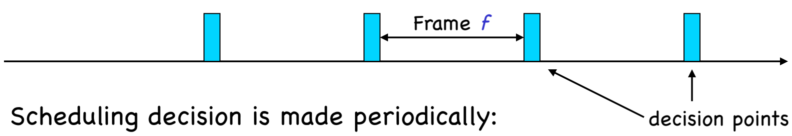
Time-Driven Scheduling

- Cyclic Schedules:
  - Frames and Major Cycles
  - Constraints on Frame Sizes
- The Cyclic Executive
- Aperiodic Workload
  - Slack Stealing
- (strictly) Sporadic Workload
  - Admission Tests
  - Priority Scheduling of (strictly) Sporadic Workload
- Static Scheduling of Workload in Frames
- Discussion
- Cyclic Executive in Practice

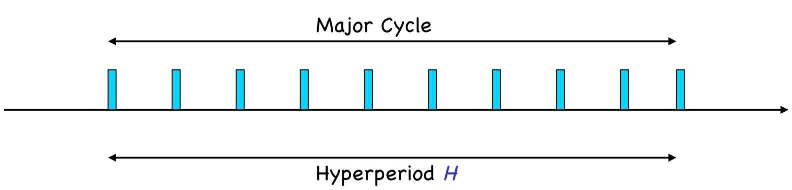
Cyclic Schedules: General Structure

- Scheduling decision is made periodically:



- Scheduling decision is made periodically:
  - choose which job to execute
  - perform monitoring and enforcement operations

- **Major Cycle**: Frames in a hyperperiod.



Frame Size Constraints

- Frames must be sufficiently long so that every job can start and complete within a single frame:

$$(1) \qquad f \geq \max(e_i)$$

- The hyperperiod must have an integer number of frames:

$$(2) \qquad f | H \qquad (f \text{ "divides" } H)$$

- For monitoring purposes, frames must be sufficiently small that between release time and deadline of every job there is at least one frame:

$$2f - (t' - t) \leq D_i$$
$$t' - t \geq \gcd(p_i, f)$$
$$(3) \qquad 2f - \gcd(p_i, f) \leq D_i$$

Frame Sizes: Example

- Task set:

	p_i	e_i	D_i
T_1 =	15,	1,	14
T_2 =	20,	2,	26
T_3 =	22,	3,	22

$H = 660$

(1) $\forall i : f \geq e_i \quad \Rightarrow f \geq 3$

(2) $f | H \quad \Rightarrow f = 2,3,4,5,6,10,..$

(3) $\forall i : 2f - \gcd(pi, f) \leq Di \quad \Rightarrow f = 2,3,4,5,6$

$\Rightarrow$ possible values for f : 3,4,5,6

Slicing and Scheduling Blocks

- Slicing

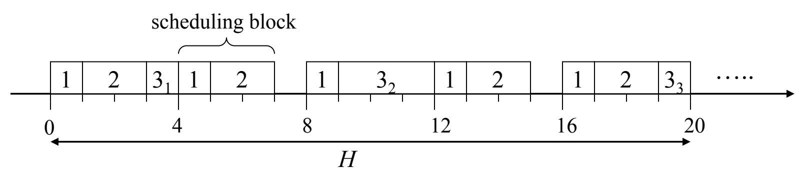
	p_i	e_i	D_i
T_1 =	(4,	1,	4)
T_2 =	(5,	2,	5)
T_3 =	(20,	5,	20)

(1) $\Rightarrow f \geq 5$
(3) $\Rightarrow f \leq 4$ } ?!

slice T_3

T_1 =	(4,	1,	4)
T_2 =	(5,	2,	5)
T_{31} =	(20,	1,	20)
T_{32} =	(20,	3,	20)
T_{33} =	(20,	1,	20)

