Resource Locking Program Example

Consider a task that updates a table of temperatures:

```
r = readInterface();
temp = hf(r);
time = gettime();
lock(temptable->lock);
i = temptable->index++;
temptable->tempdata[i] = temp;
temptable->timedata[i] = time;
unlock(temptable->lock);
```

Between `lock` and `unlock` is the critical region.

Code fragment above locks temptable resource.

Table index is incremented and new values written.

Without exclusive access, two tasks might write same entry.

Resources and Blocking

Resource
Something a task needs that is shared with other tasks.

Resource Considered in this Set: Exclusive Access to Shared Data

Some tasks may need to write shared data.

Some tasks may need to read shared data.

Cannot allow one task to read data *partially* updated by another.

This Set:

- How programs specify that exclusive access needed.
- Implications for run time.
- Protocols to limit worst-case run time.

System Assumptions

Systems Discussed Here:

- Computation by tasks (not interrupt handlers).
- System task-preemptive and uses priority scheduling.
- Distinct priority levels unless otherwise noted.
  (That is, no two tasks have same priority.)

Interrupt handlers not considered because . . .

. . . cannot normally context switch between handlers.

Resource Locking

Resource Naming

Here, resources given names R1, R2, . . .

These will refer to memory that can be accessed by multiple tasks.

Locking

A task that has *locked* a resource has exclusive access.
(No other task is allowed to read or write it.)

Tasks lock a resource when they need to make changes . . .
. . . *unlocking* the resource after making the changes.

Critical Region

The part of a program that accesses a locked resource.

Locking in Programs

Resources locked with a `lock(RES)` call, unlocked with a `unlock(RES)` call.

(Details vary with language, synchronization package, etc.).

 18-5

Execution of `lock` and `unlock`

When a task calls `lock`:

If resource available, (not locked by another task) . . .
. . . `lock` returns immediately (task continues computing).

If resource unavailable, (is locked by another task) . . .
. . . task moved to **wait** state and some other task run.

Waiting task is said to be *blocked*.

When a task calls `unlock`:

OS move a tasks waiting for resource to ready list . . .
. . . and either returns to unlocking task . . .
. . . or switches to previously waiting task (depending on scheduling).

 18-6

Impact on Timing

Blocking Time [of a task]
Time waiting for resources (during `lock` call).

By no means a second-order effect.

Must be taken into account when estimating latency, etc.

Without *locking protocols* . . .
. . . low-priority tasks can have large effect on higher-priority tasks.

Problem reduced using *locking protocols*. Cases considered.

- None. (No special treatment for blocking.)
- Priority Inheritance.
- Priority Ceiling (two variations).

 18-7

Timing Without a Locking Protocol

To compute blocking time for a task:

Find a worst case execution in which:

. . . lower-priority tasks have locked all needed resources,

. . . the lower-priority tasks are at the beginning of their critical regions,

. . . and the lower-priority tasks are preempted by other tasks. (See example).

Priority Inversion

The worst case execution described above suffers *priority inversion* . . .
. . . because high priority tasks must wait for lower priority tasks to complete.

 18-8

Example: Timing Without a Locking Protocol & Priority Inversion

Consider:

Task Name	Priority	Arrival Time	Behavior
A	3	30	Computes for 10, locks r1, computes for 5, unlocks r1.
B	2	20	Computes for 100. (Doesn't use resources.)
C	1	0	Computes for 15, locks r1, computes for 10, unlocks r1, computes for 200.

Execution highlights:

C starts at 0, locks r1 at 15, and is preempted by B at $t = 20$..

B is preempted by A at $t = 30$.

A attempts to lock r1 at $t = 40$, since it's locked A goes to wait state.

A must wait for B to finish, *then* another 5 units for C to complete its critical region.

A waiting for B to finish is an example of priority inversion.

Example: Priority Inheritance

Consider the system below (same as the previous example).

