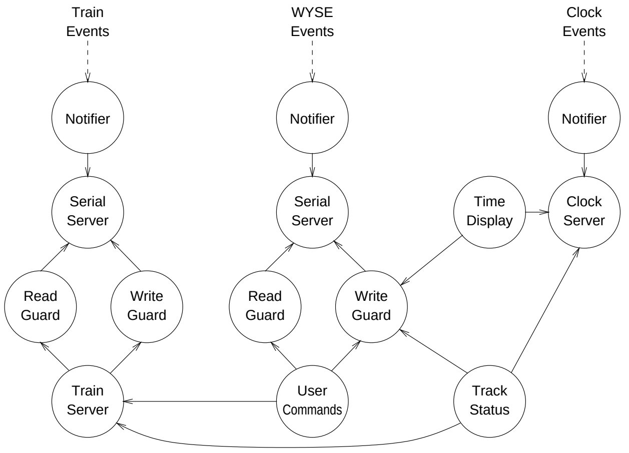
CS452/652 Real-Time Programming Course Notes

Daniel M. Berry, Cheriton School of Computer Science
University of Waterloo

Assigment 1 Process Structure



Process

This graph shows only Sends.

Replys, in the opposite directions are implied; so it is not necessary to show them.

Also, later, a potential deadlock detection algorithm depends on having only Send arcs.

Steady State

The diagram represents the steady state after all initialization is done.

∴, communication during initialization, e.g. with the name server, is not included.

You Already Know

You already know about:

- the various events,
- notifiers,
- serial servers,
- guards, and
- the clock server.

We talk about what is new.

Train Server

The train server

- receives high-level train commands from tasks,
- issues commands to track, and
- replies track information.

User Commands

The user commands process

- prompts user and
- issues commands.

Track Status

The track status process

- requests sensor updates from train server,
- gets current time, and
- displays updates on the WYSE.

Time Display

The time display process

- does GetTime() and
- displays time on the WYSE.

Priority Inversion

Priority inversion (PI) occurs when a task is forced to wait on another task of lower priority.

E.g.,

Task t_1 with priority p_1, ready

Task t_2 with priority p_2, ready

Task t_3 with priority p_3, ready

$p_1 < p_2 < p_3$. $\therefore$ t_3 is running.

Priority Inversion, Cont’d

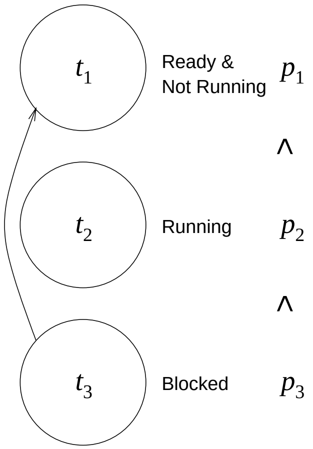
Two different scenarios.

Scenario 1

t_3 sends to t_1

$\rightarrow$ t_3 waits while t_2 runs

Scenario 1, Cont'd



t_1 Ready & Not Running p_1

$\wedge$

t_2 Running p_2

$\wedge$

t_3 Blocked p_3

Scenario 2

t_3 is blocked.

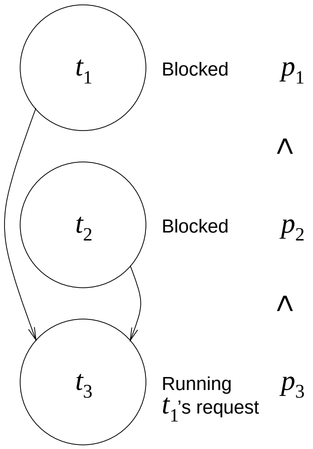
t_2 sends to t_3 and blocks.

t_1 sends to t_3 and blocks.

t_3 gets unblocked.

→ one possibility is that t_2 waits while t_3 services t_1's request.

Scenario 2, Cont'd



t_1 Blocked p_1

$\wedge$

t_2 Blocked p_2

$\wedge$

t_3 Running t_1's request p_3

Scenarios Only Possible

Each of these scenarios is possible, not guaranteed.

But the possibility is enough to cause problems if the possibility becomes a reality.

