

6. Processes and OS



- ⌘ Process abstraction
- ⌘ Context switching
- ⌘ RTOS
- ⌘ Interprocess communication
- ⌘ Task-level performance analysis and power consumption

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: processes that share the same 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.

Tasks



- ⌘ Multiple tasks means multiple processes.
 - ☐ Somewhat interchangeably used
 - ☐ A task can be composed of several processes or threads
- ⌘ A task is primarily an implementation concept
- ⌘ A process more of an implementation concept

Why multiple processes?



- ⌘ Requirements on timing and execution rate can create major problems in programming.
- ⌘ When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex.
- ⌘ Processes help with timing complexity:
 - ⌘ multiple rates
 - ⊗ multimedia
 - ⊗ automotive
 - ⌘ asynchronous input
 - ⊗ user interfaces
 - ⊗ communication systems

Multi-rate systems

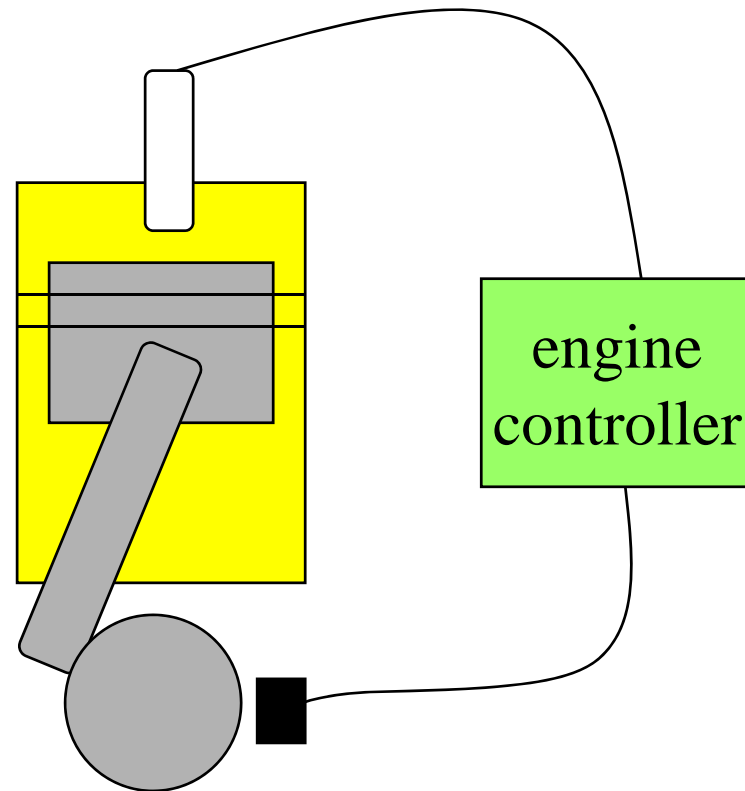


- ⌘ Tasks may be synchronous or asynchronous.
- ⌘ Synchronous tasks may recur at different rates.
- ⌘ Processes run at different rates based on computational needs of the tasks.

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.

