Yet another synchronization problem

- The dining philosophers problem

- Deadlocks
  - Modeling deadlocks
  - Dealing with deadlocks
The Dining Philosophers Problem

- Philosophers
  - think
  - take forks (one at a time)
  - eat
  - put forks (one at a time)
- Eating requires 2 forks
- Pick one fork at a time
- How to prevent deadlock?
- What about starvation?
- What about concurrency?

Slide taken from a presentation by Gadi Taubenfeld, IDC
Dining philosophers: definition

- Each process needs two resources
- Every pair of processes compete for a specific resource
- A process may proceed only if it is assigned both resources
- Every process that is waiting for a resource should sleep (be blocked)
- Every process that releases its two resources must wake-up the two competing processes for these resources, if they are interested
An incorrect naïve solution

Slide taken from a presentation by Gadi Taubenfeld, IDC

Operating Systems, 2014, Meni Adler, Danny Hendler & Amnon Meisels
Dining philosophers: textbook solution

- The solution
  - A philosopher first gets
  - only then it tries to take the 2 forks.
Dining philosophers: textbook solution code

```c
#define N 5
#define LEFT (i-1) % N
#define RIGHT (i+1) % N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
semaphore mutex = 1;
semaphore s[N]; // per each philosopher

void philosopher(int i) {
    while(TRUE) {
        think();
        pick_sticks(i);
        eat();
        put_sticks(i);
    }
}
```
void pick_sticks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_sticks(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

Is the algorithm deadlock-free? What about starvation?
Textbook solution code: starvation is possible
Monitor-based implementation

```
monitor diningPhilosophers
    condition self[N];
    integer state[N];
    procedure pick_sticks(i){
        state[i] := HUNGRY;
        test(i);
        if state[i] <> EATING
            then wait(self[i]);
    }
    procedure put_sticks(i){
        state[i] := THINKING;
        test(LEFT);
        test(RIGHT);
    }
for i := 0 to 4 do state[i] := THINKING;
end monitor
```

```
procedure test(i){
    if (state[LEFT] <> EATING &&
        state[RIGHT] <> EATING &&
        state[i] = HUNGRY)
        then {
            state[i] := EATING;
            signal(self[i]);
        }
}
```
Text-book solution disadvantages

- An inefficient solution
  - reduces to mutual exclusion
  - not enough concurrency
  - Starvation possible
The LR Solution

- If the philosopher acquires one fork and the other fork is not immediately available, she holds the acquired fork until the other fork is free.

- Two types of philosophers:
  - L -- The philosopher first obtains its left fork and then its right fork.
  - R -- The philosopher first obtains its right fork and then its left fork.

- The LR solution: the philosophers are assigned acquisition strategies as follows: philosopher $i$ is R-type if $i$ is even, L-type if $i$ is odd.

*Slide taken from a presentation by Gadi Taubenfeld, IDC*
Theorem: The LR solution is starvation-free

Assumption: “the fork is fair”.

(........ means “first fork taken”)
Deadlocks

- The dining philosophers problem

- Deadlocks
  - Modeling deadlocks
  - Dealing with deadlocks
Synchronization: Deadlocks
Deadlocks

- Deadlock of Resource Allocation:
  - Process A requests and gets Tape drive
  - Process B requests and gets Fast Modem
  - Process A requests Fast Modem and *blocks*
  - Process B requests Tape drive and *blocks*

- Deadlock situation: Neither process can make progress and no process can release its allocated device (resource)
- Both resources (devices) require *exclusive access*
Resources

- Resources - Tapes, Disks, Printers, Database Records, semaphores, etc.
- Some resources are non-preemptable (i.e. tape drive)
- It is easier to avoid deadlock with preemptable resources (e.g., main memory, database records)
- Resource allocation procedure
  - Request
  - Use
  - Release only at the end – and leave
- Block process while waiting for Resources
Defining Deadlocks

A set of processes is deadlocked if each process is waiting for an event that can only be caused by another process in the set.

Necessary conditions for deadlock:

1. **Mutual exclusion**: exclusive use of resources
2. **Hold and wait**: process can request resource while holding another resource
3. **No preemption**: only holding process can release resource
4. **Circular wait**: there is an oriented circle of processes, each of which is waiting for a resource held by the next in the circle
Modeling deadlocks

- modeled by a **directed graph** (resource graph)
  - Requests and assignments as **directed edges**
  - Processes and Resources as vertices
- **Cycle** in graph means **deadlock**

---

**Process A holds** resource Q

**Process B requests** resource Q

**Deadlock**
Different possible runs: an example

Round-robin scheduling:
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
Different possible runs: an example

An alternative scheduling:

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
Multiple Resources of each Type

Operating Systems, 2014, Meni Adler, Danny Hendler & Amnon Meisels
A Directed Cycle But No Deadlock
Resource Allocation Graph With A Deadlock

Operating Systems, 2014, Meni Adler, Danny Hendler & Amnon Meisels
Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, deadlock possible
Dealing with Deadlocks

