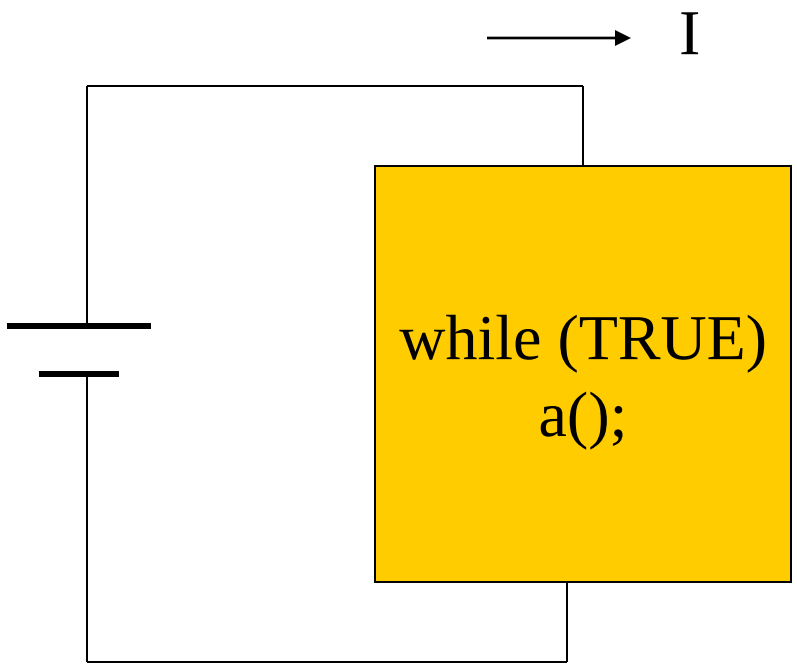
Measuring energy consumption

- Execute a small loop, measure current:



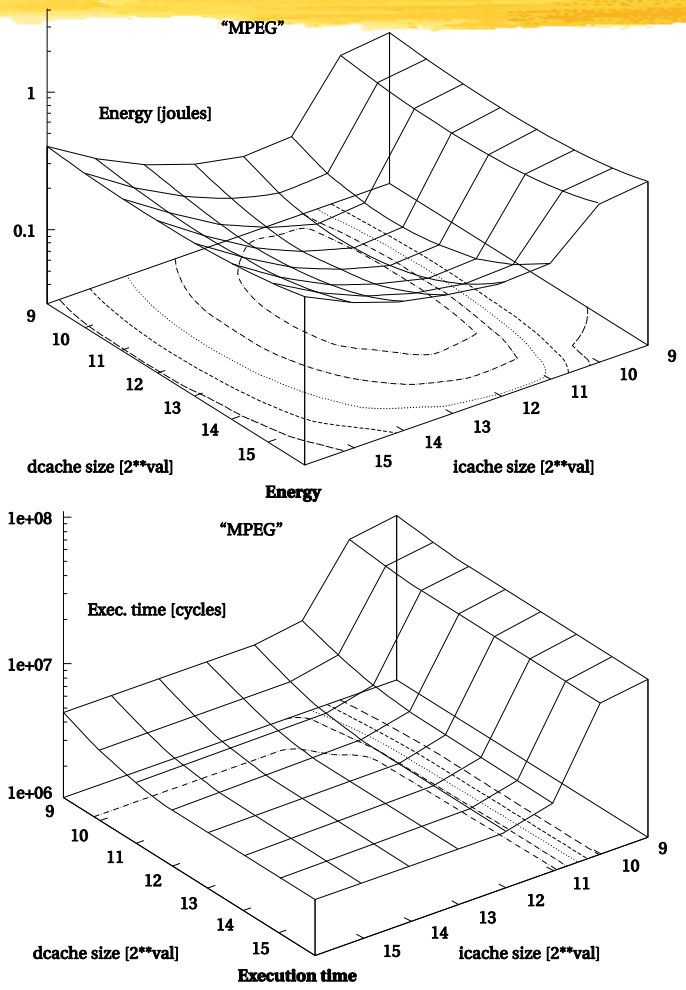
Sources of energy consumption

- Relative energy per operation (Catthoor et al):
  - off-chip 16-b memory transfer: 33
  - external I/O access: 10
  - 8x128x16 SRAM write: 9
  - 8x128x16 SRAM read: 4.4
  - 16-b multiply: 3.6
  - 16-b carry-select add: 1

Cache behavior is important

- Energy consumption has a sweet spot as cache size changes:
  - cache too small: program thrashes, burning energy on external memory accesses;
  - cache too large: cache itself burns too much power.
  - which all depend on a specific set of applications.

Cache sweet spot



[Li98] © 1998 IEEE

Optimizing for energy

- First-order optimization:
  - high performance = low energy.
- Not many instructions trade speed for energy.

Optimizing for energy

- Use registers efficiently.
- Identify and eliminate cache conflicts.
- Moderate loop unrolling eliminates some loop overhead instructions.
- Eliminate pipeline stalls.
- Inlining procedures may help: reduces linkage, but may increase cache thrashing.

Efficient loops

- General rules:
  - Don’t use function calls.
  - Use unsigned integer for loop counter.
  - Use <= to test loop counter.
  - Make use of compiler---global optimization, software pipelining.

Optimizing for program size

- Goal:
  - reduce hardware cost of memory;
  - reduce power consumption of memory units.
- Two opportunities:
  - data;
  - instructions.