(1) $\Rightarrow f \geq 3$
(3) $\Rightarrow f \leq 4$ } $f = 4$



© R. Bettati

Cyclic Executive

Input: Stored schedule: L(k) for k = 0,1,…,F-1;
Aperiodic job queue.

```
TASK CYCLIC_EXECUTIVE:
  t = 0; /* current time */        k = 0; /* current frame */
  CurrentBlock := empty;
  BEGIN LOOP
    IF <any slice in CurrentBlock is not completed> take action;
    CurrentBlock := L(k);
    k := k+1 mod F; t := t+1;
    set timer to expire at time tF;
    IF <any slice in CurrentBlock is not released> take action;
    wake up periodic task server to handle slices in CurrentBlock;
    sleep until periodic task server completes or timer expires;
    IF <timer expired> CONTINUE;
    WHILE <the aperiodic job queue is not empty>
      wake up the first job in the queue;
      sleep until the aperiodic job completes;
      remove the just completed job from the queue;
    END WHILE;
    sleep until next clock interrupt;
  END LOOP;
END CYCLIC_EXECUTIVE;
```

© R. Bettati

What About Aperiodic Jobs?

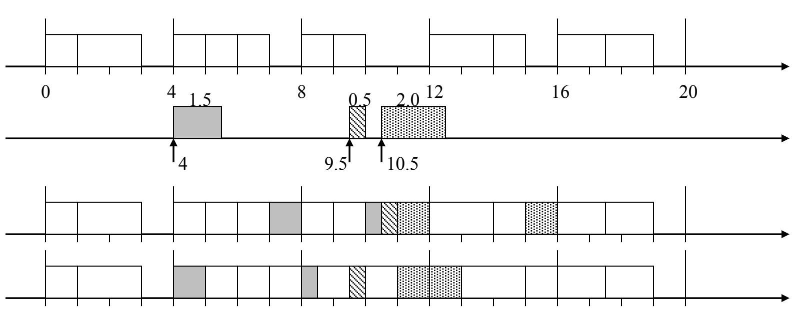
- Typically:
  - Scheduled in the background.
  - Their execution may be delayed.
- But:
  - Aperiodic jobs are typically results of external events.
- Therefore:
  - The sooner the completion time, the more responsive the system
  - Minimizing response time of aperiodic jobs becomes a design issue.
- Approach:
  - Execute aperiodic jobs ahead of periodic jobs whenever possible.
  - This is called **Slack Stealing**.

Slack Stealing (Lehoczky *et al.*, RTSS'87)

x_k Amount of time allocated to slices executed during frame F_k.

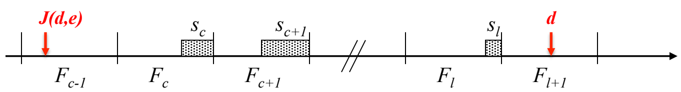
s_k **Slack** during frame F_k: $s_k := f - x_k$.

- The cyclic executive can execute aperiodic jobs for s_k amount of time without causing jobs to miss deadlines.
- Example:



(strictly) Sporadic Jobs

- Reminder: (strictly) Sporadic jobs have hard deadlines; the release time and the execution time are **not** known *a priori*. Worst-case execution time **are known** when job is *released*.
- Need **acceptance test**:



$$S(c,l) = \sum_{i=c}^{l} s_i$$: Total amount of slack in Frames F_c, …, F_l.

- Acceptance Test:

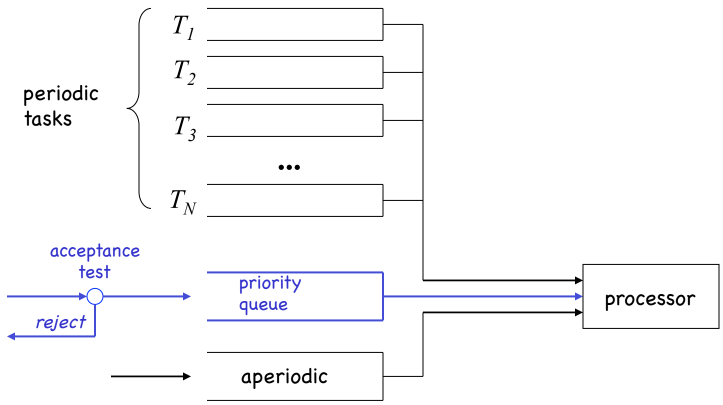
```
IF S(c,l) < e THEN
  reject job;
ELSE
  accept job;
  schedule execution;
END;
```

how?!