Duration of an Inversion

The duration of a priority inversion can be unbounded or uncontrolled.

Real-Life Example of PI

In the Mars Pathfinder, tasks communicate via an information bus.

- The bus management task, that moves data through the bus, is of high priority.

Real-Life Example of PI, Cont’d

- The meterological data gathering task runs infrequently and is of low priority. This task uses the bus directly, by
  - acquiring a semaphore, writing to the bus, and releasing the semaphore, the same semaphore the bus management task uses to access the bus,
  - so as not to interfere with the information bus management task.
- The communications task is of medium priority.

These priority assignments make sense.

Problematic Scenario

A HW interrupt causes the bus management task to wake up.

However, the meterological data gathering task holds the semaphore.

∴, the bus manager must wait until data gathering task gives up the semaphore.

Priority inversion!

Problematic Scenario, Cont’d

This priority inversion is normally not a problem.

However, if the medium priority task gets scheduled before the semaphore is released, then the bus management task cannot run.

The implemented solution: Eventually a watchdog task detects that the bus manager has not run for some time, concludes that there is a problem, and resets the system.

Problematic Scenario, Cont’d

For the Mars Pathfinder, this reset is OK and does not cause any real problem because there is no state to remember; it always sends just the current data.

For your trains software, there is state, namely the setting of all switches, the location of all trains, etc.

So a reset is not an acceptable solution to priority inversion.

How to Fix Priority Inversion

Use priority inheritance!

That is, cause a task *t* to temporarily inherit a higher priority from the higher priority task that depends on *t*.

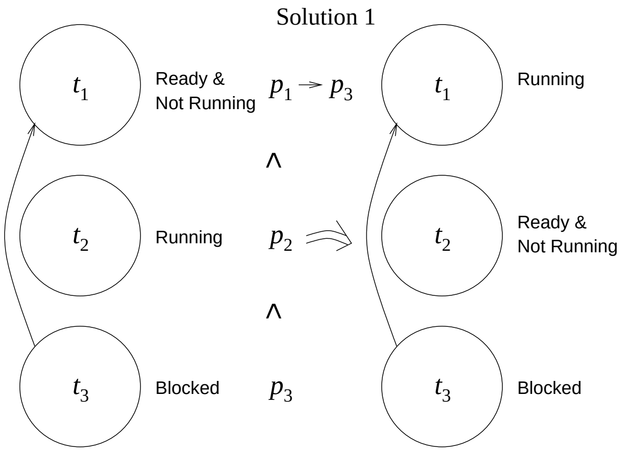
Solving Scenario 1

If tasks $t_1, \cdots, t_n$ are SEND_BLOCKED or REPLY_BLOCKED on t_0,

$$\text{actualPriority}(t_0) = \underset{0 \le i \le n}{\text{MAX}} \left(\text{assignedPriority}\ (t_i)\right)$$

I.e., promote t_0 to have the highest of the priorities of the tasks waiting on t_0. Then a medium priority task cannot preempt t_0.

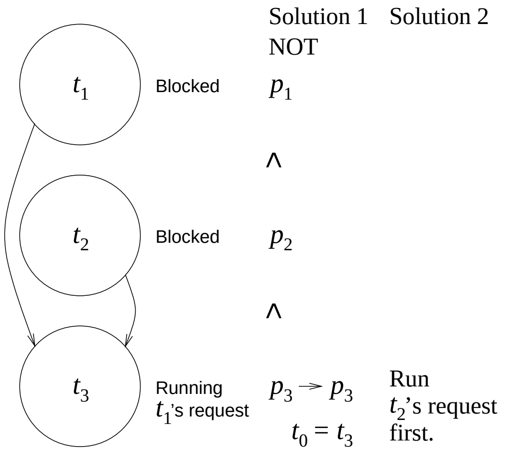
Solving Scenario 1, Cont’d



Solving problem 2

The next message received is from the highest priority SEND_BLOCKED task.

Solving Scenario 2, Cont’d



Implementation

Implementation of these solutions requires:

- order tasks by priority on any Send queue

OR

- multiple queues per task, one for each priority

Also for Solution 1, also REPLY_BLOCKED tasks must be tracked.