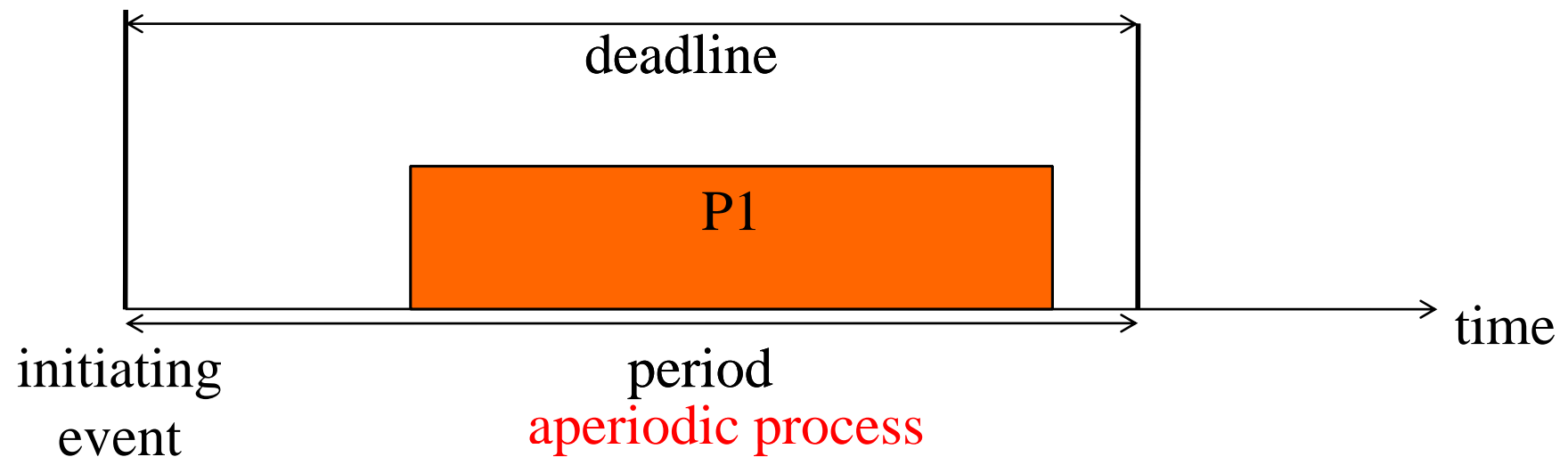
Timing specifications on processes



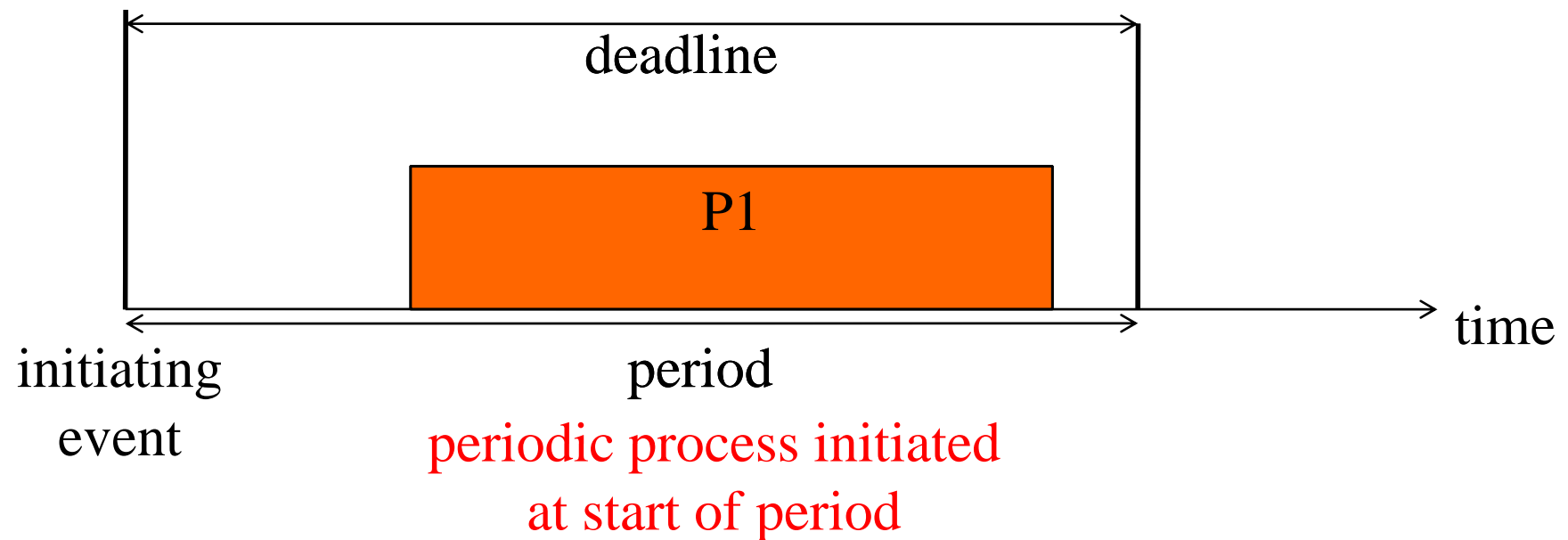
⌘ **Release time**: time at which process becomes ready.

