Real-Time Architectures
2003/2004

Scheduling (allocation) policies

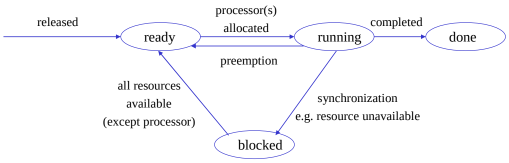
Johan Lukkien

18-04-2004

Contents

- Job life and preemption
- Policies
  - First come first served
  - Time sliced
  - Priority assignments
- Blocking
  - deadlock & priority inversion
  - concurrency control policies/protocols

The life-cycle of a job



- **Notes:**
  - processors clearly are special resources
    - needed by each job; they make a job proceed
  - while running, new jobs may be released
    - perhaps synchronization with those new ones is required

Derived terminology

- The set of resources: *Res*
- The set of ready jobs: *Ready*
  - (often said : ready *tasks*)
- The set of running jobs: *Running*
  - at most a single element for single processor systems
- The set of blocked jobs: *Blocked*
- Last three together: *Active*
- The required resources of a job: *Req*($\tau_{j,i}$)
- The allocated resources of a job: *Alloc*($\tau_{j,i}$)
  - note: a ready job is just waiting for the processor
- **NOTE**:
  - this is a dynamic picture
  - these sets don't have to be explicit in a mechanism

Resources and preemption

- A job using a resource usually generates associated state
  - registers etc. in a processor
  - variables inside an object
  - ... what about a cache?
  - ... are there state-less resources?
- Upon preemption
  - save the state
  - or destroy the state – roll-back
    - destroys the effort (work) in obtaining it
  - ....what about a network interface card?
  - associated penalty: *context-switch time*
- Otherwise (no preemption)
  - hold the resource for the time of operation
    - predictable, bounded
  - penalty for waiters: *incurred blocking time*

Contents

- Job life and preemption
- Policies
  - First (last) come first served
  - Time sliced
  - Priority assignments
- Blocking
  - deadlock & priority inversion
  - concurrency control policies/protocols

Scheduling policies

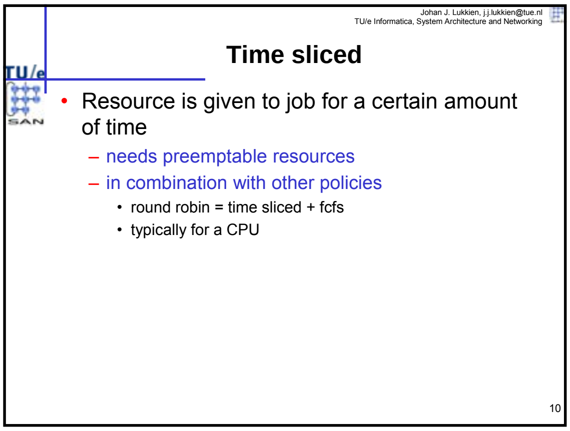
- The policy represents the strategy for allocating a resource to a job at *scheduling* points
  - informal algorithm, no mechanism yet
  - tells the decision based on the current state
    - represented by sets *Ready, Blocked, Res* and *Running* as well as available and required resources
  - mechanisms are rather different per resource
    - **processor**:
      - implicit (i.e., not visible in the program text) change of control at scheduling points
    - **passive resource**:
      - explicit locking and unlocking through e.g. semaphores
      - implicit locking and unlocking upon function entry and exit [monitors, e.g. Java]

Scheduling policies (cnt'd)

- Scheduling decisions, triggered by events
  - job release
  - job completion
  - job blocking
  - resource release
  - timers
- Policies depend on resource type and properties
  - different policies for e.g. memory allocation and processor
  - may result in conflicts

