Precedence Graphs Revisited

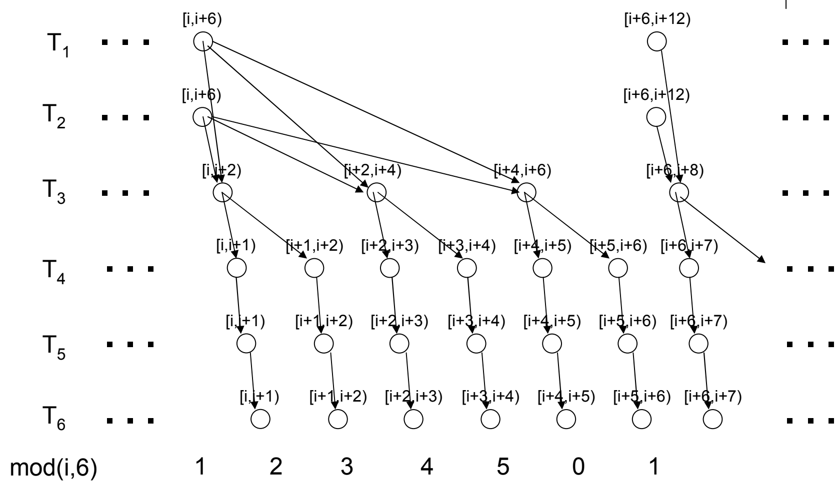
Flight controller for a helicopter; every 1/180th of a second

- Validate sensor data and select data source; in the presence of failures, reconfigure the system
- Do the following 30-Hz avionics tasks, each once every 6 cycles:
  - Keyboard input and mode selection
  - Data normalization and coordinate transformation
  - Tracking reference update
- Do the following 30-Hz computations, each once every 6 cycles
  - Control laws of the outer pitch-control loop
  - Control laws of the outer roll-control loop
  - Control laws of the outer yaw- and collective-control loop
- Do each of the following 90-Hz computations once every 2 cycles, using outputs produced by the 30-Hz computations
  - Control laws of the inner pitch-control loop
  - Control laws of the inner roll- and collective-control loop
- Compute the control laws of the inner yaw-control loop, using outputs from the 90-Hz computations
- Output commands
- Carry out built-in-test
- Wait until the beginning of the next cycle

Task/Job Definitions

- $J_{1,i}$: keyboard input and mode selection; data normalization and coordinate transformation; tracking reference update
- $J_{2,i}$: outer pitch control-law computation; outer roll control-law computation; outer yaw and collective control-law computation
- $J_{3,i}$: inner pitch control-law computation; inner roll and collective control-law computation
- $J_{4,i}$: inner yaw control-law computation
- $J_{5,i}$: output actuator commands
- $J_{6,i}$: carry out built-in test
- Time t_i is represented overleaf by i, where t_i = i* 1/180 second

Corresponding Precedence Graph (one major cycle)



Commonly-used Approaches to RT Scheduling

- Clock-driven
  - Primarily used for systems in which properties of all tasks/jobs are known at design time, such that offline scheduling techniques can be used
- Weighted round-robin
  - Primarily used for scheduling real-time traffic in high-speed, switched networks
- Priority-driven
  - Primarily used for RT systems with a mix of time-based and event-based activities

Clock-Driven Approach

- Decisions about what jobs execute at what times are made at specific time instants; these instants are chosen *a priori* before the system begins execution
- All parameters of hard RT jobs are fixed and known
- A schedule of the jobs is computed off-line and is stored for use at run-time; as a result, scheduling overhead at run-time can be minimized

Clock-driven Approach

- Frequently, make scheduling decisions at regularly spaced time instants – e.g. every 1/180th second in our avionics example
- Real implementations depend upon a hardware timer that can be set to interrupt at regular intervals
- When the system is initialized, the scheduler selects and schedules the job(s) that will execute until the next scheduling decision time; it then blocks itself waiting for the next timer interrupt
- When the timer expires, the scheduler awakes and repeats these actions

Weighted Round-robin Approach

- From OS3, you know that the round-robin approach is commonly used for scheduling time-shared applications
- Every job joins a FIFO queue when it is ready for execution; when the scheduler runs, it schedules the job at the head of the queue to execute for at most one time slice (sometimes called a quantum – typically o(tens of ms))
- If the job has not completed by the end of its quantum, it is preempted and placed at the end of the queue
- When there are N ready jobs in the queue, each job gets one slice every N time slices (N time slices is called a round)
- Weighted round robin – each job i is assigned a weight w_i; this job will receive w_i time slices every round, and a round is $\Sigma_i w_i$, for i = 1..N; regular round robin is weighted round robin where all weights are 1

