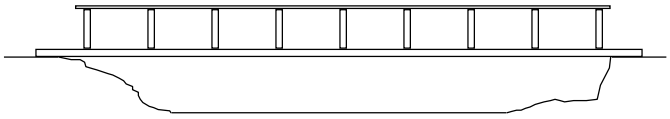
Deadlocks

- The Deadlock Problem
- Examples
- Interlude on Mars
- Resource and system model, and exact definitions
- Solutions:
  - Prevention
  - Avoidance
  - Detection and recovery
- *Reading: Silberschatz, Chapter 7*

The Deadlock Problem

- When some processes are blocked on resource requests that can never be satisfied unless drastic systems action is taken, the processes are said to be *deadlocked*.
  - In modern computer systems, possibilities for deadlocks have increased:
    - dynamic resource sharing
    - parallel programming
    - communicating processes
- Example: River crossing on a narrow bridge



  - need an agreed-upon protocol

Examples of Deadlocks

File Sharing

```
P1:              P2:
...              ...
Request(D);      Request(T);
Request(T);      ...
...              Request(D);
...              ...
Release(T);      Release(D);
Release(D);      Release(T);
```

Single Resource Sharing

A single resource *R* contains *m* allocation units, and is shared by *n* processes, and each process accesses *R* in the sequence

Req(R);Req(R);Rel(R);Rel(R);

Example: shared buffers in I/O subsystem

Locking in Database Systems

If locking done at any level lower than entire database, deadlock can occur.

```
P1:              P2:
lock(R1);        lock(R2);
...              ...
lock(R2);        lock(R1);
...              ...
```

An Extreme Example

(Holt 1971) in PL/I

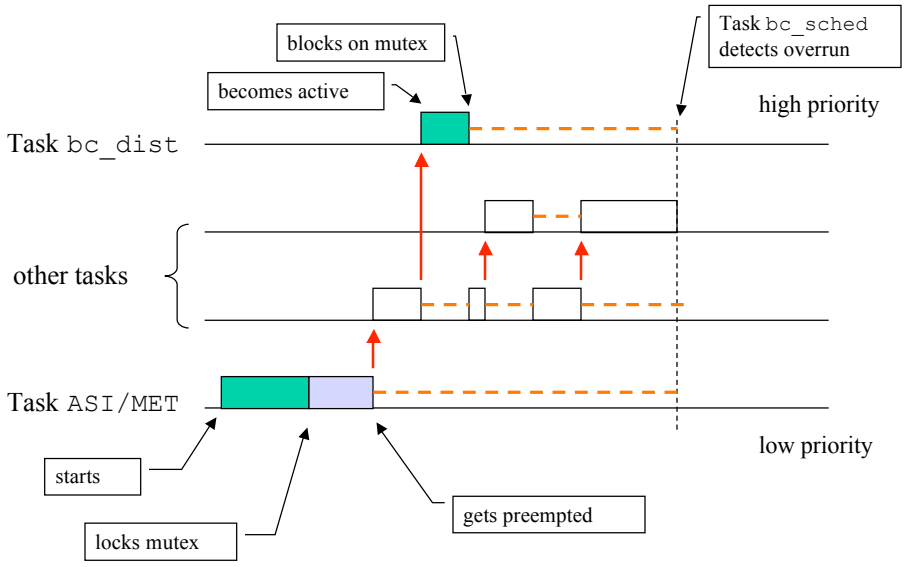
```
revenge: procedure
   options(main,task);
   wait(event);
end {revenge}
```

Interlude: Not-Quite-Deadlock … on Mars!

- Landing on July 4, 1997
- "experiences software glitches"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.
- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.
- Resources:

  http://research.microsoft.com/~mbj/Mars_Pathfinder/

Priority Inversion on Mars Pathfinder



The Resource Model

- Finite number of serially reusable **resources** R_1, ..., R_m.
- Serially reusable:
  - number of units is constant
  - either available or allocated to exactly one process (no sharing)
  - process may release a unit only if it previously acquired it.
- Set of **processes** P_1, P_2, ..., P_n.
- Operations on resources:
  - **request**: If request cannot be granted, wait until some other process releases resource
  - **use**
  - **release**

Necessary Conditions for Deadlocks

1. **Mutual exclusion**: If two processes request a resource, at least one must wait until the resource has been released.
2. **Hold and wait**: At least one process must be holding a resource and be waiting to acquire additional resources.
3. **No preemption**: Resources can only be released voluntarily by a process.
4. **Circular wait**: *(see next slides)*

Resource Allocation Graphs

System Resource Allocation Graph

$G = (V, E)$

$V = \{P,R\}$ = vertices

E = edges

where

$P = \{P_1, P_2, ..., P_n\}$: set of **processes**.

