# SYSC 5701

# Operating System Methods for Real-Time Applications

## Clock-Driven Scheduling

*Winter 2014*

RECALL

# Common Approaches For Real-Time Scheduling ( Liu Ch. 4 )

- **Clock-Driven (Time-Driven) :** scheduling decision points are specified *a priori* (static)
- **Weighted Round-Robin :** weighted jobs join a FIFO queue – weight determines amount of processor time allocated to the job ☹
- **Priority-Driven (Event-Driven)** : scheduling decisions are made as events occur (dynamic)
  - schedule ready job with highest priority

# Clock-Driven Scheduling

- job parameters are known *a priori*
- job schedule precomputed off-line and stored as a table for use at run-time

  → **table-driven scheduler**
- scheduling decision times in clock-driven system is defined *a priori*;
  - scheduler periodically wakes up and generates next portion of the schedule (from the table)

# Clock-Driven Scheduling

- Applicable when system is deterministic
  - only a few aperiodic and sporadic jobs
- Some assumptions
  - N periodic tasks in the system
  - task parameters known a priori
  - each job is ready for execution as soon as it is released

# Simplifying Assumptions

- Each task denoted by the tuple

  $( p_i, e_i, D_i )$
- Sometimes only the period and execution time is provided
  - relative deadline = period
  - **critical instant** at time = 0 !
  - denote tasks as pair $\mathbf{(p_i, e_i)}$

all tasks have a job ready at time 0

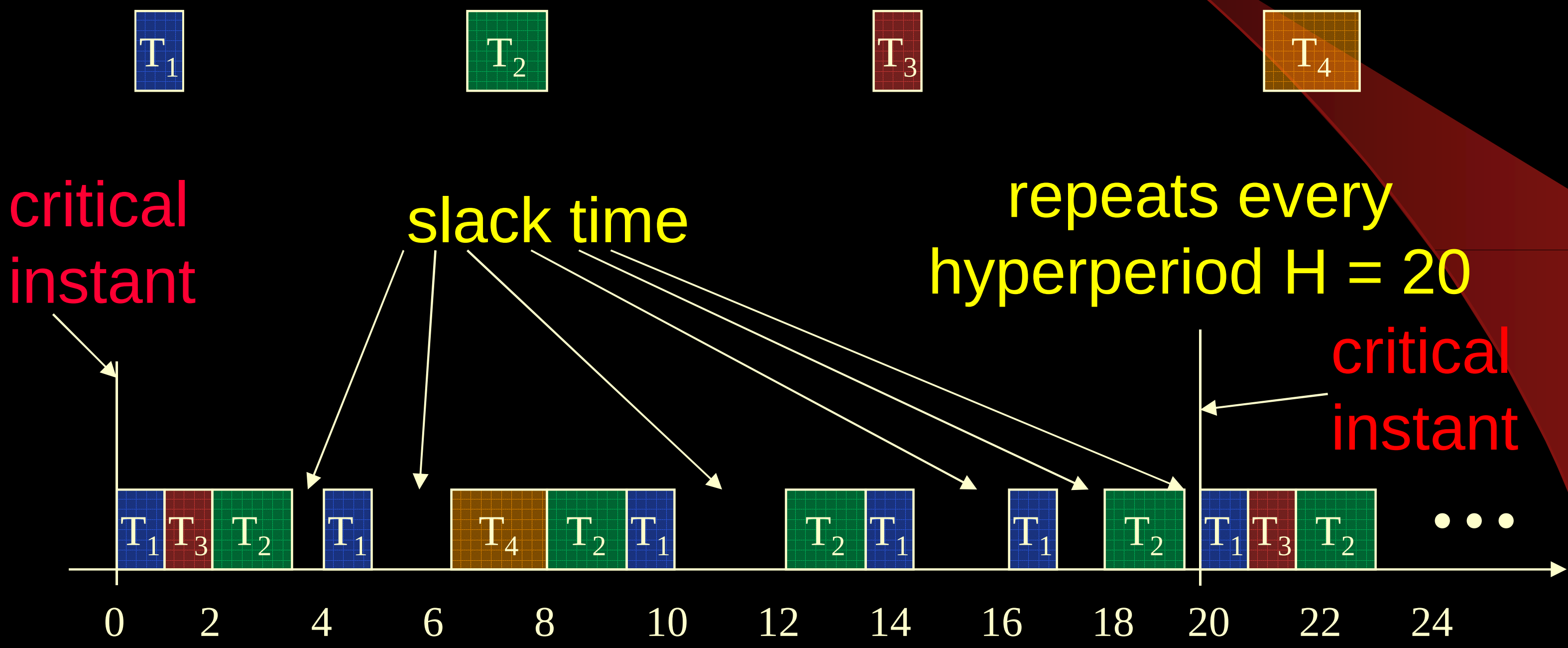
# How to Schedule?