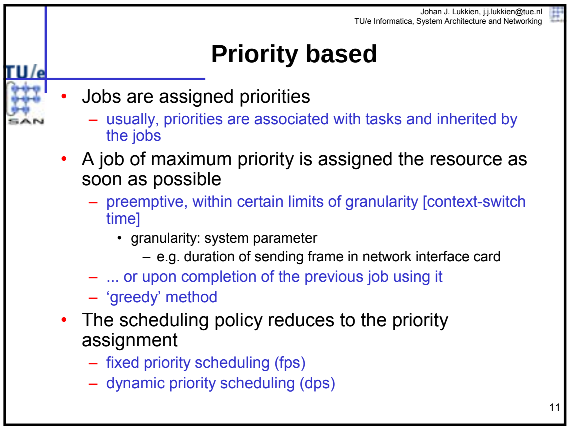
First (last) come first serve

- Resource assigned to jobs in (reverse) order of request arrival
  - resource held until job releases them
  - results in non preemptable jobs
    - only scheduling at job synchronization points criterion: select job with $est_{j,i}$ minimal (maximal)
- Applicable to all resource types
  - but typically for state-holding, non-preemptable resources
  - predictability requires upperbound to duration of any section using the resource



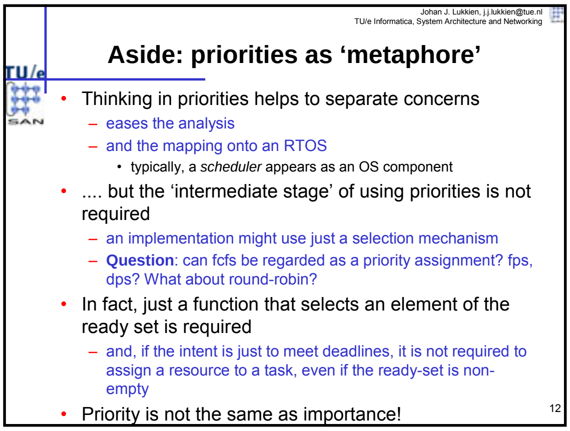
Time sliced

- Resource is given to job for a certain amount of time
  - needs preemptable resources
  - in combination with other policies
    - round robin = time sliced + fcfs
    - typically for a CPU



Priority based

- Jobs are assigned priorities
  - usually, priorities are associated with tasks and inherited by the jobs
- A job of maximum priority is assigned the resource as soon as possible
  - preemptive, within certain limits of granularity [context-switch time]
    - granularity: system parameter
      - e.g. duration of sending frame in network interface card
  - ... or upon completion of the previous job using it
  - ‘greedy’ method
- The scheduling policy reduces to the priority assignment
  - fixed priority scheduling (fps)
  - dynamic priority scheduling (dps)



Aside: priorities as ‘metaphore’

- Thinking in priorities helps to separate concerns
  - eases the analysis
  - and the mapping onto an RTOS
    - typically, a *scheduler* appears as an OS component
- .... but the ‘intermediate stage’ of using priorities is not required
  - an implementation might use just a selection mechanism
  - **Question**: can fcfs be regarded as a priority assignment? fps, dps? What about round-robin?
- In fact, just a function that selects an element of the ready set is required
  - and, if the intent is just to meet deadlines, it is not required to assign a resource to a task, even if the ready-set is non-empty
- Priority is not the same as importance!

Priority assignment policies

- Rate monotonic
  - $\rho_{j,i} > \rho_{k,l}$ iff $T_j < T_k$
- Deadline monotonic
  - $\rho_{j,i} > \rho_{k,l}$ iff $D_j < D_k$
- Shortest (longest) job next
  - $\rho_{j,i} > \rho_{k,l}$ iff $C_j < (>) C_k$
- Earliest deadline first
  - $\rho_{j,i} > \rho_{k,l}$ iff $dl_{j,i} < dl_{k,l}$
- Least Slack-Time first
  - $\rho_{j,i} > \rho_{k,l}$ iff $X_{j,i} < X_{k,l}$
- **NOTES:**
  - typical use: single processor scheduling

Non-priority driven

- Latest Release Time (‘reverse EDF’)
  - compute the schedule (not: priority assignment)
  - treat deadlines as release times and vice versa
    - determine largest deadline among the jobset
    - the job with that deadline is scheduled
    - going backwards, any contention is resolved by selecting the one with the largest (nearest-by) release time
- Result
  - possible idling with a non-empty readyset
  - use: admit background activity