- The dining philosophers problem
- Deadlocks
  - Modeling deadlocks
  - Dealing with deadlocks
Dealing with Deadlocks

Possible Strategies:

- Prevention
  structurally negate one of the four necessary conditions
- Avoidance
  allocate resources carefully, so as to avoid deadlocks
- Detection and recovery
- Do nothing (The “ostrich algorithm”)
  deadlocks are rare and hard to tackle... do nothing

Example: Unix - process table with 1000 entries and 100 processes each requesting 20 FORK calls... Deadlock.
users prefer a rare deadlock over frequent refusal of FORK
Deadlock prevention

Attack one of the 4 necessary conditions:

1. Mutual exclusion
   - Minimize exclusive allocation of devices
   - Use spooling: only spooling process requests access (not good for all devices - Tapes; Process Tables); *may fill up spools (disk space deadlock)...

2. Hold and Wait
   - Request all resources immediately (before execution)
   - *Problem: resources not known in advance, inefficient

   or

   - to get a new resource, free everything, then request everything again (including new resource)
3. No preemption
   o Not always possible (e.g., printer, tape-drive)

4. Circular wait condition
   o Allow holding only a single resource (*too restrictive*)
   o Number resources, allow requests only in ascending order:
     Request only resources numbered higher than anything currently held

*Impractical in general*
Deadlock Avoidance

- System grants resources *only if it is safe*
- basic assumption: **maximum resources required by each process is known in advance**

**Safe state:**
- Not deadlocked
- There is a scheduling that satisfies all possible future requests
Safe states: example

- Both have printer
- Both have plotter
- Both have both

Unsafe state

Printer
Plotter

Operating Systems, 2014, Meni Adler, Danny Hendler & Amnon Meisels
Safe and Unsafe states (single resource)

- Safe state:
  - Not deadlocked
  - There is a way to satisfy all possible future requests

![Resource Allocation States](image)

Fig. 6-9. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.
Banker's Algorithm, Dijkstra 1965 (single resource)

- Checks whether a state is safe

  1. Pick a process that can terminate after fulfilling the rest of its requirements (enough free resources)
  2. Free all its resources (simulation)
  3. Mark process as terminated
  4. If all processes marked, report “safe”, halt
  5. If no process can terminate, report “unsafe”, halt
  6. Go to step 1
Multiple resources of each kind

- Assume $n$ processes and $m$ resource classes
- Use two matrixes and two vectors:
  - Current allocation matrix $C_{n \times m}$
  - Request matrix $R_{n \times m}$ (remaining requests)
  - Existing resources vector $E_m$
  - Available resources vector $A_m$
Banker’s Algorithm for multiple resources

1. Look for a row of R whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock.

2. Otherwise, assume the process of the row chosen finishes (which will eventually occur). Mark that process as terminated and add the i’th row of C to the A vector.

3. Repeat steps 1 and 2 until either all processes are marked terminated, which means safe, or until a deadlock occurs, which means unsafe.
deadlock avoidance – an example with 4 resource types, 5 processes

<table>
<thead>
<tr>
<th>Tape-drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD-ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>(1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

\[
T = \begin{pmatrix}
A & 3 & 0 & 1 & 1 \\
B & 0 & 1 & 0 & 0 \\
C & 1 & 1 & 1 & 0 \\
D & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[
R = \begin{pmatrix}
A & 1 & 1 & 0 & 0 \\
B & 0 & 1 & 1 & 2 \\
C & 3 & 1 & 0 & 0 \\
D & 0 & 0 & 1 & 0 \\
E & 2 & 1 & 1 & 0 \\
\end{pmatrix}
\]

Is the current state safe? Yes, let’s see why...

We let D run until it finishes
deadlock avoidance – an example with 4 resource types, 5 processes

<table>
<thead>
<tr>
<th></th>
<th>Tape-drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD-ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(6, 3, 4, 2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>(2, 1, 2, 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ C = \begin{array}{ccc}
A & 3 & 0 & 1 & 1 \\
B & 0 & 1 & 0 & 0 \\
C & 1 & 1 & 1 & 0 \\
D & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 0 & 0 \\
\end{array} \]

\[ R = \begin{array}{ccc}
A & 1 & 1 & 0 & 0 \\
B & 0 & 1 & 1 & 2 \\
C & 3 & 1 & 0 & 0 \\
D & 0 & 0 & 0 & 0 \\
E & 2 & 1 & 1 & 0 \\
\end{array} \]

We now let E run until it finishes

Next we let A run until it finishes
deadlock avoidance – an example with 4 resource types, 5 processes

Tape-drives  Plotters  Scanners  CD-ROMs
E = (6  3  4  2)
A = (5  1  3  2)

C =

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>P</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

R =

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>P</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Finally we let B and C run.
If B now requests a Scanner, we can allow it.
This is still a safe state...

