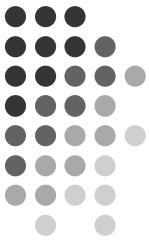
Concurrency 8C

Multitasking &
Operating Systems

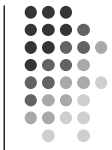


Operating Systems & Multitasking

- **Multitasking** - The coordination of several computational processes on one processor.
- An **operating system** (OS) is the system software responsible for the direct control and management of hardware and basic system operations.
- An OS provides a foundation upon which to run application software such as word processing programs, web browsers, etc.

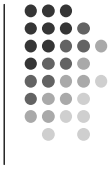
from Wikipedia.org

Operating System "Flavors"



- UNIX
  - System V, BSD, Linux
  - Proprietary: Solaris, Mac OS X
- Windows
  - Windows NT
  - Windows 2000
  - Windows XP
  - Windows Vista (2007)

Multitasking

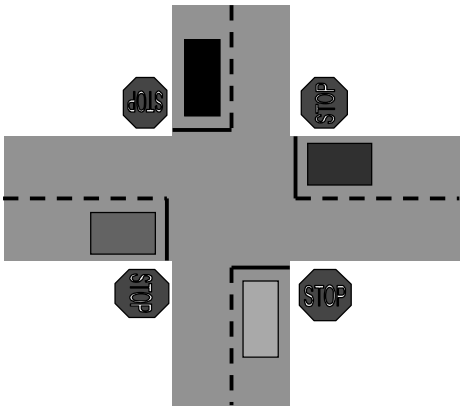


- Each (un-blocked) application runs for a very short time on the computer's processor and then another process runs, then another...
- This is done at such a fast rate that it appears to the computer user that all applications are running at the same time.
  - How do actors appear to move in a motion picture?

Critical Section

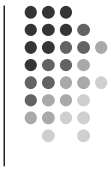
- A **critical section** is a section of computer code that must only be executed by one process or thread at a time.
- Examples:
  - A printer queue receiving a file to be printed
  - Code to set a seat as reserved
  - Web server that generates and returns a web page showing current registration in course

A Critical Section



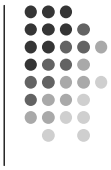
Cars may not make turns through this intersection. They may only proceed straight ahead. When a car stops at this intersection, it can proceed straight ahead if there is no car to its left and no car to its right. Otherwise it must wait some random amount of seconds and then check again to see if it's safe to proceed. If not, it waits again, and so on.

Shared Resources



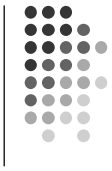
- memory
- tape drives
- disk drives
- printers
- communication ports
- input devices (keyboard, mouse)

Deadlock



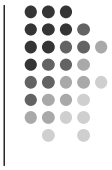
- Deadlock is the condition when two or more processes are all waiting for some shared resource that other processes of the group hold, causing all processes to wait forever without proceeding.
- How can deadlock occur at the intersection with the 4-way stop?

Dining Philosophers Problem



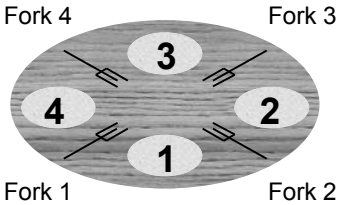
- Each philosopher thinks for a while, then picks up his left fork, then picks up his right fork, then eats, then puts down his left fork, then puts down his right fork, thinks for a while...
  - We assume here that each philosopher thinks and eats for random times, and a philosopher cannot be interrupted while he picks up or puts down a single fork.
- Each fork models a "resource" on a computer controlled by an OS.
- Original problem proposed by Dijkstra

Dining Philosophers Problem



- There are N philosophers.
- Philosopher i does the following:
  1. THINK
  2. Pick up fork i.
  3. Pick up fork (i modulo N)+1.
  4. EAT
  5. Put down fork i.
  6. Put down fork (i modulo N)+1.
  7. Go to step 1.

N=4



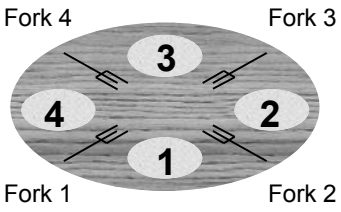
NOTE: (i modulo N) + 1 = i + 1, if 1 ≤ i < N
(i modulo N) + 1 = 1, if i = N

Dining Philosophers Problem

- How can deadlock occur?

1. THINK
2. Pick up fork i.
3. Pick up fork (i modulo N)+1.
4. EAT
5. Put down fork i.
6. Put down fork (i modulo N)+1.
7. Go to step 1.

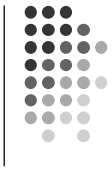
N=4



Semaphores

- A (binary) semaphore S is a shared variable that holds an integer that is initialized to 1.
- Two "atomic" operations on semaphore S:
  - Request S
    - while S is not equal to 1, wait some random time
    - subtract 1 from S
  - Release S
    - add 1 to S

Using Semaphores

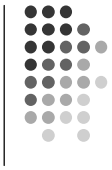


- We can use a semaphore to protect a critical section so that only one process accesses this section at a given time.
- Technique:

  non-critical program code
  Request S
  CRITICAL SECTION
  Release S
  non-critical program code

Initially, semaphore S is 1. First process to request S sets S to 0, blocking other processes from proceeding once they get to their request operations. Once the first process finishes running the critical section, it releases S, causing S to increase to 1, allowing another process to proceed into the critical section.

Solution for the Philosophers?

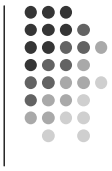


- F[i] = semaphore for fork i
- Philosopher i performs the following:

1. THINK
2. request F[i]
3. request F[(i mod N) + 1]
4. EAT
5. release F[i]
6. release F[(i mod N) + 1]
7. goto step 1

NO!
Deadlock can still occur!
HOW?

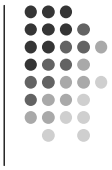
Solution for the Philosophers?



- Let F[i] = semaphore for fork i
- Philosopher i performs the following:
  1. THINK
  2. If i is not equal to N: request F[i], and then request F[i+1] otherwise: request F[1], and then request F[N]
  3. EAT
  4. If i is not equal to N: release F[i], and then release F[i+1] otherwise: release F[1], and then release F[N]
  5. Goto step 1

YES! How does this prevent deadlock?

Summary



- Distributed computing and multitasking are very challenging problems in computer science.
- The simple rules of sequential computation no longer apply when we compute
  - in a distributed manner across networks
  - in a single computer system in a multitasking environment