⌘ **Deadline**: time at which process must finish.

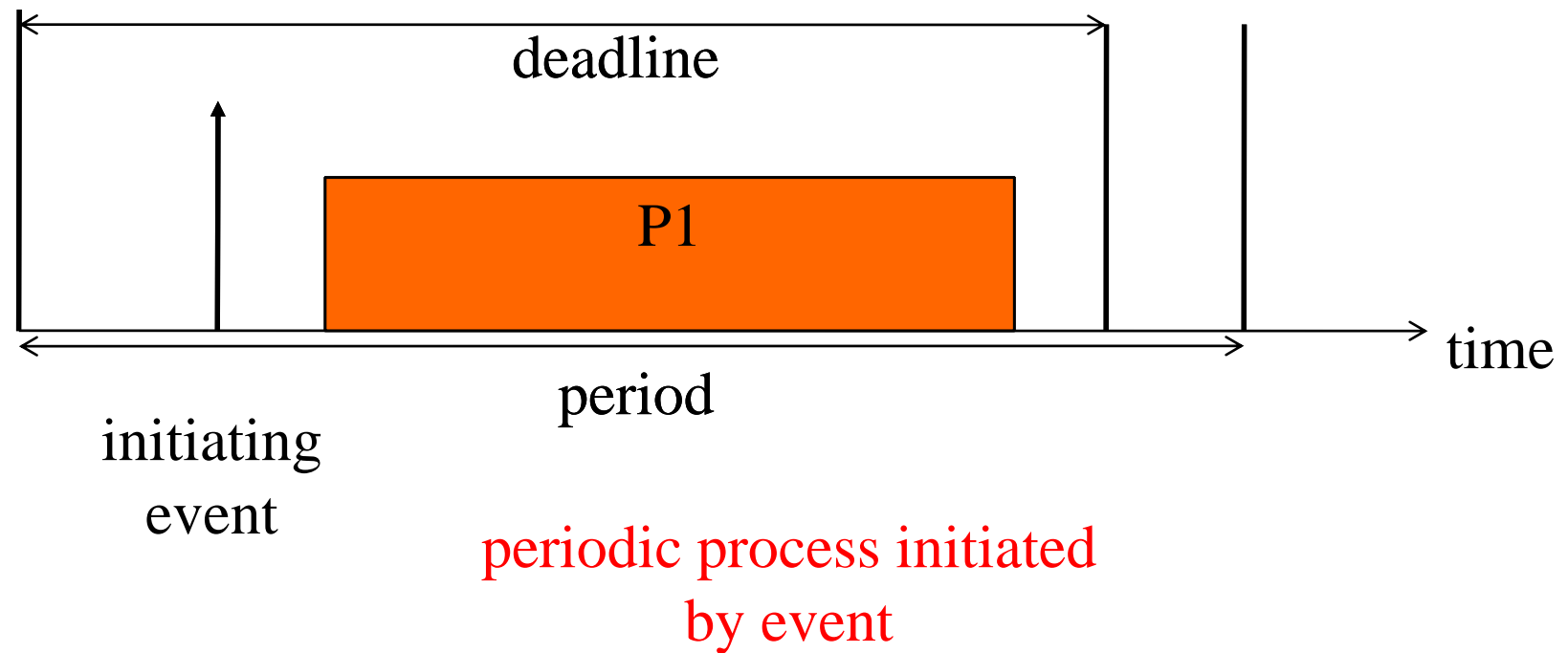
Release times and deadlines



Release times and deadlines



Release times and deadlines

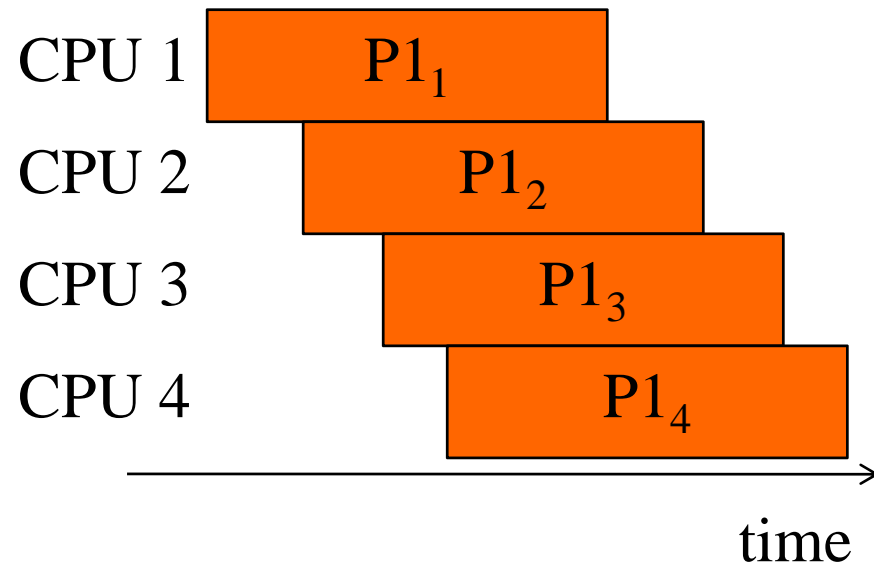


Rate requirements on processes

⌘ **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.

Example: Space Shuttle software error