Deadlock!

Note that priority inheritance prevents priority inversion, but not *deadlock*

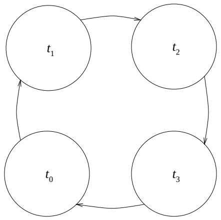
Deadlock is a cyclic resource dependency.

Among tasks $t_0, \ldots, t_{n-1}$, each task t_i holds a resource that is needed by $t_{(i+1) \bmod n}$ to proceed.

$\therefore$, None of $t_0, \ldots, t_{n-1}$ can ever run.

Cyclic Resource Dependency

A cyclic resource dependency is called also “a cyclic send pattern”.



If each T_i Sends before any T_i Receives, the tasks deadlock.

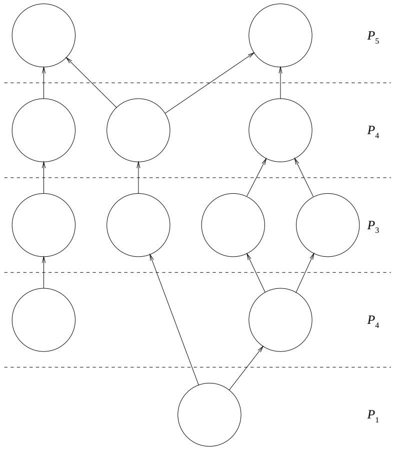
Deadlock, Cont’d

A cycle in the *steady-state* process diagram indicates a potential deadlock.

$\therefore$, in your applications, your steady-state diagram *must* be an acyclic graph, as is the graph at the beginning of this section of slides.

If the process diagram is acyclic, it can be written as a hierarchy.

Hierarchy



After Making the Hierarchy

Assign higher priorities to process that are higher in the graph.

This method assumes that processes are usually blocked:

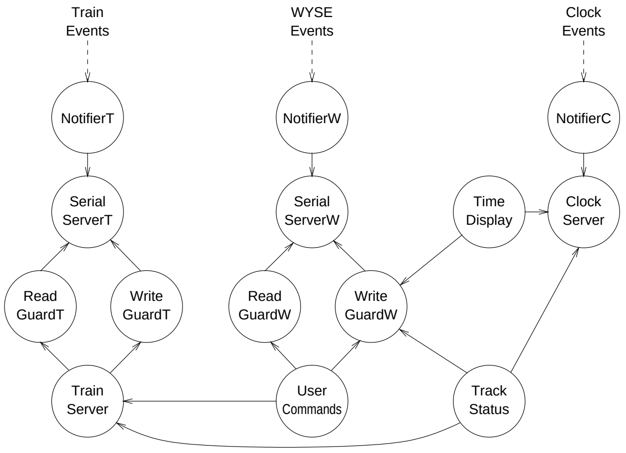
- Long-running tasks should have low priority.
- Interactive tasks should have high priority.

Hierarchy for Assignment 1

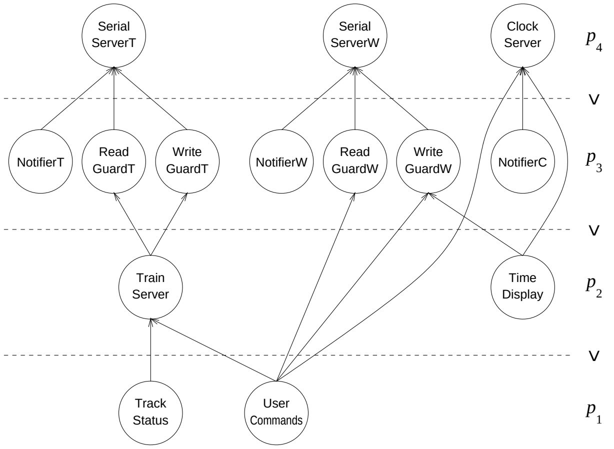
Let’s build the hierarchy for the suggested process structure for Assignment 1.

First, make each task name unique.