$R = \{R_1, R_2, ..., R_m\}$: set of **resources**.

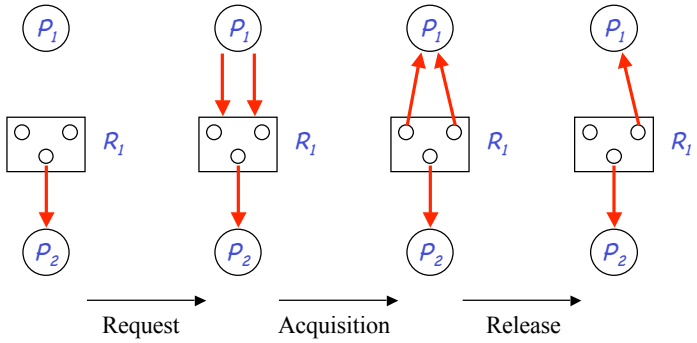
Edges represent *waiting-for* or *allocated-to* relations.

- (P_i, R_j) in G : Process P_i is waiting for Resource R_j (**request edge**)

- (R_j, P_i) in G : Resource R_j is allocated to Process P_i (**assignment edge**)

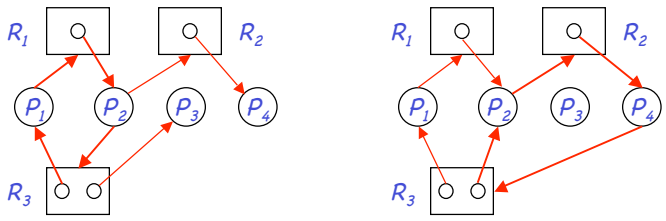
Resource Allocation Graphs: Example

$V = (P = \{P_1, P_2\}, R = \{R_1\})$
$E_{initial} = \{(R_1, P_2)\}$
$E_{final} = \{(R_1, P_2), (R_1, P_1)\}$



Resource Allocation Graphs and Deadlocks

- Observation 1: If a RAG does not have a cycle, then no process is deadlocked.
- Observation 2: If a RAG has a cycle, then a deadlock may exist.
- The existence of cycles in the RAG is necessary but not sufficient for a deadlock.
- Example:



cycle, no deadlock

cycle, deadlocked

Special Cases

- **Single-unit resources**: A cycle becomes a sufficient and necessary condition for deadlock:
  - necessary: shown earlier
  - sufficient: Every process in a cycle *C* must have an entering and an exiting edge. Therefore, it must hold a resource in *C* while it has an outstanding request for resources in *C*. Every resource in *C* is held by some process in *C*. Therefore, every process in *C* is blocked by a resource in *C* that can be made available only by a process in *C*.

Deadlock Prevention

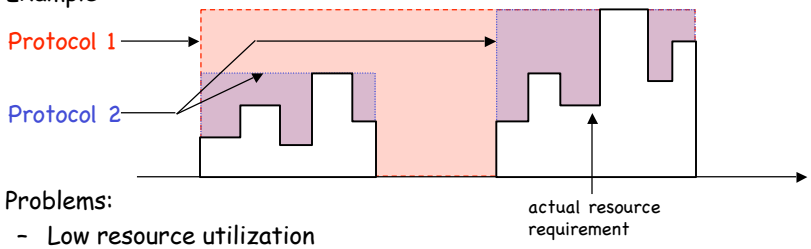
Prevent occurrence of deadlock by preventing occurrence of any one of the 4 necessary conditions for deadlock.

Deadlock Prevention: (1) Mutual Exclusion

- A processor never needs to wait for shareable resources.
- Make resources shareable!
- Fine with read-only files (may not need exclusive access)
- Huh?! A shareable lock?!

Deadlock Prevention: (2) Hold and Wait

- Guarantee that a processor requesting resources does not hold resources already.
  - Protocol 1: Assign resources at beginning of execution.
  - Protocol 2: Allow process to request resources only if it has none.
- Example:



- Problems:
  - Low resource utilization
  - Starvation

Deadlock Prevention: (3) No Preemption

- Make resources preemptive.
- Example protocols:
  - Preempt resources held by a process when that process is denied request of a resource.
  - Preempt resource held by a process when that particular resource is requested by another process.
- Problem: Some resources are inherently non-preemptive.
  - Message slots on communication links, printer, tapes, locks.