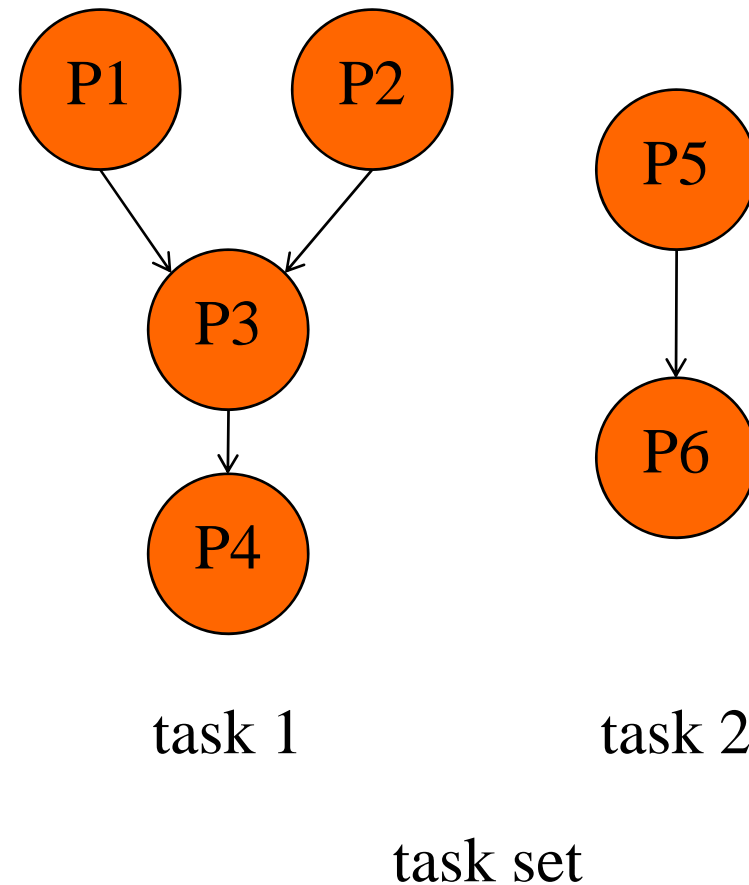


⌘ Space Shuttle's first launch was delayed by a software timing error:

- ☐ Primary control system PASS and backup system BFS.
- ☐ BFS failed to synchronize with PASS.
- ☐ Change to one routine added delay that threw off start time calculation.
- ☐ 1 in 67 chance of timing problem.