- supported by **hardware timer**
- at run-time the scheduler dispatches jobs according to the preconceived schedule designed off-line
- the problem then becomes, how to design this periodic static schedule or **cyclic schedule**

# Example

- consider the following tasks and schedule:

$\{T_1 = (4,1)\ ,\ T_2 = (5,1.8),\ T_3 = (20,1)\ ,\ T_4 = (20,2)\}$




schedule was designed arbitrarily!

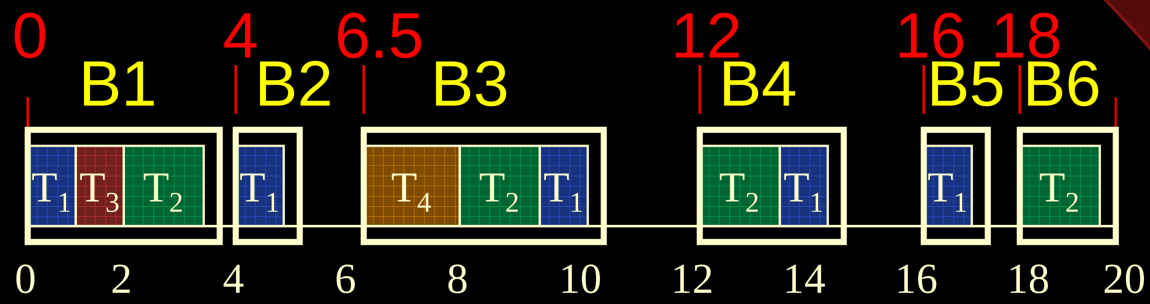


Carleton UNIVERSITY

# Simple Table Driven Scheduler Implementation for Example:

- organize “blocks of activities” in hyperperiod

burst start times:




| B1: | B2: | B3: | |
|---|---|---|---|
| call T1 | call T1 | call T4 | |
| call T3 | | call T2 | etc. |
| call T2 | | call T1 | |

# Organize Blocks in HTable

| Block | Relative StartTime |
| --- | --- |
| B1 | 4 |
| B2 | 2.5 |
| B3 | 5.5 |
| B4 | 4 |
| B5 | 2 |
| B6 | 2 |

← Relative time until start of next burst

# Static Clock-Driven Scheduler based on HTable

i = 0 ;

<set timer to expire at time HTable[i].StartTime>

call HTable[i].Block;

**timer ISR**

cyclic repetition!

i = i+1 MOD #of bursts;

<set timer to expire at time HTable[i].StartTime >

call HTable[i].Block;

# Analysis of Example

- arbitrary schedule
- could # of blocks be reduced?
- could # of blocks increase?
  - worst case = one task per time interrupt?

- is there a more systematic approach?

# Frame Scheduling

NB: static (off-line) scheduling!