Deadlock Prevention: (3) Circular Wait

- Impose a total ordering on resources and request resources in increasing order.
- Ordering:

$$F : R \rightarrow N$$

- Request resources in order of their increasing value of F.
- No circular wait condition can occur.

Deadlock Avoidance

- Deadlock **prevention**: restrict the way how requests can be made *a priori*. Problem: low device utilization
- **Alternative**: Treat each request individually, and temporarily delay it when it may cause a deadlock later.
- Need additional information about requesting process: How much information?
  only current request *vs.* complete request sequence
- Compromise: e.g. information about which resources process may request in the future (and maximum amount of each). Example:
  - Database application: 2 locks per database, 20 blocks of memory, 10 blocks of temporary disk space
  - Scientific computation: 300 blocks of memory, 500 blocks of temporary disk space, printer.

Resource Allocation States

- **Resource allocation state**: Number of allocated resources, available resources, maximum claims of processes.
- **Safe sequence**: Sequence of process execution (P_1, ..., P_n) (each process runs to completion) such that all processes can successfully terminate, starting from given resource allocation state.
- **Safe** resource allocation state: There is at least one safe sequence for the state.
- **Unsafe** resource allocation state: No safe sequence exists.
- Unsafe states may lead to deadlocks.

A Scheme for Deadlock Avoidance

- **Observation 1:** A system in a safe state is not deadlocked.
- **Observation 2:** Delaying a request does not change a safe state into an unsafe state.
- **Scheme:** Whenever a process requests a resource that is available, check whether granting the request would move the system into an unsafe state. If so, delay the request.
- **Problem:** Reduction of resource utilization.

The Banker's Algorithm (Dijkstra, Haberman)

- Have every process declare its maximum resource requirements (*i.e.* maximum number of units required for each resource).
- Whenever process requests resources, determine (in the `request()` routine) if granting the request at this time leaves system in safe state. If not, delay the request.
- Data structures:

```
int available[m];/* units of R_j available */
int max_i[m];    /* maximum resource requirements of P_i */
int alloc_i[m];  /*current allocation of resources to P_i*/
int need_i[m];   /* need_i[j] = max_i[j] - alloc_i[j]      */
```

- Partial relation "<=" on vectors:

  x in N^m, y in N^m : x <= y iff for all i = 0,...,m-1 : x[i] <= y[i]

  <1,1,1> <= <2,5,7>

  <1,1,1> NOT <= <2,0,7>

The Banker's Algorithm

P_i:

```
void request(int req_vec[]) {
  if (req_vec >= need_i)
      raise_hell();       /* exceeded promised maximum      */
  if (req_vec >= available)
      wait();             /* resources not available        */
  available -= req_vec;
  alloc_i    += req_vec;
  need_i     -= req_vec;
  if (! state_is_safe()) {
      available += req_vec; /* restore old state            */
      alloc_i    -= req_vec;
      need_i     += req_vec;
      wait();             /* wait until state would be safe */
  }
}
```

Determine Safety of State

```
int state_is_safe() {
  int  temp_av[m] = available;
  bool finish[n]  = (FALSE,...,FALSE);
  int  i;
  while (finish!=(TRUE,...TRUE)){
    /* Find P_i such that finish[i] = FALSE and */
    /* need_i <= temp_av.                       */
    for (i=0; (i<n)&&(finish[i]||(need_i > temp_av); i++) {
      if (i == n) {
        return FALSE;
      }
      else {
        temp_av  += alloc_i;
        finish[i] = TRUE;
      }
    }
  }
  return TRUE;
}
```

Banker's Algorithm: Example

		R1(7)	R2(7)	R3(7)
max:	P1	5	3	1
	P2	3	2	3
	P3	2	3	1
	P4	5	0	3
alloc:	P1	3	3	1
	P2	2	2	2
	P3	0	1	1
	P4	0	0	1
need:	P1	2	0	0
	P2	1	0	1
	P3	2	2	0
	P4	5	0	2
available:		2	1	2

Four examples:

P2: request([1,1,1])

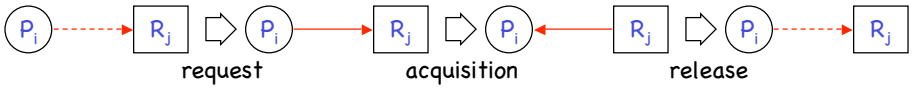
P2: request([1,0,1])

P4: request([5,0,0])

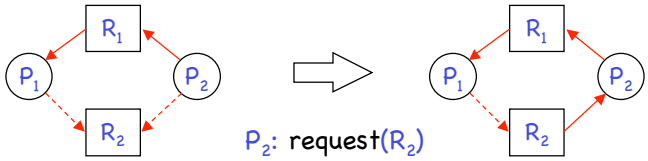
P3: request([2,0,0])

Single-Unit Resources: Claim Graphs

- Claim graph: Variation of resource allocation graph (RAG).
  - claim edge (P_i, R_j) : Process P_i may request resource R_j sometimes in the future.



