Interprocess Synchronization and Communication
Remote Procedure Calls (RPC)

• Higher-level, procedural interface to IPC

• To the programmer: looks like a regular procedure call
  • Procedure is in a separate address space, does not share global variables

• Each RPC needs separate process
  • Reads parameters, runs remote procedure, returns result
  • Done using send and receive primitives . . .
Implementation

**caller process**

send(RP_guard, parameters) → receive(caller, parameters)

receive(RP_guard, results) ← send(caller, results)

**RP_guard process**

receive(caller, parameters) → RP parameters

send(caller, results)
Example: Producer Consumer Problem

procedure producer;
begin
  while true do begin
    // produce a nextp
    send("RP_guard", nextp);
    end;
end;

procedure consumer;
begin
  while true do begin
    receive("RP_guard", nextc);
    // consume nextc
    end;
end;
Common Concurrency Problems

• Atomicity violation bugs
• Order violation. bugs
• Livelock
• Deadlock
Atomicity Violation Problems

• When an operation that is supposed to be indivisible is not
• Simple example: checking permission to access file, then accessing it

```c
if (access(file1, W_OK) < 0 ||
    (fd = open(file1, O_WRONLY)) < 0){
    perror(file1);
    exit(1);
}
```
Atomicity Violation Problems

```c
if (access(file1, W_OK) < 0 ||
    (fd = open(file1, O_WRONLY)) < 0){
    perror(file1);
    exit(1);
}
```

Attacker executes:
```
rm file1
ln /etc/passwd logfile
```
Example from MySQL

• Thread 1
  if (thd->proc_info) {
    fputs(thd->proc_info, ...);
  }

• Thread 2
  thd->proc_info = NULL;
Order Violation Bug

- Two operations should be done in one order, but instead are done in another order
- First order works
- Second order causes problems
Order Violation Bug

• Thread 1

```c
void init() {
    mThread = PR_CreateThread(mMain, ...);
}
```

• Thread 2

```c
void mMain(...) {
    mState = mThread->State;
}
```
The Fix

• When available, use locks (semaphores, etc.) to create a critical section among the statements to ensure indivisibility or specific order.
Livelock

• Processes loop, neither advancing until the other does
• “Livelock” as processes are active
• Example: proposed software solution #3 for concurrency

```
var interested: array[0..1] of boolean = false;
    // who wants to enter critical section
interested[i] = true;  // ... entry section
while interested[j] do
    /* nothing */
    . . .  // ... critical section
interested[i] = false;  // ... exit section
```
Deadlock

- Resource manager: the part of the kernel responsible for managing resources
  - request: asks the resource manager to give the process a resource
  - release: informs resource manager that process no longer needs a resource that it has been given
Example

• A system has 2 devices, $a$ and $b$
• A system has 2 processes $p$ and $q$
• The following occurs
  • $p$ requests device $a$, and resource manager allocates it to $p$
  • $q$ requests device $b$, and resource manager allocates it to $q$
  • $p$ requests device $b$, but resource manager cannot allocate it, so $p$ blocks until $b$ becomes free
  • $q$ requests device $a$, but resource manager cannot allocate it, so $q$ blocks until $a$ becomes free
• Processes $p$ and $q$ are now deadlocked
Deadlock vs. Starvation

• Deadlock occurs when a needed resource is never available for reallocation

• Starvation occurs when a needed resource is available for reallocation but never assigned to the process requesting it
  • Example: the dining philosopher’s problem, where everyone picks up left fork, and puts it down, and picks it up again . . .
Approaches to Allocation

• *Liberal*: whenever a request can be granted, do so; if not, block process until request can be granted

• Conservative: be willing to deny a request on occasion to prevent deadlock

• Serialization: processes cannot hold resources concurrently, so if one process requests and is granted a resource, no other process can acquire another resource
  - *Example*: in 2 device example, once $p$ acquires $a$, $q$’s request for $b$ would be denied
Resource Types

- **Reusable resources**: these have a fixed total inventory: none are created, and none destroyed.
  - Units are requested and acquired from a pool of available units and after use are returned to the pool where other processes can get them.
  - *Examples*: processors, memory, tape drives, etc.

- **Consumable resources**: have no fixed number of units; created (produced) or acquired (consumed) as needed
  - Unblocked producer may release any number of units which become immediately available; once acquired, units cease to exist.
  - *Examples*: messages, information in I/O buffers, etc.