Data size minimization

- Reuse constants, variables, data buffers in different parts of code.
  - Requires careful verification of correctness.
- Generate data using instructions.

Reducing code size

- Avoid function inlining.
- Choose CPU with compact instructions.
- Use specialized instructions where possible.

Program validation and testing

- Validation
  - Does it work as it is intended?
- Test
  - Concentrate here on functional verification.
- Major testing strategies:
  - Black box doesn't look at the source code.
  - Clear box (white box) does look at the source code.

Clear-box testing

- Examine the source code to determine whether it works:
  - Can you actually exercise a path?
  - Do you get the value you expect along a path?
- Testing procedure of 3 steps:
  - Controllability: provide program with inputs.
  - Execute.
  - Observability: examine outputs.

FIR filter with limiter

```
firout = 0.0;
for (j=curr, k=0; j<N; j++, k++)
    firout += buff[j] * c[k];
for (j=0; j<curr; j++, k++)
    firout += buff[j] * c[k];
if (firout > 100.0) firout = 100.0;
if (firout < -100.0) firout = -100.0;
```

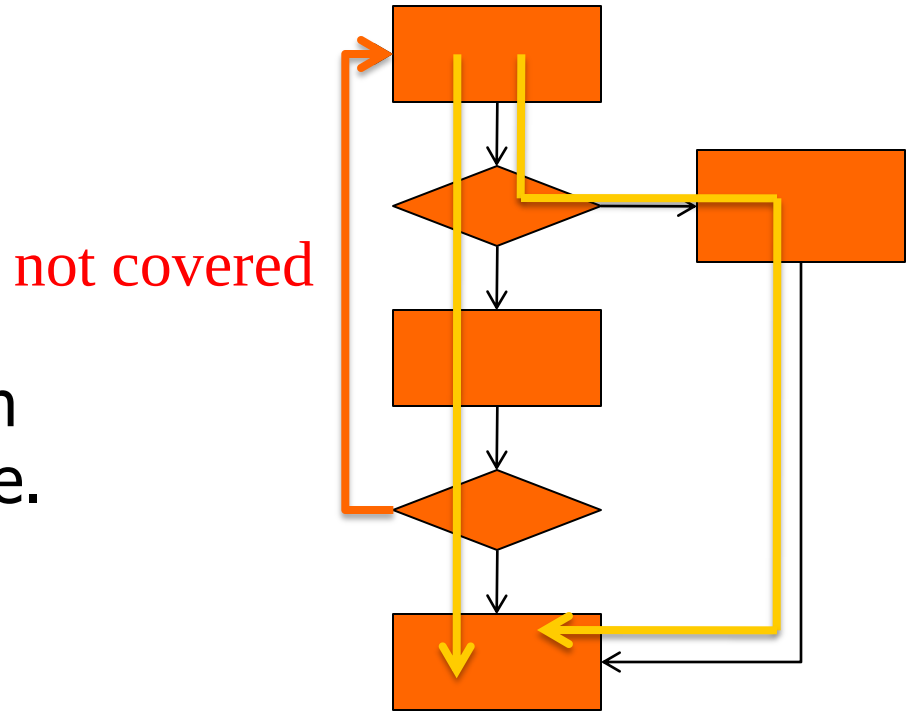
- What if we want to test whether the limiting code works?
- Controllability problem:
  - Must fill circular buffer with desired N values.
- What if we want to test the FIR filter itself?
- Observability problem:
  - Want to examine firout by setting a breakpoint before limit testing.

Execution paths and testing

- Paths are important in functional testing as well as performance analysis.
- In general, an exponential number of paths through the program.
  - Show that some paths dominate others.
  - Heuristically limit paths.

Choosing the paths to test

- Possible criteria:
  - Execute every statement at least once.
  - Execute every branch direction at least once.
- Equivalent for structured programs without gotos.



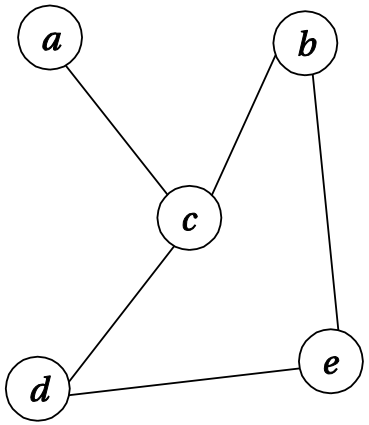
Two paths that covers all statements

How to choose a set of paths that covers the program's behavior

- Intuition tells us that a relatively small number of paths should be able to cover most practical programs.
- Graph theory helps us get a quantitative handle on this problem

Basis paths

- Approximate CDFG with undirected graph.
- Undirected graphs have basis paths:
  - All paths are linear combinations of basis paths.