- Whenever new process starts, we add its claim edges to the RAG.
- Whenever process request resources, test if acquisition would generate a cycle in the RAG (causing an unsafe state).



Deadlock Detection & Recovery

- Deadlock prevention and avoidance are cautious approaches. May overly reduce resource utilization.
- Alternative: Periodically analyze RAG, detect deadlocks, and initiate recovery.
- Advantages:
  - *A priori* knowledge of resource requirements not needed.
  - Higher resource utilization
- Disadvantages:
  - Cost of recovery

Multiple-Unit Resources

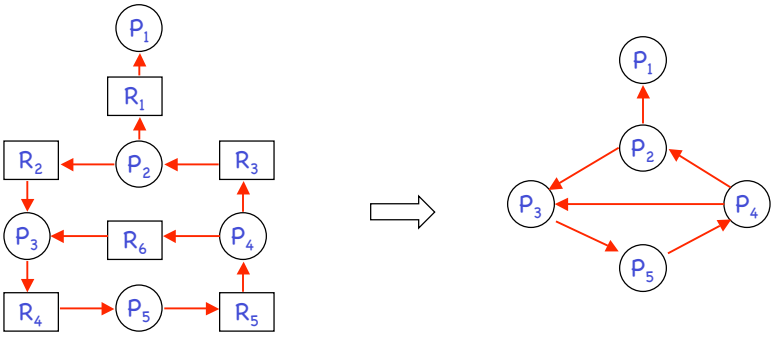
```
int available[m]; /* resources available      */
int alloc_i[m];    /* resources allocated to P_i */
int rec_vec_i[m];  /* currently requested by P_i */

int temp_av[m] = available;
bool finish[n] = (FALSE, ..., FALSE);
bool found     = TRUE;
for (i=0, i<n, i++)
  if (rec_vec_i == (0,...,0)) finish[i] = TRUE;
while(found) {
  found = FALSE;
  for(i=0, (i<n) && (!found), i++) {
    if ((!finish[i]) && (req_vec_i < temp_av))
      /* assume P_i runs to completion */
      {temp_av += alloc_i; finish[i]=TRUE; found=TRUE;}
  }
}
/* for any finish[i] == FALSE, P_i is deadlocked */
```

Deadlock Detection: Example

		R1(7)	R2(7)	R3(7)
alloc:	P1	2	3	0
	P2	2	2	2
	P3	3	1	1
	P4	0	0	4
req:	P1	0	0	0
	P2	1	0	1
	P3	2	0	3
	P4	5	0	0
available:		0	1	0

Single-Unit Resources: Wait-For Graphs

- Wait-For Graph: "RAG without resource nodes"
- Example:



resource allocation graph *wait-for graph*

- Cycle in wait-for graph is necessary and sufficient condition for deadlock.

Cycle Detection in Wait-For Graphs

```
/* w_i : out-degree of node i */
S := {i | node i is a sink};

for all i in S do begin
  for all j such that (j,i) is edge do begin
    delete_edge(j,i);
    w_j := w_j-1;
    if w_j = 0 then S := S + {j};
  end;
end;

if (S <> N) then cycle_exists;
```

Cycle Detection in Directed Graphs (Pseudocode)

How to Use Deadlock Detection:

- How frequently to invoke deadlock detection:
  - after every request *vs.* at longer intervals
  - indication-triggered (e.g. drop in CPU utilization)
- Recovery from deadlock:
  - Termination of deadlocked processes.
  - Preemption of resources (may require process rollback)
  - Policies for termination/rollback.

The Engineer's Approach

Q: Why not ignore deadlocks altogether?

- The Ostrich Algorithm (Tanenbaum): pretend there is no problem
- Deadlocks in Unix:
  - process table: size limits total number of processes
    - scenario:
      - process table with 100 entries.
      - 10 processes that fork off 12 subprocesses each.
  - i-node table: size limits the number of open files.
  - *etc.*
- Most users prefer an occasional deadlock to a rule that unduly restricts them.