- partition hyperperiod H into equal-sized frames
- constant frame length f = frame size
    - H is an integral multiple of f
- scheduling decision for a frame made at the start of the frame
    – no preemption within frame

# Frame Monitoring

- scheduler must be designed to ensure that at start of each frame:

1. jobs scheduled for execution in frame have been released and are ready
2. overrun does not occur
   – i.e. jobs in previous frames completed
3. jobs in the frame will meet their deadlines if completed by end of frame

# Frame Size Constraints

- every job must be able to start and complete within a frame:

$$f \geq \max(e_i)$$

- for at least one task Ti: $\lfloor p_i / f \rfloor - p_i / f = 0$

⌊ floor function ⌋ (round down, integer result)

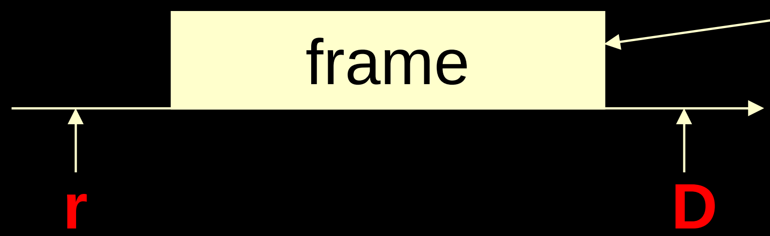
Why?

So frame divides evenly into hyperperiod.

# Frame Size Constraints (2)

- to ensure that every job completes by its deadline:  want f small enough that there is at least one frame between the release time and deadline of each job

ensures that job has a frame in which to execute




- Liu concludes constraint met when:

$$2f - \gcd(p_i, f) \leq D_i$$

**gcd** = greatest common divisor

# Cyclic Schedule Creation for Previous Example

$T = \{(4, 1), (5, 1.8), (20, 1), (20, 2)\}$

- Constraints on possible values of $f$

  $f \geq \max(1, 1.8, 1, 2) \geq 2$

  $f$ = a divisor of one $p_i$

  → one of 1, 2, 4, 5, 10, 20

  satisfy first constraint

  $2f - \gcd(p_i, f) \leq D_i$ ????

# Determining f

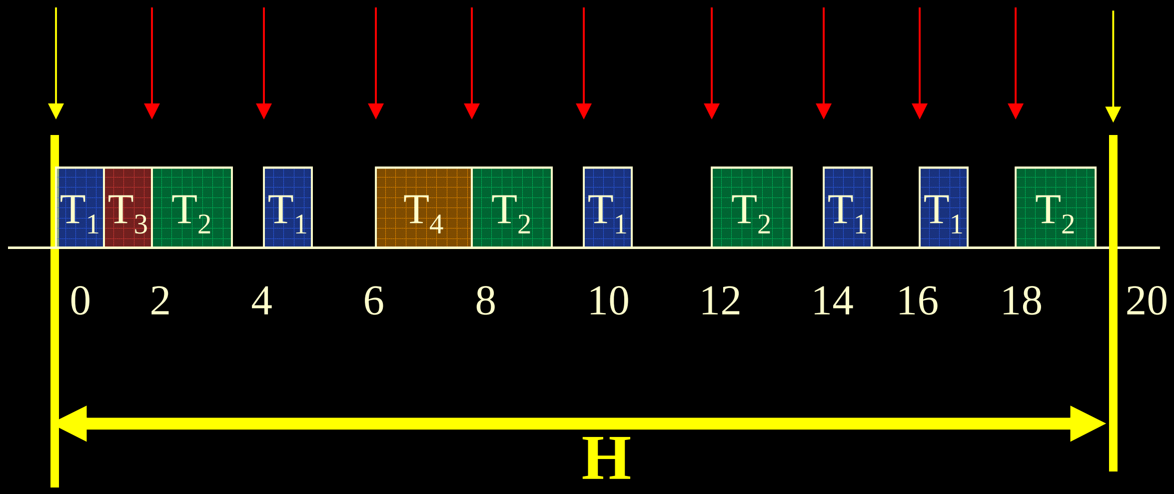
consider $2f - \gcd(p_i, f) \le D_i$ for 2, 4, 5, 10, 20

| pi | Di | 2f –gcd(pi, f=2 | f=4 | f=5 | f=10 | f=20 ) |
|---|---|---|---|---|---|---|
| 4 | 4 | 4-2 | 8-4 | 10-1 | 20-2 | 40-2 |
| 5 | 5 | 4-1 | 8-1 | 10-5 | 20-5 | 40-2 |
| 20 | 20 | 4-2 | 8-4 | 10-5 | 20-10 | 40-2 |

- therefore, f = 2 only case to satisfy!