Graph

	a	b	c	d	e
a	0	0	1	0	0
b	0	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

Incidence matrix

a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
e	0	0	0	0	1

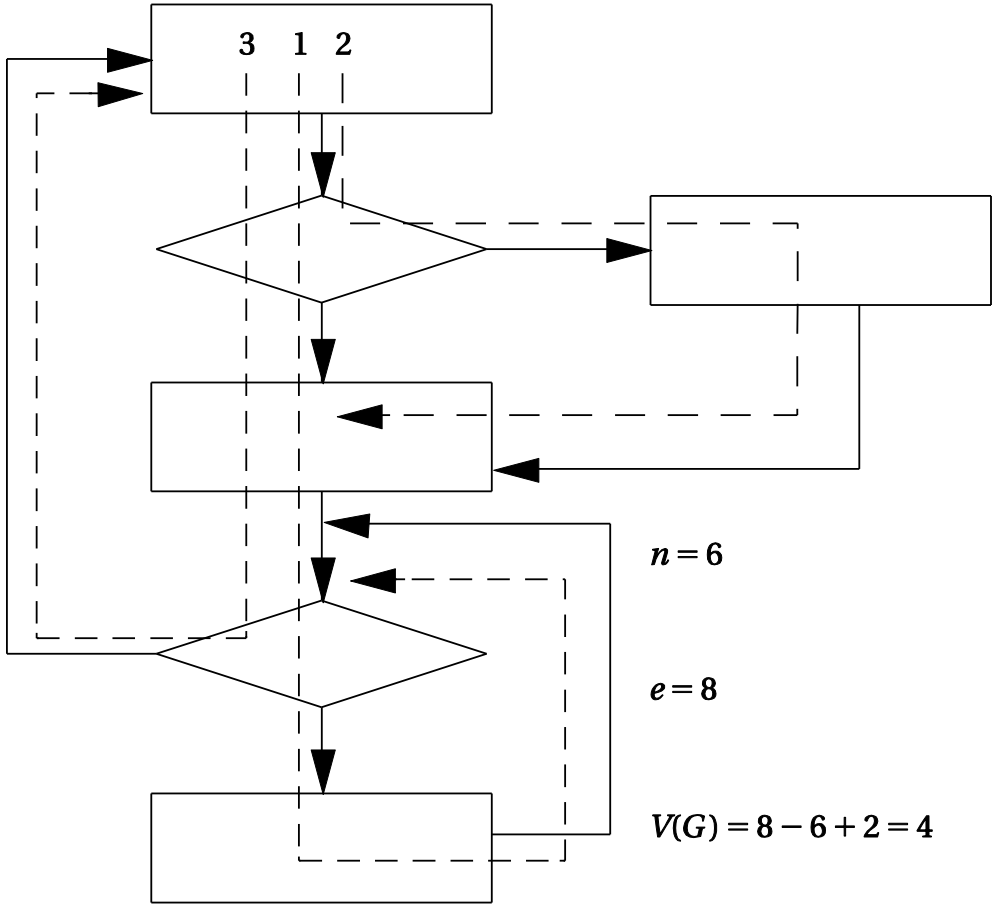
Basis set

Cyclomatic complexity

- A measure of the control complexity of a program
- Cyclomatic complexity is an upper bound on the size of basis sets:
  - e = # edges
  - n = # nodes
  - p = number of graph components
  - M = e – n + 2p.

Cyclomatic complexity

- M=4
- Because there are only three paths, it is an overly conservative bound.



Branch testing

- Heuristic for testing branches.
  - Exercise true and false branches of conditional.
  - Exercise every simple condition at least once.
- One of the reasons to use many different tests is to maximize the chances that supposedly unrelated elements will cooperate to reveal the error in a particular situation

Branch testing example

- Correct version:
  - if (a || (b >= c)) { printf("OK□n"); }
- Incorrect version:
  - if (a && (b >= c)) { printf("OK□n"); }
- Test:
  - a = F
  - (b >=c) = T
- Example:
  - Correct: [0 || (3 >= 2)] = T
  - Incorrect: [0 && (3 >= 2)] = F
- This test picks up the error.

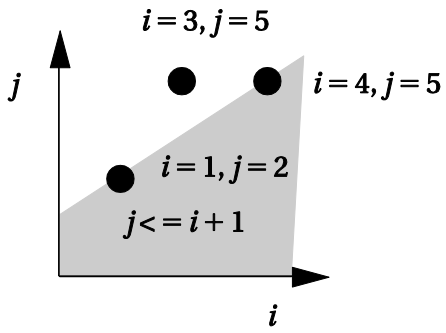
Another branch testing example

- Correct version:
  - if ((x == good_pointer) && x->field1 == 3)) { printf("got the value□n"); }
- Incorrect version:
  - if ((x = good_pointer) && x->field1 == 3)) { printf("got the value□n"); }
- Incorrect code changes pointer.
  - Assignment returns new LHS in C.
- Test that could catch error:
  - (x != good_pointer) && x->field1 = 3)

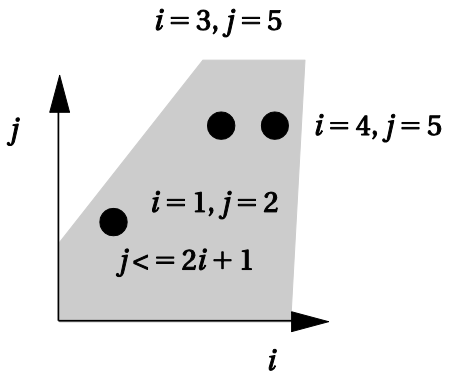
    can't detect error

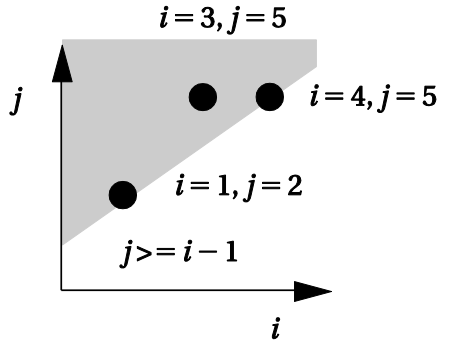
Domain testing for conditionals

- Heuristic test for linear inequalities.
- Test on each side of boundary of inequality.
- It is not working in testing both true and false paths when typing error exists in inequality.



