunning—;
     if(numRunning==1)morestuff();
  }
}
```

Barrier Synchronization with Join

```
SubThread[] t = SubThread[N];
// Start the parallel threads
for (i = 0; i < N; i++) {
  t[i] = new SubThread();
  t.start();
}
// Wait for thread to terminate
for (i = 0; i < N; i++)
    t[i].join();
```

.NET Barrier Synchronization

```
HANDLE  T[N];
// Start parallel threads
for (i = 0; i < N; i++ ) {
  T[i] =
  CreateThread(NULL,0,func,&parm,0,NULL);
}
// Wait for all threads to complete
WaitForMultipleObjects(N, T, true, INFINITE );
```

A good program *(and one that gets a good grade)*

- uses multiple processes or threads.
- produces the correct result.
- will not deadlock under any situation.
- has maximum concurrency.
- utilizes an efficient algorithm.
- does not use busy waiting.
- To aid in understanding the operation of the algorithm, the program should print useful information whenever an important event occurs.

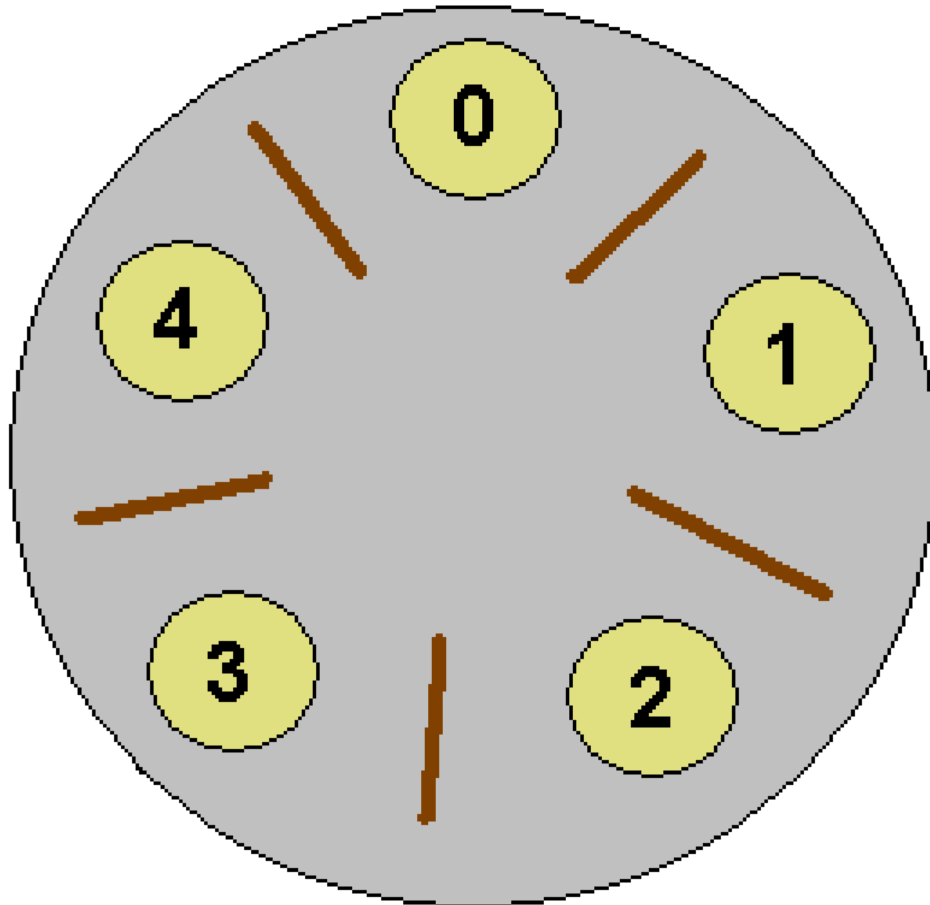
Dining Philosophers Problem

The philosophers spend their time thinking and eating on independent schedules. To eat, each philosopher needs both the chopstick to her right and the chopstick to her left.

Note that adjacent philosophers cannot eat at the same time because they both need the same chopstick.

Dining Philosophers

There are 5 philosophers who sit around a table. Between each philosopher is a chopstick.



Dining Philosophers Solution

- This problem has 5 resources (the chopsticks) that require exclusive use.
- A semaphore can be assigned to each chopstick to provide exclusive access.

Dining Philosophers Solution

Each chopstick resource is protected by a semaphore initialized to 1.

```
philosopher i
do forever {
  think;
  p(chopstick(i));              // right
  p(chopstick((i+1)%5));        // left
  eat;
  v(chopstick(i));
  v(chopstick((i+1)%5));
}
```

Potential Problem

- If all five philosophers get hungry at the same time and each grabs the chopstick to their right, none of them will have another chopstick to use and they will all starve.
- A possible solution is to have the odd numbered philosophers grab the right and then the left while even numbered philosophers grab the chopsticks in the reverse order.