# Cyclic Schedule with f = 2

- possible schedule :



# Problems with Frame Constraints?

- what if tasks won't all satisfy constraints?
- e.g. can't meet both:
  - minimum f to ensure a frame between release and deadline, and
  - f greater than execution time
- can't ensure that a job will be able to complete in one frame! ???

# Job Slices

- solution: partition jobs of a task into slices with smaller execution times

| $e_i$ | $e_{i1}$ | $e_{i2}$ |
|---|---|---|

execution time before

execution time after

- schedule slices in different frames
  - planned preemption!

# Design Decisions:

1. choose frame size f
2. partition jobs into slices
3. places slices in frames

- choices are not independent !
  - algorithm for choices in Liu 5.8

# Cyclic Executives

- modify clock-driven scheduler to make scheduling decisions on frame boundaries
  - don't need to adjust timer
  - job slices are organized into blocks
- use slack time to execute aperiodic/sporadic jobs (NB. dynamic, not static decisions!!)
  - special "servers" ?
  - run in background ?

# Overrun

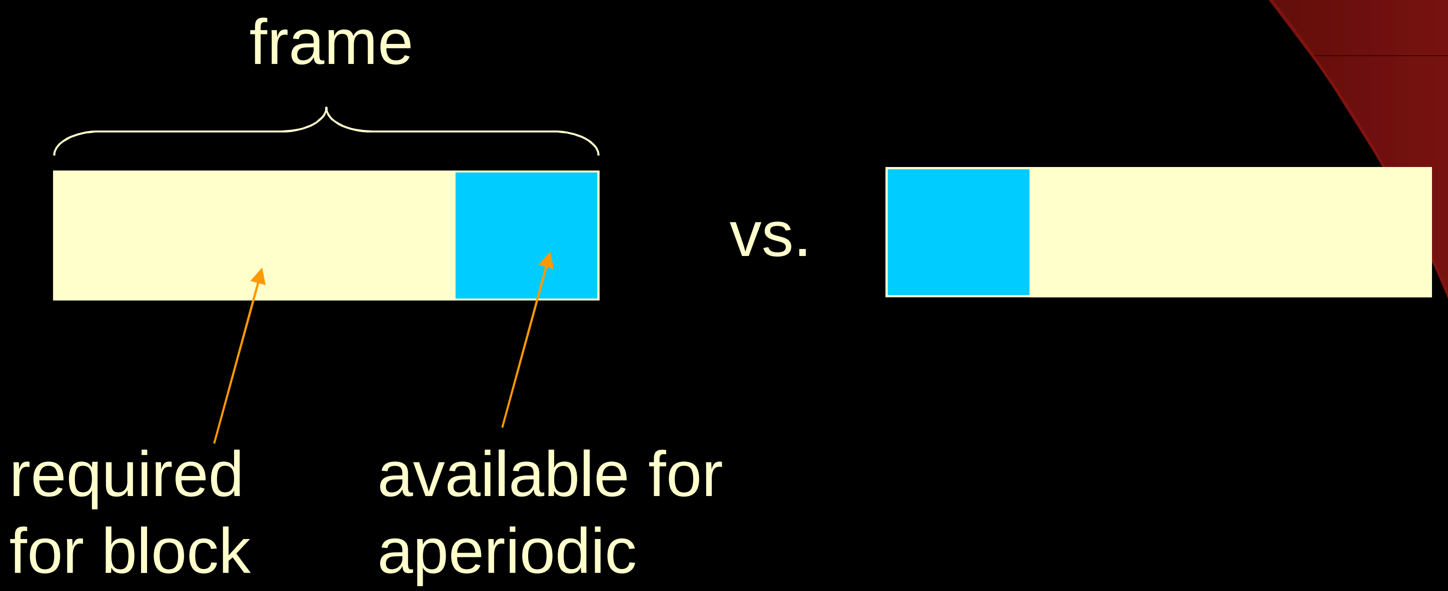
- if current block not completed by time next block starts → frame overrun !
- options:
  - abort the offending block
  - let block complete in background
  - finish the job and force others to be late
- exception handling! (gnarly)