Correct test

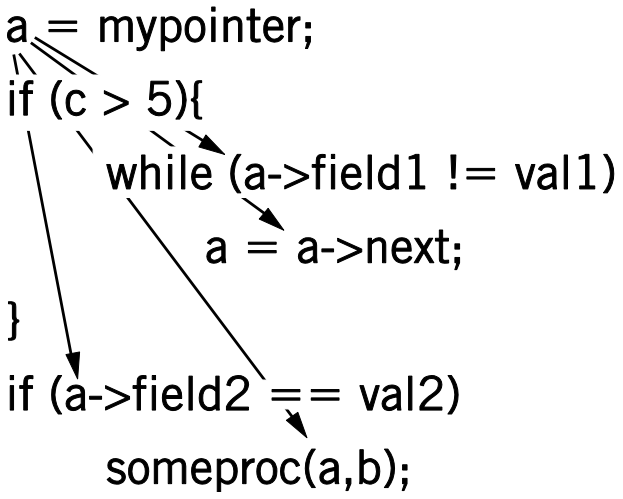




Incorrect tests

Def-use pairs

- Variable def-use:
  - Def: when value is assigned (defined).
  - Use: when used on right-hand side.
- Exercise enough def-use pairs.
  - Requires testing correct path.



- p-use: predicate use
- c-use: computation use

Def-use analysis

- Def-use analysis can be performed on a program using iterative algorithms
- Data flow testing chooses tests that exercises chosen def-use pairs.

Loop testing

- Loops need specialized tests to be tested efficiently.
- Heuristic testing strategy:
  - Skip loop entirely.
  - One loop iteration.
  - Two loop iterations.
  - k iterations much below n.
  - n-1, n, n+1 iterations where n is max.

Black-box testing

- Complements clear-box testing.
  - May require a large number of tests.
- Tests software in different ways.

Black-box test vectors

- Random tests.
  - May weight distribution based on software specification.
- Regression tests.
  - Tests of previous versions, bugs, etc.
  - May be clear-box tests of previous versions.

How much testing is enough?

- Coverage
- Exhaustive testing is impractical.
- One important measure of test quality---bugs escaping into field.
- Good organizations can test software to give very low field bug report rates.
- Error injection measures test quality:
  - Add known bugs.
  - Run your tests.
  - Determine % injected bugs that are caught.

Testing summary

- Statement testing: every node (statement)
- Branch testing: every branch
- Path testing: every path
- Statement and branch testing can fail to expose many common errors
- Path testing is usually infeasible since programs with loops have infinitely many paths
- Using control flow analysis
  - Stronger than branch testing
  - Weaker than path testing

Testing summary

- Using data flow analysis
  - Bridge the gap between branch testing and path testing
  - All testing criteria suffers form weakness that can't find test vector for un-executable path in program
- Applicability property of a testing criterion
  - iff for every program P there exists some tests set which is C-adequate for P.
  - Statement testing, branch testing, path testing, and DF testing all fail to satisfy the applicability property.
  - Furthermore, for each of them it is undecidable whether a test set exists which adequately tests a given program.

Testing summary

- One way to enforce the applicability
  - Restrict the class of programs to the set of programs that satisfy it.
  - Defne new testing criteria
- An applicable family of data flow testing criteria
  - by Frankl and Weyuker [1988]

6. Processes and operating systems

- Multiple tasks and multiple processes.
  - Specifications of process timing.
- Preemptive real-time operating systems.
- Processes

Reactive systems

- Respond to external events.
  - Engine controller.
  - Seat belt monitor.
- Requires real-time response.
  - System architecture.
  - Program implementation.
- May require a chain reaction among multiple processors.

Tasks and processes

- A task is a functional description of a connected set of operations.
- (Task can also mean a collection of processes.)
- Threads are processes that share the address space
- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Why multiple processes?

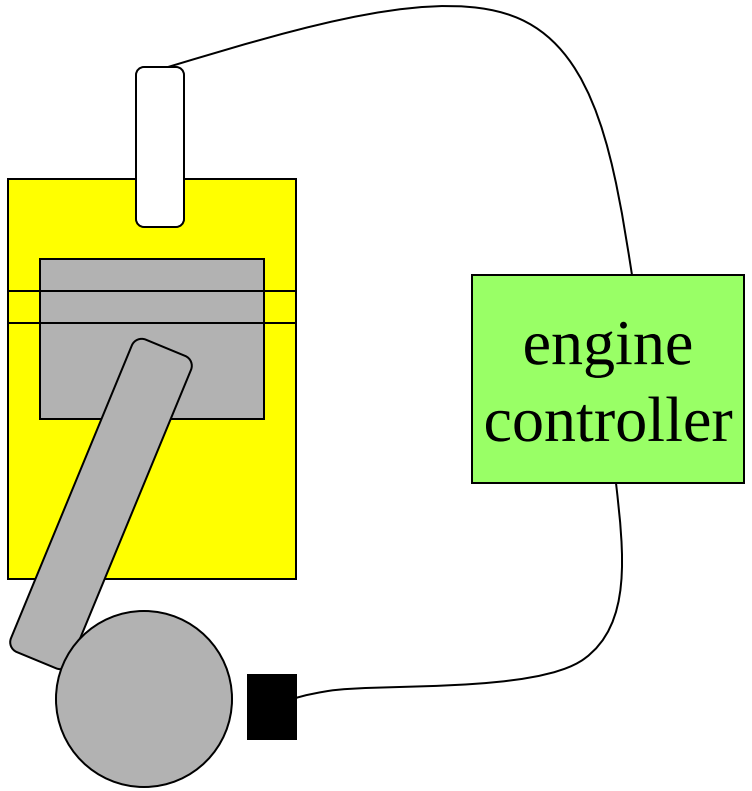
- Multiple tasks means multiple processes.
- Processes help with timing complexity:
  - multiple rates -> multiple tasks
    - Examples: multimedia, automotive
  - asynchronous input
    - Examples: user interfaces, communication systems

Multi-rate systems

- Tasks may be synchronous or asynchronous.
- Synchronous tasks may recur at different rates.
- Processes run at different rates based on computational needs of the tasks.

Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter



Typical rates in engine controllers

Variable	Full range time (ms)	Update period (ms)
Engine spark timing	300	2
Throttle	40	2
Air flow	30	4
Battery voltage	80	4
Fuel flow	250	10
Recycled exhaust gas	500	25
Status switches	100	20
Air temperature	Seconds	400
Barometric pressure	Seconds	1000
Spark (dwell)	10	1
Fuel adjustment	80	8
Carburetor	500	25
Mode actuators	100	100

Real-time systems

- Perform a computation to conform to external timing constraints.
- Deadline frequency:
  - Periodic.
  - Aperiodic.
- Deadline type:
  - Hard: failure to meet deadline causes system failure.
  - Soft: failure to meet deadline causes degraded response.
  - Firm: late response is useless but some late responses can be tolerated.

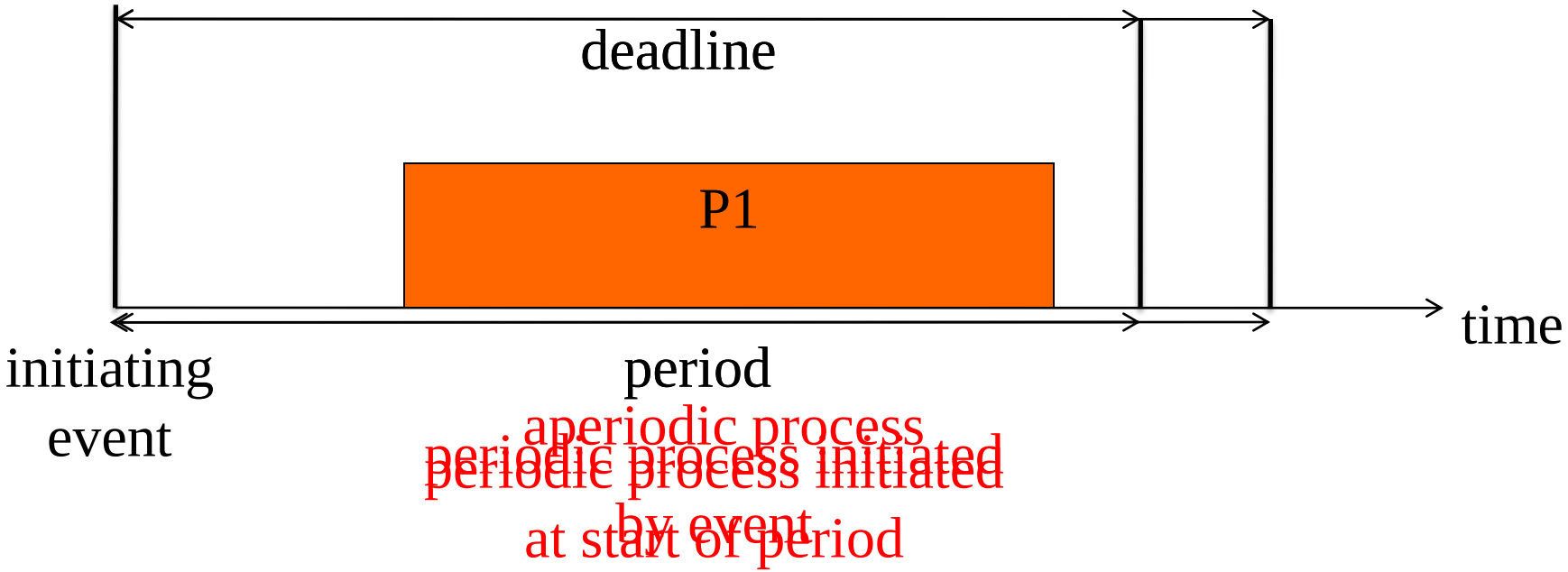
Timing specifications on processes

- Release time
  - time at which process becomes ready.
- Deadline
  - time at which process must finish.
- Aperiodic process
  - Initiated by an event
  - Release time is measured from that event