Implementation requirements

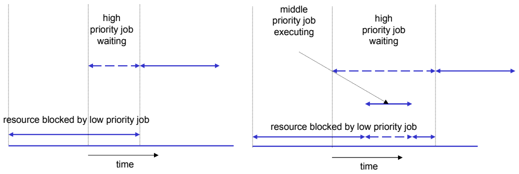
- It should be possible to base scheduling decisions taken in the running system on
  - limited, local dynamic information
  - static information
    - e.g. stored schedule, fixed priority
- Data structures for storing decision supporting information
  - limited in size
  - efficient
    - preference for *O(1)* operations
    - *O(***log** *n)* acceptable
    - avoid *O(n)* operations

Contents

- Job life and preemption
- Policies
  - First come first served
  - Time sliced
  - Priority assignments
- Blocking
  - deadlock & priority inversion
  - concurrency control policies/protocols

Priority inversion

- A low priority job obtains a resource; a high priority job waits on it
- A middle priority job pre-empts the low priority job
  - the high priority job now waits on the middle priority job
  - ... and executes effectively at the low priority



Priority inversion

- A pair of alternating middle priority jobs can block the high priority job indefinitely
  - *unbounded priority inversion:*
    - comparable to unfair synchronization (in OS)
    - example: Mars rover
- A (concurrency control) policy should
  - at least bound the inversion time
  - adhere to the job priorities as good as possible

Sources of blocking

- Direct blocking
  - another job holds the resource
- Push-through blocking (... preemption)
  - job *A* suffers push-through blocking if a lower priority job changes priority temporarily to a higher one than *A*
    - due to a specific concurrency control protocol (see later)
- Chained blocking (transitive blocking)
  - sequence of blockings
    - job *A* blocks on (resource *R0* held by) job *B*
    - job *B* blocks on (resource *R1* held by) job *C*
- Deadlock:
  - circular waiting
  - greedy consumers
  - blocking critical sections
- Avoidance blocking
  - blocking according to a strategy to avoid some of the above

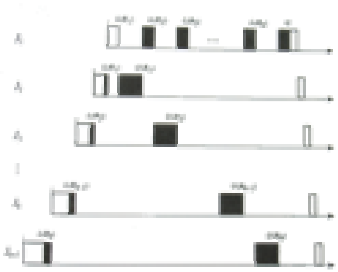
---

Avoiding deadlock

- Let critical sections terminate
  - in principle, no blocking operations in critical sections
- Use a fixed order in acquiring resources
  - *P(m);P(n);* .... in one job may deadlock with *P(n);P(m);*... in another job
- Avoid greediness in jobs
  - if a set of similar resources is needed, acquire them at once rather than one at a time
- Use the priority ceiling protocol
  - .... see later

- *In general: avoid cyclic waiting!*

Chained blocking



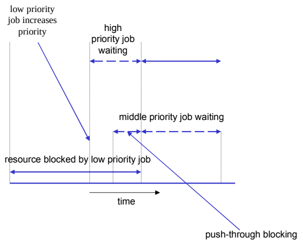
Effects of blocking

- Unpredictable or indefinite blocking times
  - unpredictable: due to priority adaptations or inversions
  - indefinite (deadlock): usually due to programming errors
- Unpredictable or unbounded delays
  - delay after preemption much longer than initially estimated
- … leading to adapted scheduling policies
  - “concurrency control protocols”

Priority inheritance protocol

- The priority of a job $\tau_{j,i}$ is *dynamically adjusted* to be the maximum of
  - the priority of any job that is blocked on the allocated resources of $\tau_{j,i}$
  - (... and its own priority)
- This adjustment is done *transitively*, i.e., if the priority of a waiter becomes adjusted then this adjustment is forwarded
- ...middle priority jobs will wait now.
- Resulting decision points for adjustment of priority of $\tau_{j,i}$:
  - any time *another job than* $\tau_{j,i}$ becomes blocked on a resource of $\tau_{j,i}$!