- **We will not discuss deadlock analysis of consumable resources.**
Policies to Handle Deadlock

• *Ignore it*: okay if deadlocks are rare and users know how to recover

• *Prevention*: ensure deadlock can never occur
  • If granting request could cause deadlock, deny request
  • 4 conditions must hold for deadlock to occur

• *Avoidance*: use knowledge of the process’ future behavior to constrain the pattern of resource allocation

• *Detection and recovery*: determine when a system, processes are deadlocked and recover from it
  • Most useful when deadlocks infrequent and cost of recovery is low
Deadlock Prevention

• A *safe state* is one that can never lead to deadlock
• So restrict the system so all states are safe
• Several designs for this, all based on breaking 1 of 4 conditions all of which must hold for deadlock to be possible
Deadlock Prevention

Deadlock requires 4 conditions to hold simultaneously:

• **Mutual exclusion**: when a process has acquired a resource, no other process can acquire it

• **No preemption**: when a process has acquired a resource, it cannot be reallocated until the process releases it

• **Circular wait, resource waiting**: blocked processes form a circular chain, with each holding a resource requested by another member of the chain and holding a resource held by another member of the chain

• **Hold and wait, partial allocation**: a process may request resources while holding other resources
Circular Wait

process 1 → acquired → resource 1

resource 1 → requested → resource 2

resource 2 → acquired → process 2
Deadlock Prevention

• Only 1 process at a time may hold resources
  • Breaks circular wait as process 2 can never acquire resources while process 1 has any resources
  • Effectively eliminates multiprogramming

• Processes must request, and acquire, all resources it might need at one time
  • Breaks circular wait as no process can wait on a resource allocated to another process
  • Resources may be requested but never used
  • Resources may be allocated long before use
Deadlock Prevention

• Classes of resources are ordered, and constraints placed upon ordering resources in different classes
  • Called hierarchical ordering policy or ordered resource policy

• How: divide resources into $n$ classes
  • Process can request allocations from class $c_i$ if and only if it has no allocation from classes $c_{i+1}, \ldots, c_n$
  • If it needs to get such a resource, it must release all resources it has and request them too
  • Breaks hold and wait as processes do not hold resources when blocked awaiting another resource assignment

• Some resources must be allocated before a process needs it
Deadlock Avoidance

• Use Banker’s Algorithm, which determines if system is in a safe or unsafe state by trying to finish

• Example: if a request is granted, then after that:
  process $p_1$ has 4 resource units, needs 4 more
  process $p_2$ has 2 resource units, needs 1 more
  process $p_3$ has 2 resource units, needs 7 more
  2 resource units are available
Deadlock Avoidance

1. Satisfy \( p_2 \); then
   process \( p_1 \) has 4 resource units, needs 4 more
   process \( p_3 \) has 2 resource units, needs 7 more
   4 resource units are available

2. Satisfy \( p_1 \); then
   process \( p_3 \) has 2 resource units, needs 7 more
   8 resource units are available

3. Satisfy \( p_3 \); all processes finished

So this is a safe state and the request is granted
Deadlock Avoidance

• Example: if a request is granted, then after that:
  process $p_1$ has 4 resource units, needs 4 more
  process $p_2$ has 2 resource units, needs 1 more
  process $p_3$ has 3 resource units, needs 6 more
  1 resource unit are available
Deadlock Avoidance

1. Satisfy $p_2$; then
   process $p_1$ has 4 resource units, needs 4 more
   process $p_3$ has 3 resource units, needs 6 more
   3 resource units are available

$p_1, p_3$ cannot finish

So this is an unsafe state and the request is denied
Problems with Banker’s Algorithm

1. Banker’s algorithm requires a fixed number of resources
   • If something goes off line for repair or maintenance, the system may be put into an unsafe state without any action by the processes;

2. Banker’s algorithm requires a fixed number of processes
   • This is unreasonable, especially in time sharing systems.

3. Banker's algorithm guarantees all requests will be granted in a finite time
   • But printing your program (due today) next year grants your request in a finite time. You need a better guarantee than that!
Problems with Banker’s Algorithm

4. Banker's algorithm requires jobs to release their resources in a finite time
   • Suppose a process grabs a resource and then blocks indefinitely, waiting for an external event to occur. Again, you need a better guarantee that that!

5. Banker's algorithm requires users to know and state process needs in advance.
   • Infeasible in many cases (especially in time-sharing)
Deadlock Detection and Recovery

• System generates a resource graph
• It looks for loops
• If it finds one, it breaks it
  • It can reallocate resources
  • It can terminate processes