Periodic processes

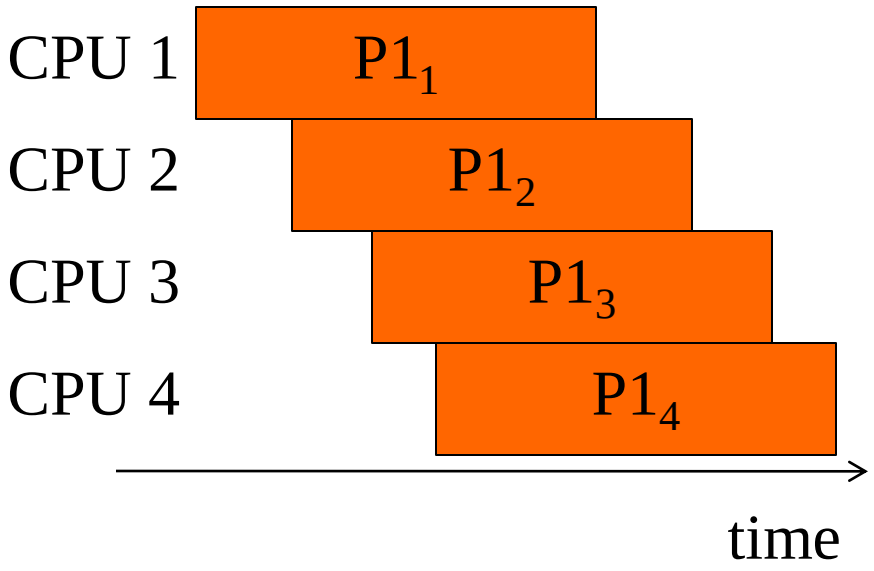
- Release time can be
  - At the beginning of the period
  - At the arrival time of certain data; at a time after the start of the period
- Deadline can be
  - At the end of the period
  - At some time after the end of the period

Release times and deadlines



Rate requirements on processes

- Period: interval between process activations.
- Rate: reciprocal of period.
- Initiation rate may be higher than period---several copies of process run at once.

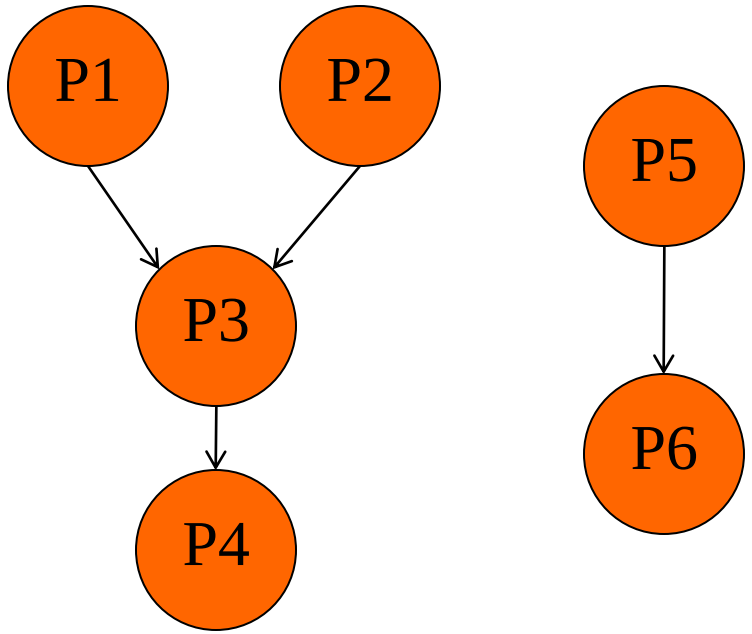


Timing violations

- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

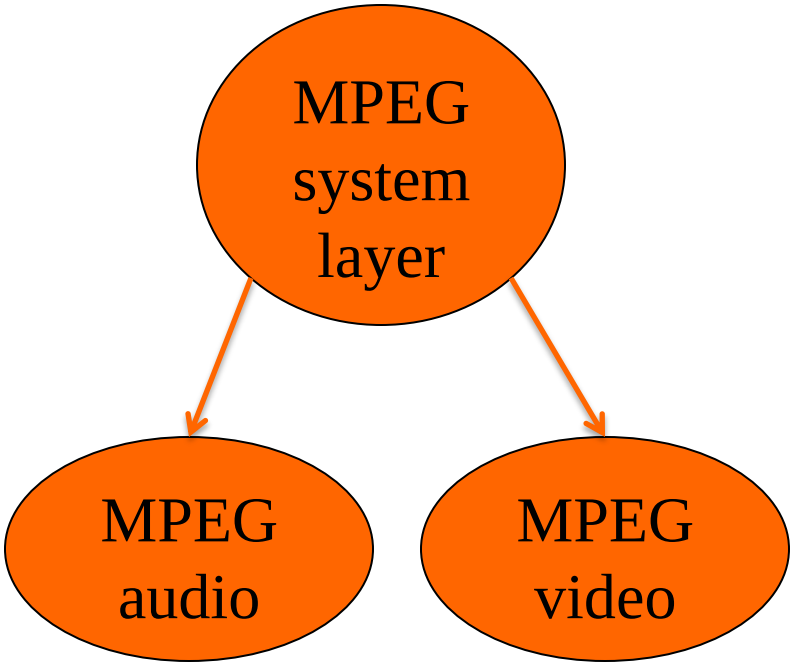
Task graphs

- Tasks may have data dependencies---must execute in certain order.
- Task graph shows data/control dependencies between processes.
- Task: connected set of processes.
- Task set: One or more tasks.



Communication between tasks

- Task graph assumes that all processes in each task run at the same rate, tasks do not communicate.
- In reality, some amount of inter-task communication is necessary.
  - It's hard to require immediate response for multi-rate communication.



Process execution characteristics

- Process execution time T_i.
  - Execution time in absence of preemption.
  - Possible time units: seconds, clock cycles.
  - Worst-case, best-case execution time may be useful in some cases.
- Sources of variation:
  - Data dependencies.
  - Memory system.
  - CPU pipeline.