Scheduling of Accepted Jobs

- **Static scheduling**:
  - Schedule as large a slice of the accepted job as possible in the current frame.
  - Schedule remaining portions as late as possible.
- Mechanism:
  - Append slices of accepted job to list of periodic-task slices in frames where they are scheduled.
- Problem: **Early commit**.
- Alternatives:
  - Rescheduling upon arrival.
  - Priority-driven scheduling of sporadic jobs.

EDF-Scheduling of Accepted Jobs



Acceptance Test for EDF-Scheduled Sporadic Jobs

- Sporadic Job J with deadline d arrives:
- Test 1: Test whether current amount of slack before d is enough to accommodate J. (*)
  If not, <u>reject</u>!
- Test 2: Test whether sporadic jobs still in system with deadlines after d will miss deadline if J is accepted. (**)
  If yes, <u>reject</u>!
- <u>Accept</u>!

- (*) Define $S(J_i)$: Amount of slack up to time d_i after J_i has been scheduled.
- (**) Update all $S(J_i)$ with $d_i > d$, that is,

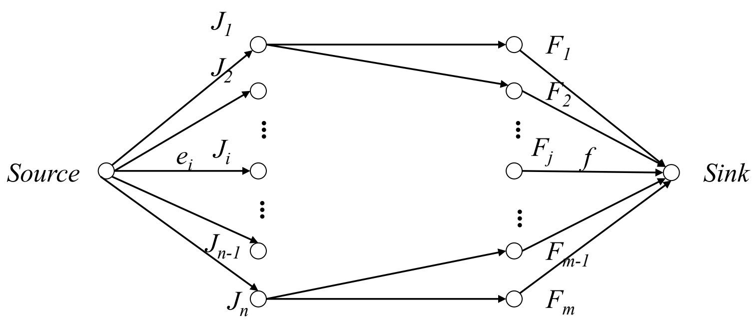
$$\forall i \quad \text{such that} \quad d_i > d: \quad S(J_i) = S(J_i) - e$$

Accept. Test for EDF Spor. Jobs (Implementation)

- Define
  $S_{i,k}$: slack in Frames F_i, ..., F_k
- Precompute all $S_{i,k}$ in first major cycle
- Initial amounts of slack in later cycles can be computed as
  $S_{i+jF,k+j'F} = S_{i,F} + S_{1,k} + (j'-j)S_{1,F}$
- Compute current slack of job with release time in F_{c-1} and deadline in F_{l+1}:
  $S^{new}_{c,l} = S_{c,l} - \Sigma_{(dk<d)} e_k(c)$
- Implementation:
  - Initially compute $S_{c,l}$ for newly arriving job. If negative, reject.
  - Whenever job with earlier deadline arrives, decrease this value. If negative, reject new job.

Static Scheduling of Jobs in Frames

- Layout of task schedule for cyclic executive can be formulated as a schedule for jobs in a hyperperiod.
- This can be formulated as a **network flow problem**.



Pros and Cons of Clock-Driven Scheduling

- Pros:
  - Conceptual simplicity
  - Timing constraints can be checked and enforced at frame boundaries.
  - Preemption cost can be kept small by having appropriate frame sizes.
  - Easy to validate: Execution times of slices known *a priori*.
- Cons:
  - Difficult to maintain.
  - Does not allow to integrate hard and soft deadlines.