Priority inheritance



Behavior of PIP

- A job $\tau_{j,i}$ requiring a PIP-resource *r*
  - waits – w.r.t. this resource – for at most one lower priority job...
    - what are the possible waiting situations??
  - ... that, for this waiting time, executes with at least the priority of $\tau_{j,i}$
    - any additional blocking time on this resource has to come from *higher priority* jobs than $\tau_{j,i}$
- A job not requiring *r*
  - can incur blocking from lower priority jobs: push-through blocking
    - if it is of middle priority
    - but not more times than there are lower priority jobs (why?)
- Chained blocking is possible
  - however, limited – see exercise

Highest locker

- Determine the *ceiling* of a resource *r* by *beforehand* computing the maximum priority of any job that could use *r*
- While using *r,* a job $\tau_{j,i}$ is *dynamically* given the priority *ceiling(r)+1*
- During a critical section, the job may not suspend itself

- Meant for *implementing* critical sections
  - locking is implicit
- Limit case:
  - while using any resource, execute at maximum priority
    - non-preemptive
- Decision points:
  - whenever a resource is used
  - based on a value stored with the resource

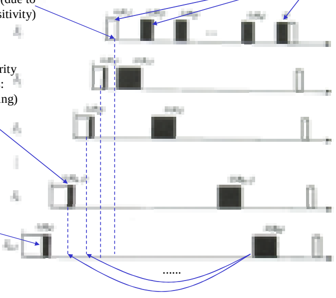
Behavior of highest locker

- Needs an analysis during development
  - needs to be *maintained* throughout that process
- No chained blocking
  - why?
- Blocking of all potential users starts when the resource is used by one of them
  - even when they don't need the resource
- As with priority inheritance:
  - need to wait at most once for each lower priority job
  - jobs may incur push-through blocking
- Best design practice
  - specify the used resources at the interface (cf. component-based software engineering)

Priority ceiling protocol

- Determine the *ceiling* of each resource *r* by *beforehand* computing the maximum priority of any job that could use *r*
- Each job $\tau_{j,i}$ knows $sc_{j,i}$, its *system ceiling*, the maximum ceiling of any resource used by another job
  - $sc_{j,i} = (\mathbf{max}\ r : r \in Alloc(\tau)$ for $\tau_{j,i} \neq \tau \in Active : ceiling(r))$
  - $r^*_{j,i} : ceiling\ (r^*_{j,i}) = sc_{j,i}$
- Upon resource request of *r* by job $\tau_{j,i}$
  - block and wait, if
    - *r* in use
    - or $sc_{j,i} \geq \rho_{j,i}$
  - when blocking, use priority inheritance
    - in the first case the job using *r* inherits the priority of $\tau_{j,i}$
    - the second case the job using $r^*_{j,i}$ inherits priority of $\tau_{j,i}$

Priority ceiling



J_{k+1} changes priority to *1* (due to transitivity)

all resources are available after completion of J_{k+1}

J_{k+1} changes priority to *k*; J_k blocks: avoidance blocking)

For jobs different from J_{k+1} the system ceiling is 1

……

Behavior of priority ceiling

- As in highest locker: needs an analysis during development
  - needs to be *maintained* throughout that process
  - similarly: should be visible at interfaces
- Implementation possible that combines a “job ready queue” with semaphore administration
- No chained blocking
- Avoids deadlock
- Waits at most one critical section for lower priority jobs (!)

Exercises

- For a given job *J*
  - assume there are *n* lower priority jobs
  - and that *J* needs *m* resources

  What is the maximum number of job executions *J* is blocked in case of the
  - priority inheritance protocol
  - highest locker protocol
  - priority ceiling protocol
- Explain (prove) that in the highest locker protocol no chained blocking is possible.

Exercises

- Is deadlock and example of (cyclic) chained blocking? What is the difference?

Exercise

- Consider a task-set *Z* with the following characteristics. Assume $\varphi_j = 0$.

task	T_j	C_j
τ_1	8	1
τ_2	10	4
τ_3	12	3

(a) For $D_j = T_j$, draw a schedule for:

- Rate monotonic (RM);
- Shortest (longest) job next (SJN and LJN);
- Least Slack-Time first (LSF).
- Earliest deadline first (EDF).

Exercise

b) For $D_1 = T_1$, $D_2 = T_2$, and $D_3 = 7$, draw a schedule for:

- Deadline monotonic (DM);

c) From the six fixed-priority assignments possible for Z, five have been applied above. Identify and construct the lacking one, and conceive a mnemonic name for it.