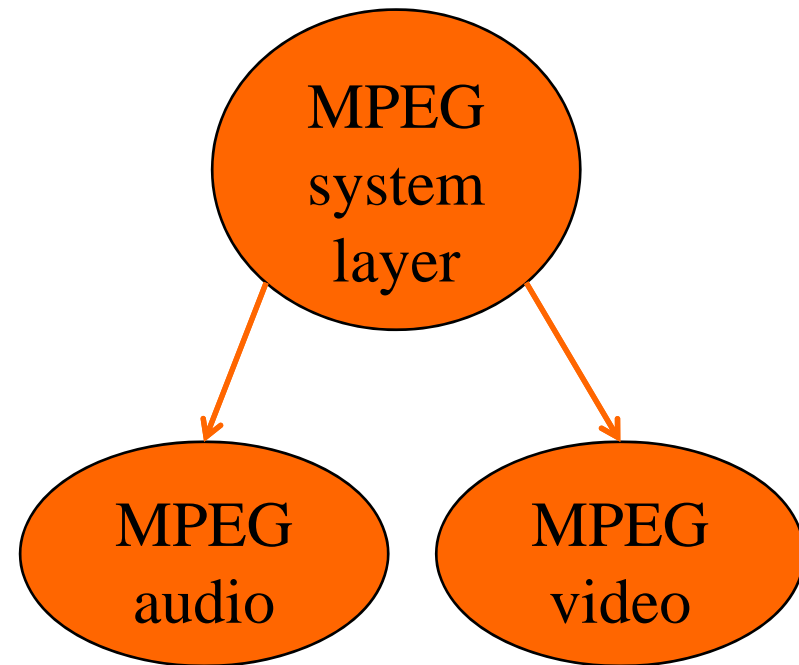
Task graphs

- ⌘ 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:

$$\begin{aligned}\text{☒ } U &= (\text{CPU time for useful work}) / (\text{total available CPU time}) \\ &= [\sum_{t_1 \leq t \leq t_2} T(t)] / [t_2 - t_1] \\ &= T/t\end{aligned}$$

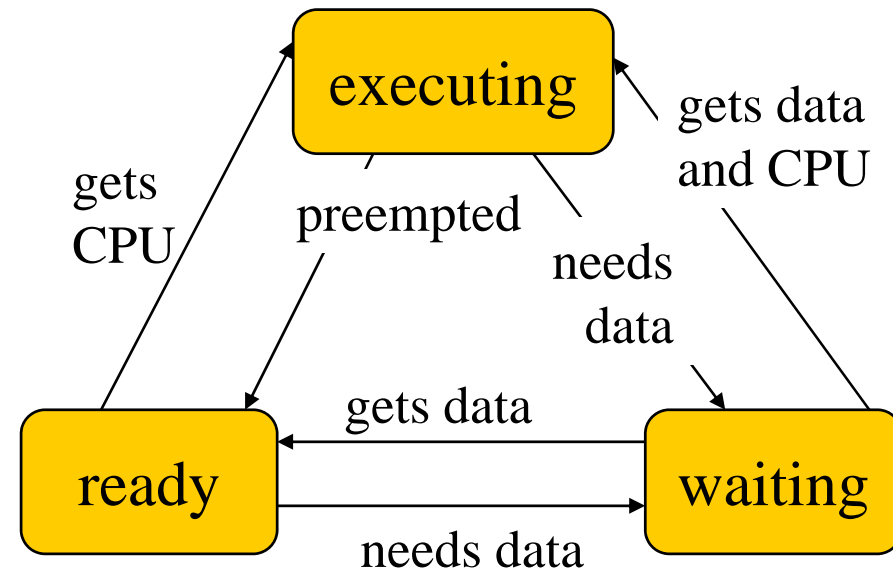
State of a process

⌘ A process can be in one of three states:

⏏ **executing** on the CPU;

⏏ **ready** to run;

⏏ **waiting** for data.



