Processes

Dmitri Loguinov
Texas A&M University

January 26, 2024
Homework #1

• When running A*
  - Incorrect # of nodes if weight is integer in $q = L + \frac{w}{d+1}$
• Basic BFS and DFS
  - Order of traversal on this graph?

```
Adjacency list
A: E, D, B
B: A, G
C: E, D, F
D: A, C
E: A, C
F: C, G
G: B, F
```
Homework #1

- Refresh the concept of search
  - Assume an undirected graph $G = (V,E)$
  - Start node $s \in V$
- Maintain two structures
  - Unexplored set $U$
  - Discovered set $D$
- Approach #1:

```java
U.add(s)
while ( U.notEmpty() )
    x = U.removeNextNode() // node to explore
    if ( D.find(x) == true ) // if already explored, ignore
        continue
    N = G.getNeighbors(x) // N is a set of nodes
    if ( N.size() == 0 ) break // exit?
    for each y in N
        U.add(y)
```

Any problems?
Homework #1

• This code fails to actually insert anything into D

• Correct version:

```python
U.add(s)
while ( U.notEmpty () )
    x = U.removeNextNode ()
    if ( D.find(x) == true ) // if already explored, ignore
        continue
    D.add (x)
    N = G.getNeighbors (x)
    if ( N.size() == 0 ) break // exit?
    for each y in N
        U.add (y)
```

Any drawbacks?

• Requires huge storage as each node may be pushed into U as many times as there are links to it
  - Not advisable in practice
Homework #1

- Approach #2 inserts a single copy of each node in U:

```java
U.add(s); D.add(s); // s = source node
while ( UnotEmpty() )
    x = U.removeNextNode()
    N = G.getNeighbors(x)
    if ( N.size() == 0 ) break // exit?
    for each y in N
        if ( D.find(y) == false ) // has been pushed in U?
            U.add(y)
            D.add(y)
```

Always use this version!

- For most types of non-trivial exploration, approach #2 is far superior to #1
- What if D has a function that combines find/add?
  - Can directly use STL set’s insert() function
Homework #1

• When you find the exit, how far is it from s?
• Idea: make U keep track of tuples (nodeID, distance)

```c
U.add(s, 0); D.add(s);
while (U.notEmpty())
    t = U.removeNextTuple(); // t is a tuple
    N = G.getNeighbors(t.ID)
    if (N.size() == 0)
        printf("Found at distance %d\n", t.distance);
        break;
    for each y in N
        if (D.find(y) == false) // new node?
            U.add(y, t.distance + 1);
            D.add(y);
```

• Note that U.add() also needs light intensity for bFS/A*
  - See the handout for details
Homework #1

- Reusing the search algorithm
  - Create a base class

```cpp
class Ubase {
    virtual void Add (uint64 ID, int distance, float intensity) = 0;
    virtual UnexploredRoom RemoveNextTuple (void) = 0;
    ...
};
```

- Inherit four classes

```cpp
class Ubreadth : public Ubase {
    // implement a queue here
};
class Udepth : public Ubase {
    // implement a stack here
};
```

- Create base pointer to a specific class, then send it to search()

```cpp
Ubase *ptr;
if (searchType == BFS)
    ptr = new Ubreadth;
else if ...
    Search (ptr);

Search (Ubase *U)
{
    while (U->size() > 0)
        ...
}
```
Chapter 3: Roadmap

3.1 What is a process?
3.2 Process states
3.3 Process description
3.4 Process control
3.5 Execution of the OS
3.6 Security issues
3.7 Unix process management

Part II

Chapter 3: Processes
Chapter 4: Threads
Chapter 5: Concurrency
Chapter 6: Deadlocks
From the 1960s, jobs were described by a special data structure that allowed the OS to systematically monitor, control, and synchronize them.

This became known as a process, which consists of:
- Program in execution
- Data
- Stack
- Process Control Block (PCB)

Note that programs stored on disk do not become processes until they are started.
Processes

- Processes with shared memory
  - If shared memory is created by a process, it can be accessed in other processes in the system
  - This is called *memory mapping*
  - Just like named pipes, shared memory in Windows is addressable using some unique name
Chapter 3: Roadmap

3.1 What is a process?
3.2 Process states
  3.3 Process description
  3.4 Process control
  3.5 Execution of the OS
  3.6 Security issues
  3.7 Unix process management
Process States

- Process trace
  - Offsets (i.e., relative addresses) of instructions executed by a process

- CPU trace
  - Sequence of absolute addresses executed by the CPU
  - Suppose OS allows 6 CPU instructions in a slice, needs 3 to perform a process switch
Process States

• This brings us to the issue of how the OS keeps track of processes and what runs next

• Simple 2-state model:

  - Implementation:
Process States

- Process termination
  - Normal completion
  - User request (e.g., Ctrl-C)
  - Request from another process
  - Access violation
  - Arithmetic error (division by zero)
  - Invalid instruction
  - Privileged instruction
  - Not enough RAM (bad_alloc exception)

- Stealthy crashes
  - Severe stack corruption may cause program to quit without any warning or error

- If code crashes in Release mode, will it crash in Debug?
  - Not necessarily
  - Some bugs can be seen only in release mode
  - Reasons?

- What about vice versa?
Process States

• Notice that 2-state model has no simple way of selecting the next ready process
  - Some might be blocked on I/O or events

• Next version, called 5-state model, solves this:

7-state model: suspends blocked processes to disk; medium-term scheduler activates them back to RAM
Chapter 3: Roadmap

3.1 What is a process?
3.2 Process states
3.3 Process description
3.4 Process control
3.5 Execution of the OS
3.6 Security issues
3.7 Unix process management
Execution Modes

• CPU provides at least 2 execution modes
  - User mode prohibits all I/O instructions, virtual table manipulation, access to blocks of RAM not owned by process, and modification of certain registers
  - Kernel mode has no restrictions

• Some architectures allow more than 2 modes
  - These are often called protection rings
  - More granularity to allow “intermediate” privileges to certain processes (e.g., printer driver should be able to perform I/O, but not modify virtual-memory tables)

• Intel/AMD CPUs support 4 execution levels
  - Some older supercomputers had 8
Execution Modes

- Consider a hypothetical 4-ring system:
  - Ring 3 always user mode
  - Ring 0 always kernel
  - Rings 1 and 2 depend on the implementation

- Windows and Linux support only rings 0 and 3
  - Partly because other architectures these can run on (e.g., PowerPC and MIPS) traditionally had only 2 modes
  - Partly to reduce complexity

- Main drawback of 2-level systems
  - Any driver crash bluescreens the system and forces a reboot
Execution Modes

- Microsoft virtualization server (Hyper-V) is an exception
  - Virtual machines (VM) allow multiple guest OSes to run transparently on the CPU
- Guest OSes are managed by the virtual machine monitor (VMM) called hypervisor
  - In contrast to normal kernels that are called supervisors
- Hypervisor runs in ring 0, guest OS in ring 1
  - AMD-V was supported starting with Athlon 64 (2006) and Intel VT-x starting with Pentium 4 (2005)