# Improving Response of Aperiodic Jobs

- can use knowledge of deadlines to advantage!
- all slices in a block must complete within their frame
  - – no advantage to completing earlier vs. later in the frame
- instead of allocating slack at end of frame, could use it at beginning!

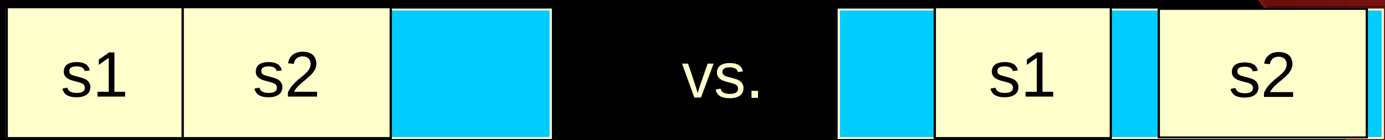
# Slack Stealing

- execute aperiodic jobs ahead of periodic jobs in a frame whenever possible



# Slack Stealing Further

- scheduler can allow aperiodic jobs to execute whenever there is slack in a frame
- could interleave between slices in a block




- increases aperiodic throughput

  → increases (management) overhead !

# Sporadic Jobs

- hard deadlines!
- assume minimum release, max execution and deadline times are known
- when sporadic job released – perform an **acceptance test**:
  - if jobs already scheduled + new job are **feasible** → then **admit** the job

# Sporadic Job Deadline

- sporadic job can use any slack available in any frame prior to its deadline
- if enough slack exists to meet deadline, then **admit** and **schedule** the job
- if insufficient slack – reject the job immediately
- if more than one sporadic job waiting – order them earliest deadline first

# Implementation

- sporadic job queue → EDF ordering
- in each frame:

1. execute the periodic block first
2. then dynamically accept (or reject) from sporadic job queue
3. then allow aperiodic jobs

- Liu text has more details (5.6.3)

# Mode Changes

- changes in operational mode can impact schedule
- mode change → "reconfigure" system
  - possibly different set of jobs
  - possibly different job parameters
  - may need initialization phase to delete "old jobs" and initialize "new jobs"

# Mode Changes (con't)

- change scheduling table for periodic jobs
- how to handle outstanding sporadic jobs from "old" mode?
  - must still meet their deadlines (?)
  - may not be possible due to reduced amounts of slack available (?)
  - requires careful handling (gnarly)

# Summary of Cyclic Executive

"loop forever" :

- wake up and execute at *tf* intervals (frame boundaries)
- retrieve the data structure which defines a frame
- wake up the periodic task server
- service the sporadic job queue
- service the aperiodic job queue
- perform general maintenance
  – manage slack time, perform error checking

# Pros of Clock-Driven Scheduling

advantages of clock-driven scheduling:

- simple to understand
- the validation problem is very easy (deterministic)
- precedence and dependency can be dealt with off-line by choice of the schedule

# Cons of Clock-Driven Scheduling

<u>disadvantages</u> of clock-driven scheduling:

- not well suited for applications with varying temporal & resource requirements
  - where exact nature of the workload model is not known a priori
- not always easy to design, usually <u>hard to change</u> !
- sophisticated approaches → overheads