Utilization

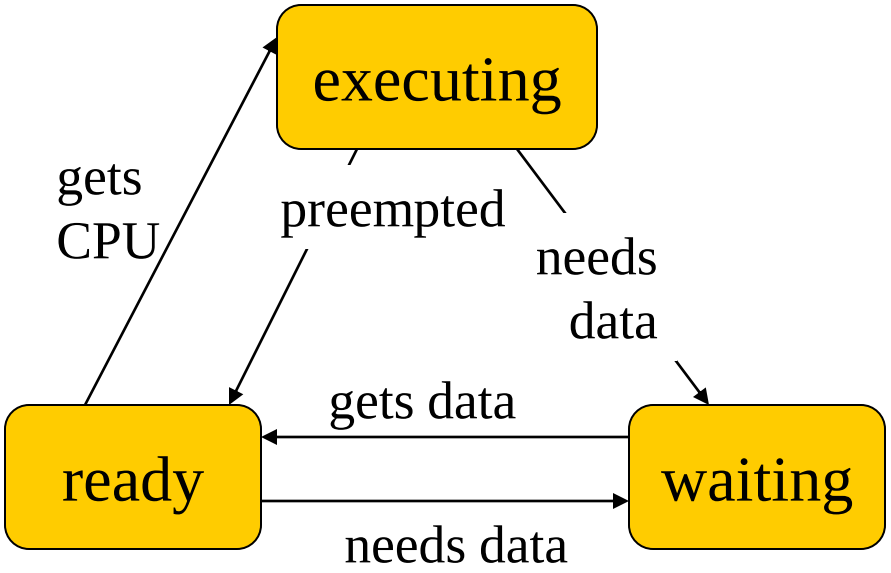
- CPU utilization:
  - Fraction of the CPU that is doing useful work.
  - Often calculated assuming no scheduling overhead.
- Utilization:
  - U = (CPU time for useful work)/ (total available CPU time)

    = $[\Sigma_{t1 \le t \le t2} T(t)] / [t2 - t1]$

    = T/t

State of a process

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.

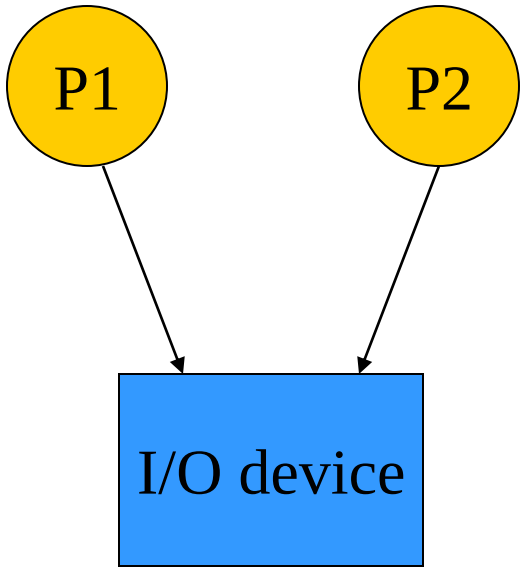


Scheduling problem

- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?

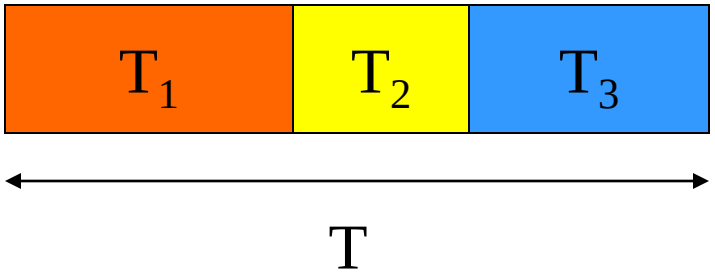
Scheduling feasibility

- Resource constraints make schedulability analysis NP-hard.
  - Must show that the deadlines are met for all timings of resource requests.



Simple processor feasibility

- Assume:
  - No resource conflicts.
  - Constant process execution times.
- Require:
  - $T \geq \Sigma_i T_i$
  - Can't use more than 100% of the CPU.



Hyperperiod

- Hyperperiod: least common multiple (LCM) of the task periods.
- Must look at the hyperperiod schedule to find all task interactions.
- Hyperperiod can be very long if task periods are not chosen carefully.

Hyperperiod example

- Long hyperperiod:
  - P1 7 ms.
  - P2 11 ms.
  - P3 15 ms.
  - LCM = 1155 ms.
- Shorter hyperperiod:
  - P1 8 ms.
  - P2 12 ms.
  - P3 16 ms.
  - LCM = 96 ms.

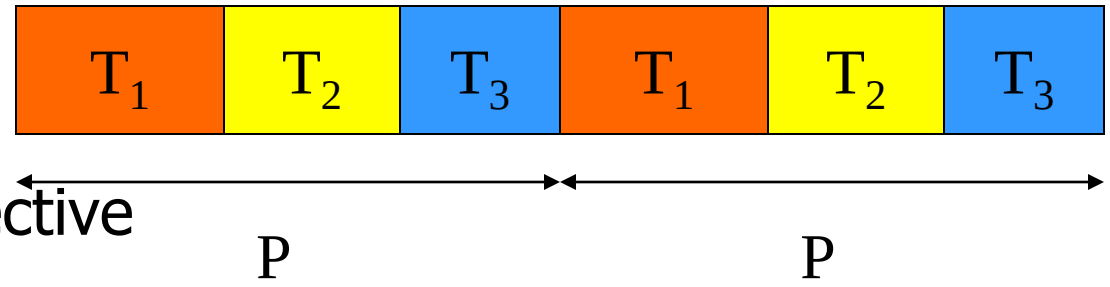
Simple processor feasibility example

- P1 period 1 ms, CPU time 0.1 ms.
- P2 period 1 ms, CPU time 0.2 ms.
- P3 period 5 ms, CPU time 0.3 ms.