Hierarchy for Assignment 1, Cont’d



Hierarchy for Assignment 1, Cont’d



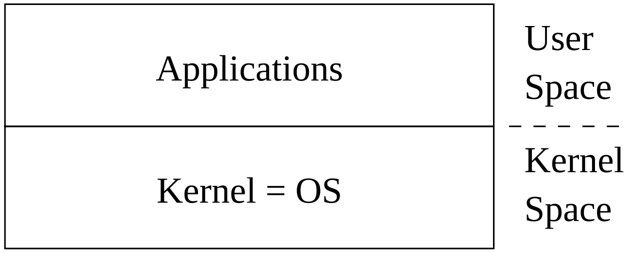
OS Design Principles

Two choices:

- Monolithic
- Microkernel

Monolithic

- Entire OS runs in kernel space.
- The OS is one big program.



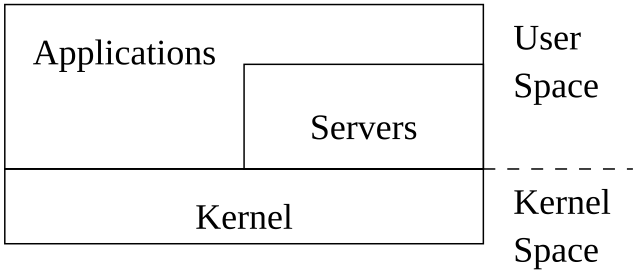
Monolithic, Cont’d

- The OS is easy to get wrong!
- If one OS module fails, the entire OS may go down.
- But, the OS is very efficient, once the bugs are worked out; less communication overhead

Microkernel

- The kernel, consisting of only memory management (GDT), IPC, scheduling, is small.
- Non-essential OS services are implemented as user-space programs, called *servers*, which include file systems, device drivers, and networking.

Microkernel, Cont’d



Microkernel, Cont’d

- If an OS service fails, it can be restarted without bringing down the kernel.
- Performance depends on
  - ○ fast IPC and
  - ○ fast context switching.
- Server development is easier than kernel development
- The OS is more secure, in the sense that less of the OS has access to all of memory.

Microkernel, Cont’d

Examples:

Mach, QNX, Minix, AmigaOS

First Microkernel OS, that happened also to be real time:

D.R. Cheriton, M.A. Malcom, L.S. Melen, G.R. Sager, “Thoth, A Portable Real-Time Operating System, *Communications of the ACM*, **22**:2, pp. 105–115, February 1975.

Application Code

The same design question can be applied to application code:

“One task or several?”

or

“Why not make the application a big loop polling the user’s input?”

Task Abstraction

A task

- is an independent autonomous agent and
- can be a basic application structuring unit.

Task Abstraction, Cont'd

An individual task is easy to understand; it

- is sequential,
- is deterministic,
- executes independently,
- has its own address space, and
- interacts with other tasks through *visible* interfaces.

The behavior of a server is specified by the messages it receives and the reply it generates in response to each received message.

E.g., the clockServer is specified by the semantics of its methods:

- Delay,
- GetTime, and
- DelayUntil.

Multiprocess Structuring

Multiprocess structuring can be done using stereotyped team structures and team members, …

Sort of process patterns ☺

Task Stereotypes

- Servers
- Workers
- Clients

Server Stereotypes

- proprietor — synchronizes accesses to a resource, e.g., serial server, for, e.g. video display
- distributor — acquires data, stores, and distributes them, e.g., state of track
- administrator — assigns and monitors work done by other tasks, e.g., managing a pool of worker tasks

Worker Stereotypes

- notifier — monitors events
- courier — moves data from server to server
- guard — controls accesses to a server

Client Stereotypes

- usually application specific — drives the high-level application logic.

Proprietor

According to Cheriton, a proprietor manages a resource and provides synchronous access, using mutual exclusion

```
Proprietor(){
    Initialize();
    while(1){
        (pid,request) ← Receive();
        Reply(pid,Service(pid,request));
    }
}
```

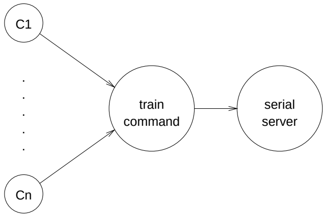
Proprietor, Cont'd

The details of Service distinguishes one proprietor from another.