<table>
<thead>
<tr>
<th>Tape-drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD-ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(6, 3, 4, 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>(1, 0, 1, 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
A & 3 & 0 & 1 & 1 \\
B & 0 & 1 & 1 & 0 \\
C & 1 & 1 & 1 & 0 \\
D & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
A & 1 & 1 & 0 & 0 \\
B & 0 & 1 & 0 & 2 \\
C & 3 & 1 & 0 & 0 \\
D & 0 & 0 & 1 & 0 \\
E & 2 & 1 & 1 & 0
\end{bmatrix}
\]

If E now requests a Scanner, granting the request leads to an unsafe state.
This state is unsafe

Tape-drives  Plotters  Scanners  CD-ROMs
E = (6 3 4 2)
A = (1 0 0 0)

We must not grant E’s request
Deadlock Avoidance is not practical

- Maximum resource request per process is unknown beforehand
- Resources may disappear
- New processes (or resources) may appear
Deadlock Detection and Recovery

- Find if a deadlock exists
- if it does, find which processes and resources it involves
- Detection: detect cycles in resource graph
- Algorithm: DFS + node and arc marking
Find cycles:

For each node, N, in the graph, perform the following 5 steps with N as the starting node

1. Initialize \( L \) to the empty list and designate all arcs as \textit{unmarked}
2. Add the current node to the \textit{end of} \( L \) and check if the node appears twice in \( L \). If it does, the graph contains a cycle, terminate.
3. If there are any \textit{unmarked arcs} from the given node, go to 4., if not go to 5.
4. Pick an unmarked outgoing arc and mark it. Follow it to the new current node and go to 2.
5. We have reached a \textit{deadend}. Go back to the previous node, make it the current node and go to 3. If all arcs are marked and the node is the initial node, there are no cycles in the graph, terminate
Detection - extract a cycle

1. Process A holds R and requests S
2. Process B holds nothing and requests T
3. Process C holds nothing and requests S
4. Process D holds U and requests S and T
5. Process E holds T and requests V
6. Process F holds W and requests S
7. Process G holds V and requests U

Fig. 6.3. (a) A resource graph. (b) A cycle extracted from (a).
When should the system check for deadlock?

- Whenever a request is made - too expensive
- every $k$ time units...
- whenever CPU utilization drops below some threshold
  (indication of a possible deadlock..)
Recovery

- Preemption - possible in some rare cases
  *temporarily take a resource away from its current owner*

- Rollback - possible with checkpointing
  *Keep former states of processes (checkpoints) to enable release of resources and going back*

- Killing a process - easy way out, may cause problems in some cases, depending on process being *rerunable*...

- Bottom line: *hard to recover from deadlock, avoid it*
Example - deadlocks in DBMSs

- For database records that need locking first and then updating (two-phase locking)
- Deadlocks occur frequently because records are dynamically requested by competing processes
- DBMSs, therefore, need to employ deadlock detection and recovery procedures
- Recovery is possible - transactions are "checkpointed" - release everything and restart
Additional deadlock issues

- Deadlocks may occur with respect to actions of processes, not resources - waiting for semaphores.
- Starvation can result from a bad allocation policy (such as smallest-file-first, for printing) and for the “starved” process will be equivalent to a deadlock (cannot finish running).
- Summary of deadlock treatment:
  - Ignore problem
  - Detect and recover
  - Avoid (be only in safe states)
  - Prevent by using an allocation policy or conditions.
The Situation in Practice

- Most OSs in use, and specifically Windoes, Linux..., ignore deadlock or do not detect it
- Tools to kill processes but usually without loss of data
- In Windows NT there is a system call *WaitForMultipleObjects* that requests all resources at once
  - System provides all resources, if free
  - There is no lock of resources if only few are free
  - Prevents Hold & Wait, but difficult to implement!
Linux: the Big Kernel Lock

Linux was first designed with Coarse-Grained Locking

- The whole kernel was wrapped in a giant lock around it to avoid deadlocks (kernel / interrupt handlers / user threads) introduced in Linux 2.0.

- Work began in 2008 to remove the big kernel lock: [http://kerneltrap.org/Linux/Removing_the_Big_Kernel_Lock](http://kerneltrap.org/Linux/Removing_the_Big_Kernel_Lock)

- It was carefully replaced with fine-grained locks until it was removed in Linux 2.6.39 (in 2011!) [https://lwn.net/Articles/424657/](https://lwn.net/Articles/424657/)
Xv6 and deadlocks

- Xv6 uses a few coarse data-structure specific locks; for example, xv6 uses a single lock protecting the process table and its invariants, which are described in Chapter 5.

- A more fine-grained approach would be to have a lock per entry in the process table so that threads working on different entries in the process table can proceed in parallel.

- However, it complicates operations that have invariants over the whole process table, since they might have to take out several locks.

- To avoid such deadlocks, all code paths must acquire locks in the same order. Deadlock avoidance is another example illustrating why locks must be part of a function’s specification: the caller must invoke functions in a consistent order so that the functions acquire locks in the same order.

- Because xv6 uses coarse-grained locks and xv6 is simple, xv6 has few lock-order chains. The longest chain is only two deep. For example, ideintr holds the ide lock while calling wakeup, which acquires the ptable lock.