LCM		5.00E-03	
	peirod	CPU time	CPU time/L
P1	1.00E-03	1.00E-04	5.00E-04
P2	1.00E-03	2.00E-04	1.00E-03
P3	5.00E-03	3.00E-04	3.00E-04
	total CPU/LCM		1.80E-03
	utilization		3.60E-01

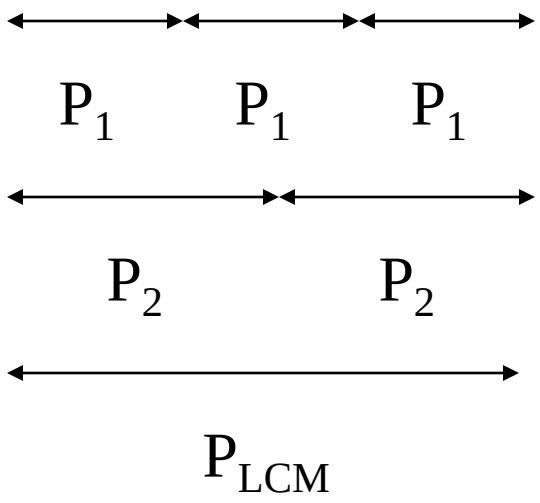
Cyclostatic/TDMA

- Schedule in time slots.
  - Same process activation irrespective of workload.
  - Always same order
- Time slots may be equal size or unequal.



TDMA assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Trivial scheduler -> very small scheduling overhead.



TDMA schedulability

- Always same CPU utilization (assuming constant process execution times).
- Can't handle unexpected loads.
  - Must schedule an extra time slot for aperiodic events.

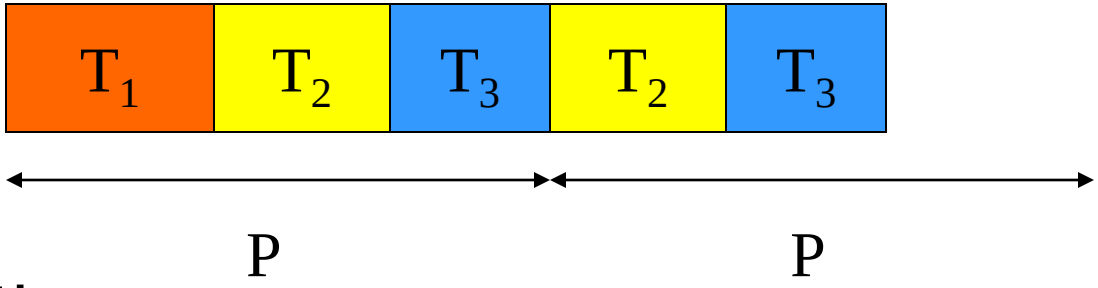
TDMA schedulability example

- TDMA period = 10 ms.
- P1 CPU time 1 ms.
- P2 CPU time 3 ms.
- P3 CPU time 2 ms.
- P4 CPU time 2 ms.

TDMA period		1.00E-02	
	CPU time		
P1	1.00E-03		
P2	3.00E-03		
P3	2.00E-03		
P4	2.00E-03		
total	8.00E-03		
utilization	8.00E-01		

Round-robin

- Schedule process only if ready.
  - Always test processes the same order.
  - Skip a process if not ready
- Variations to cyclostatic:
  - Constant system period.
  - Start round-robin again after finishing a round.



Round-robin assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Best done with equal time slots for processes.
- Simple scheduler -> low scheduling overhead.
  - Can be implemented in hardware.

Round-robin schedulability

- Can bound maximum CPU load.
  - May leave unused CPU cycles.
- Can be adapted to handle unexpected load.
  - Exploit time slots at end of period.

Schedulability and overhead

- The scheduling process consumes CPU time.
  - Not all CPU time is available for processes.
- Scheduling overhead must be taken into account for exact schedule.
  - May be ignored if it is a small fraction of total execution time.

Running periodic processes

- Need code to control execution of processes.
- Simplest implementation: process = subroutine.

while loop implementation

- Simplest implementation has one loop.
  - No control over execution rate of processes.
  - All the processes run at the same rate

```
while (TRUE) {
    p1();
    p2();
}
```

Timed loop implementation

- Encapuslate set of all processes in a single function that implements the task set,.
- Use timer to control execution of the task.
  - No control over execution rate of individual processes.

```
void pall(){
    p1();
    p2();
}
```

Multiple timers implementation

- Each task has its own function.
- Each task has its own timer.
  - May not have enough timers to implement all the rates.

```
void pA(){ /* rate A */
    p1();
    p3();
}
void pB(){ /* rate B */
    p2();
    p4();
    p5();
}
```

Timer + counter implementation

- Use a software count to divide the timer.
- Only works for clean multiples of the timer period.

```
int p2count = 0;
void pall(){
    p1();
    if (p2count >= 2) {
        p2();
        p2count = 0;
        }
    else p2count++;
    p3();
}
```

p2 runs 1/3 the rate of p1 and p3

Implementing processes

- All of these implementations are inadequate.
- Need better control over timing.
- Need a better mechanism than subroutines.