Proprietor, Cont'd

The train command proprietor

- deals with one client at a time and
- may send messages to other servers.

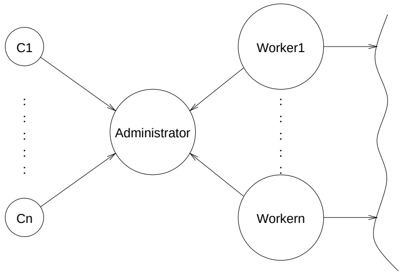


Adminstrator

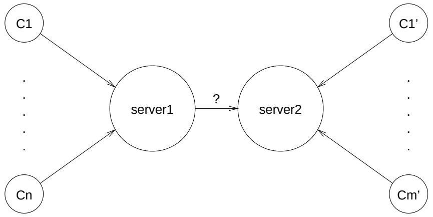
According to Morven Gentleman, an administrator

- is a generalized proprietor,
- may spawn workers, or agents, to handle requests,
- may prioritize requests, so that service is not always FIFO, and
- can use parameters in the clients' messages to determine which client's request to process next.

Adminstrator, Cont’d



Consider This Situation



Server 1 needs to send data to Server 2.

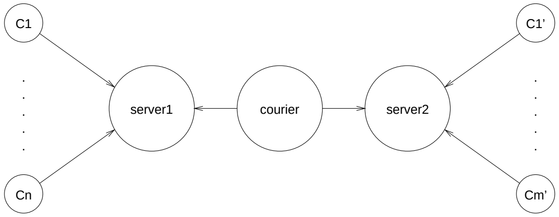
Situation, Cont’d

- If Server 1 sends to Server 2, then Server 1 is SEND_BLOCKED. ∴, Server 1 cannot receive from a client.
- Same for Server 2

In general, servers should not ever do Send.

Solution

Have a courier task.



Courier

A courier moves data from one specified server, named by pid0, to another specified server, named by pid1.

So it is a worker.

```
Courier(pid0,pid1){
    while(1){
        Send(pid0,msg1,msg0);
        Send(pid1,msg0,msg1);
    }
}
```

Generic Courier

CreateCourier(pid) — an OS service, and is optional.

This does Courier(MyPid(),pid).

Other Worker Stereotypes

Notifiers

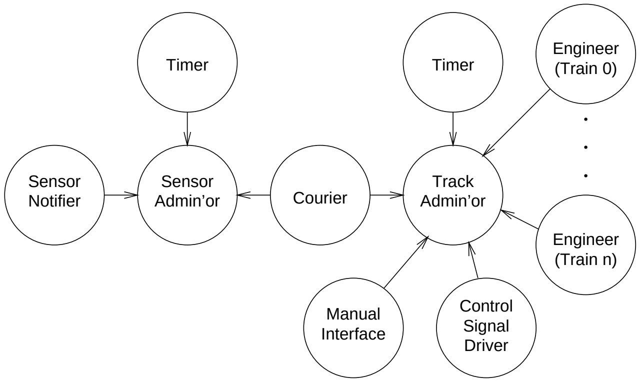
Guards

For more details, see W.M. Gentleman, “Message Passing Betwen Sequential Processors: the Reply Primitive and the Administrator Concept”, *Software Practice & Experience*, **11**, pp.436–466, 1981.

Suggest Train Application Structure

As suggested by Gentleman:

Train Application Structure



Structure, Cont’d

Verifying that this graph as no cycles is left as an exercise for the student!

What are the processes that have no outgoing arcs?

As a matter of fact, it has no cycles!

Two Administrators

- Track Administrator mangages current state of the track and the positions of the trains.
- Sensor Administrator summarizes and validates sensor information; it interprets each sensor hit as evidence of a train’s position, as spurious, or as indicating hardware failure.

Other Tasks

- Timer sends a message every *k* ticks.
- Engineer computes the next objective for one train, either move forward on completion of a subgoal or complete an alternative on failure.
- Control Signal Driver sends commands to the track.
- Manual Interface passes on user commands.

Multiple Administrators

- increases modularity and
- decreases wait time for time-critical clients, e.g. Notifiers.