Task Name	Priority	Arrival Time	Behavior
A	3	30	Computes for 10, locks r1, computes for 5, unlocks r1.
B	2	20	Computes for 100. (Doesn't use resources.)
C	1	0	Computes for 15, locks r1, computes for 10, unlocks r1, computes for 200.

Execution Highlights

The execution is the same as the previous example up to $t = 40$.

At $t = 40$, when A tries to lock r1, C's dynamic priority is set to 3 ...
... and so C runs instead of B, and A can resume in just 5 more time units.

Locking Protocols

Idea: Avoid priority inversion by adjusting priority of locking tasks.

Priority used above now called *static priority*.

Tasks now also have *dynamic priority*.

Initially, dynamic priority set to static priority, adjusted by locking protocol.

Two Protocols

- Priority Inheritance Protocol
  Dynamic priority based on *blocked* tasks.
- Ceiling Protocols (Two variations given)
  Dynamic priority based on resources.

Performance of Priority Inheritance

Blocking Time with Priority Inheritance

Blocking time for task X is sum of

for each distinct resource locked by X

largest critical region (CR) of lower priority task that accesses resource.

For example,

if X locks r1 and r2 ...

... and a lower priority task can lock r1 with a CR time of 5 ...

... and two lower priority tasks can lock r2 with CR times of 10 and 12 ...

... the worst case blocking time is 17.

Notice that run time of a task suffers ...
... if it accesses *many* resources ...
... that are also accessed by lower priority tasks ...
... because when the task starts all its resources may be locked.

Priority Inheritance

Implementation of Priority Inheritance

The dynamic priority of a task locking a resource ...
... is set to the maximum of:
... its own priority,
... and the priority of tasks blocked on the resource.

That is, a task in a critical region "inherits" the priority of waiting tasks.

Ceiling Protocols

Each resource has a *priority ceiling* . . .
. . . the maximum dynamic priority of a task that can access it.

(Access by a higher priority task would be a programming error.)

Two Ceiling Protocols: *Immediate* and *Original*.

Immediate Ceiling Protocol
Dynamic priority of locking task immediately assigned ceiling.

Original Ceiling Protocol
Ceiling prevents other tasks from locking resources (but not from running).

Both Protocols

Ensure that a task will never find more than one of its resources locked . . .
. . . thus limiting blocking time.

Immediate Ceiling Protocol

Immediate Ceiling Protocol Details

Each task assigned static priority.

Each resource assigned a priority ceiling.

Task's dynamic priority initially set to static priority.

When locking:

Error if task dynamic priority higher than resource ceiling.

Wait until resource available.

"Old" priority saved.

Dynamic priority set to resource's ceiling.

Unlocking

Old priority restored.

Original Ceiling Protocol

Original Ceiling Protocol Details

Each task assigned static priority.

Each resource assigned a priority ceiling.

Task's dynamic priority initially set to static priority.

System maintains a *global priority*:
. . . the highest priority ceiling of locked resources . . .
. . . or $-\infty$ if no resources locked.

Original Ceiling Protocol Details, continued

When locking an unlocked resource:

Error if task dynamic priority higher than resource ceiling.

Lock granted if dynamic priority higher than *modified* global priority . . .
. . . otherwise put on wait list.

Modified global priority for a task is the global priority determined without including resources locked by task (but including resources locked by other tasks).

When locking an already locked resource:

Error if task dynamic priority higher than resource ceiling.

Task that locked resource inherits current task's dynamic priority.

Current task put on wait list.

Original Ceiling Protocol Details, continued

Blocked task removed from wait list when:

resource available and task's dynamic priority higher than global priority.

Unlocking

Old priority restored.

Global priority adjusted.

Comparison of Immediate and Original

Immediate

Less complex.

Reduces time that resources locked.

Can enforce (in other words impose) access ordering.

That is, when a task locks multiple resources, their ceilings must form an increasing sequence.

Original

Avoid unnecessary delay of tasks that do not lock resources.

Common to Both Protocols

Worst case blocking time is single largest critical region . . .
. . . of any lower priority task that accesses resources . . .
. . . with ceilings at or above task's priority.