Weighted Round-robin Approach

- By giving each job a fraction of the processor, a round-robin scheduler delays the completion of every job
- If it is used to schedule precedence-constrained jobs, the response time of a chain of jobs can be unduly large
- If a successor job can incrementally consume output from a predecessor (e.g. UNIX pipes), then this is a reasonable approach, since a job and its successors can execute concurrently in a pipelined fashion
- In high-speed switching networks
  - Message transmission is carried out in a pipeline fashion
  - A downstream switch can begin to xmit an earlier portion of a message as soon as it receives that portion without having to wait for the arrival of the later portion
  - WRR does not require a sorted priority queue, only a RR queue; for ultra high speed networks, priority queues with the required speed are very expensive

Priority-driven Approach

- Priority-driven algorithms NEVER intentionally leave any resource idle.
- Scheduling decisions are made when events such as releases and job completions occur; hence, such algorithms are ***event-driven***
- Also called greedy scheduling (makes locally optimal decisions), list scheduling and work-conserving scheduling
- Locally optimal scheduling decisions are often NOT globally optimal

Priority-driven Approach

- Most scheduling algorithms used in non real-time systems are priority-driven
  - First-In-First-Out
  - Last-In-First-Out

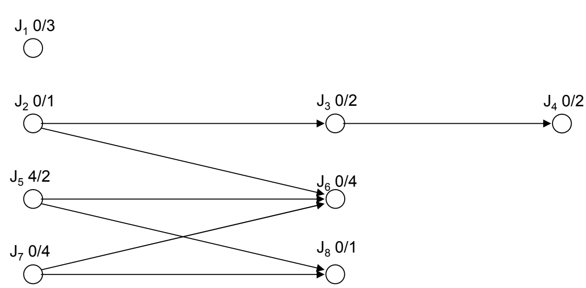
  } Based upon release times
  - Shortest-Execution-Time-First
  - Longest-Execution-Time-First

  } Based upon execution times

Priority-driven Approach

- Consider the following example:
  - Jobs J_1 … J_8, where J_i had higher priority than J_k if $i < k$
  - Jobs are scheduled on two processors P_1 and P_2
  - Jobs communicate via shared memory, so comms costs are negligible
  - The schedulers keep one common priority queue of ready jobs
  - All jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or a job completes

Priority-driven Approach



Priority-driven Approach

Time	Ready to run	P_1	P_2
0	1, 2, 7	1	2
1	1, 3, 7	1	3
3	4, 7	4	7
4	4, 5, 7	4	5
5	5, 7	7	5
6	7	7	-
8	6, 8	6	8
9	8	-	8
12	-	-	-

Priority-driven Approach

Assume jobs are non-preemptable:

Time	Ready to run	P_1	P_2
0	1, 2, 7	1	2
1	1, 3, 7	1	3
3	4, 7	4	7
4	4, 5, 7	4	7
5	5, 7	5	7
7	6, 8	6	8
8	6	6	-
11	-	-	-

Priority-driven Approach

- Dynamic vs Static Systems
  - If jobs are scheduled on multiple processors, and a job can be dispatched to any of the processors, the system is ***dynamic***
  - A job migrates if it starts execution on one processor and is resumed on a different processor
  - If jobs are partitioned into subsystems, and each subsystem is bound statically to a processor, we have a ***static*** system.
  - Expect static systems to have inferior performance (in term of the makespan of the jobs) relative to dynamic systems

Priority-driven Approach

- Sometimes, the release time of a job may be later than that of its successors, and its deadline may be earlier than that of its predecessors
- Effective release time
  - If a job has no predecessors, its effective release time is its release time
  - If it has predecessors, its effective release time is the maximum of its release time and the effective release times of its predecessors
- Effective deadline
  - If a job has no successors, its effective deadline is its deadline
  - It if has successors, its effective deadline is the minimum of its deadline and the effective deadline of its successors
- Scheduling is then based upon the effective values

Priority-driven Approach

- Priority assignment based upon deadlines
  - Earliest deadline first (EDF)
    - This algorithm is optimal as long as preemption is allowed and jobs do not contend for resources
  - Least Slack Time first (LST)
    - At any time t, the slack of a job with deadline d is d-t minus the time required to complete the remaining portion of the job
    - This algorithm is also optimal under the same conditions as EDF
  - Neither algorithm is optimal if jobs are non-preemptable or if there is more than one processor

Priority-driven Approach

- Behaviour under load
  - Clairvoyant scheduler – an imaginary algorithm that knows all future release times for all jobs
  - A system is ***overloaded*** if even a clairvoyant scheduler is unable to come up with a feasible schedule
  - In an overload situation, some jobs must be discarded (***shed***) in order to allow other jobs to complete in time
  - During overload, measure the performance of an algorithm by the amount of work the scheduler can feasibly schedule; the larger this amount, the better the algorithm
  - Value of a job = its execution time if it completes by its deadline, 0 otherwise
  - Value of a schedule = sum of the value of all jobs
  - Optimal algorithm if it always produces a schedule of maximum possible value for every finite set of jobs
  - For on-line scheduling, it is imperative to keep the system from becoming overloaded using some overload management or load shedding algorithms