Putting the Cyclic Executive into Practice

T. P. Baker, Alan Shaw, "The Cyclic Executive Model and Ada"

- Implementation approaches for a Cyclic Executive: Solutions and Difficulties
  - Naive solution using the DELAY statement
  - Using an interrupt from a hardware clock
  - Dealing with lost or buffered interrupts
  - Handling frame overruns

Naive Solution Using the DELAY Statement

```
class Slice {
public:
    virtual void execute() = 0;
};

class JobA : public Slice {
public:
    void execute() {
        < do something here >
    }
};
```

```
class CyclicExecutive {
    int frame_length;
    std::vector<int,std::vector<Slice>> frames;

    void AddJob(Slice& slice, int frame) {…}

    void Run() {
        int frame_no = 0;
        int next_time = gettimeofday() + frame_length;
        for(;;) {
            for(auto const& slice : frames[frame_no]) {
                slice.execute();
            }
            frame_no = (frame_no+1) % frames.Length();
            if (gettimeofday() > next_time) {
                HandleOverrun();
            }
            usleep(next_time - gettimeofday());
            next_time += frame_length;
        }
    }
};
```


Using an Interrupt from a Hardware Clock

```
class CyclicExecutiveV2 { // VANILLA IMPLEMENTATION!!
    int frame_length;
    std::vector<int,std::vector<Slice>> frames;
    int frame_no = 0;
    bool work_finished = true;

    void AddJob(Slice& slice, int frame) {…}

    static void ScheduleFrame(int signo) {
        if (! work_finished) HandleOverrun();
        work_finished = false;
        for(auto const& slice : frames[frame_no]) {
            slice.execute();
        }
        frame_no = (frame_no + 1) % frames.Length();
        work_finished = true;
    }

    void Run() {
        signal(SIGALRM, ScheduleFrame);
        setitimer(ITIMER_REAL, &interval, NULL);
        for(;;) pause();
    }
};
```

Dealing with Lost or Buffered Interrupts

```
class CyclicExecutiveV3 { // VANILLA !!
    int frame_length;
    std::vector<int,std::vector<Slice>> frames;
    int frame_no = 0;
    volatile bool work_finished = true;
    Semaphore sem = 0;

    void AddJob(Slice& slice, int frame) {…}

    static void ScheduleFrame(int signo) {
        if (! work_finished)
            HandleOverrun();
        work_finished = false;
        sem.V();
    }

    void Run() {
        std::thread worker(DoTheWork);
        signal(SIGALRM, ScheduleFrame);
        setitimer(ITIMER_REAL, &interval, NULL);
        for(;;) pause();
    }
};
```

in separate thread…

```
static void DoTheWork() {
  for(;;) {
    sem.P();
    for(auto const& slice
              : frames[frame_no]) {
        slice.execute();
    }
    frame_no = (frame_no + 1)
              % frames.Length();
    work_finished = true;
  }
}
```


Handling Frame Overruns (I)

ABORTION:

```
task type ACTION is -- the task that does the work
  entry NEXT_FRAME;
end ACTION;

type ACCESS_ACTION is access ACTION;

CURRENT_ACTION: ACCESS_ACTION:= new ACTION;

task body CYCLIC_EXECUTIVE_5 is
begin loop accept TIMER_INTERRUPT;
           select CURRENT_ACTION.NEXT_FRAME;
             else abort CURRENT_ACTION;
                  CURRENT_ACTION:= new ACTION;
           end select;
       end loop;
end CYCLIC_EXECUTIVE_5;
```

Source: T. P. Baker, Alan Shaw, "The Cyclic Executive Model and Ada"

Handling Frame Overruns (II)

EXCEPTIONS:

```
task body CYCLIC_EXECUTIVE_6 is
begin loop accept TIMER_INTERRUPT;
           select ACTION.NEXT_FRAME;
             else raise ACTION'failure;
           end select;
      end loop;
end CYCLIC_EXECUTIVE_6;

task body ACTION is
   FRAME_NUMBER: INTEGER:= 1;
begin loop accept NEXT_FRAME;
         begin FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
               case FRAME_NUMBER is
               when 0=> A; B; C; D1;
               when 1=> A; B; D2;
               end case;
         exception when others=> RECOVER_FROM_OVERRUN;
         end;
      end loop;
end ACTION;
```

Source: T. P. Baker, Alan Shaw, "The Cyclic Executive Model and Ada"