The scheduling problem



⌘ Can we meet all deadlines?

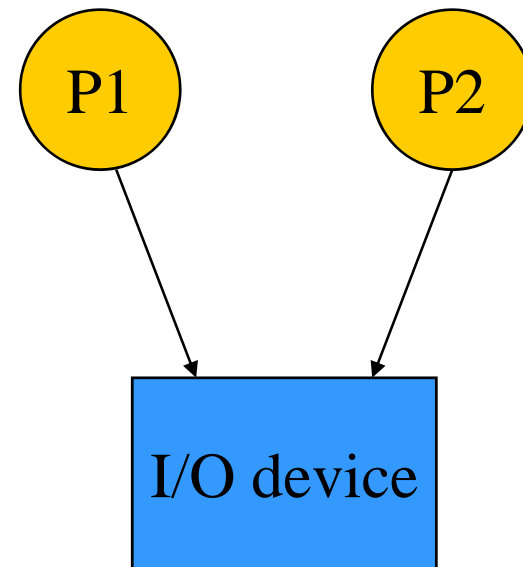
☑ Must be able to meet deadlines in all cases.

⌘ How much CPU horsepower do we need to meet our deadlines?

Scheduling feasibility

⌘ Resource constraints make schedulability analysis NP-hard.

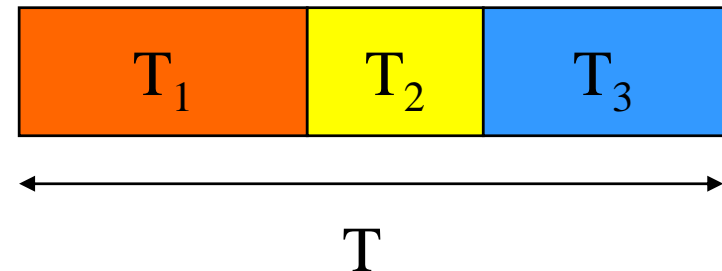
☑ Must show that the deadlines are met for all timings of resource requests.



Simple processor feasibility

⌘ Assume:

- ⊞ No resource conflicts.
- ⊞ Constant process execution times.



⌘ Require:

- ⊞ $T \geq \sum_i T_i$
- ⊞ 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

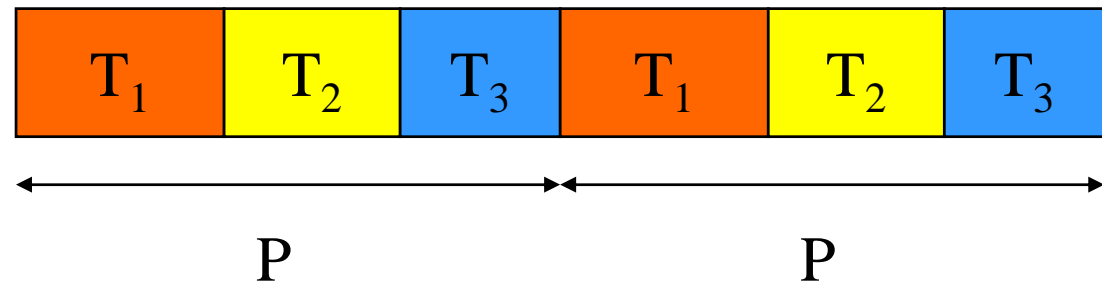
- ⌘ P1 period 1 ms, CPU time 0.1 ms.
- ⌘ P2 period 1 ms, CPU time 0.2 ms.
- ⌘ P3 period 5 ms, CPU time 0.3 ms.
- ⌘ In a hyperperiod, 5 P1 and P2 executed five times and P3 once
 $U = 1.8 \text{ ms} / 5 \text{ ms} = 36\%$

LCM		5.00E-03	
	peirod	CPU time	CPU time/LCM
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 scheduling

⌘ Schedule in time slots.

⏏ Same process activation irrespective of workload.

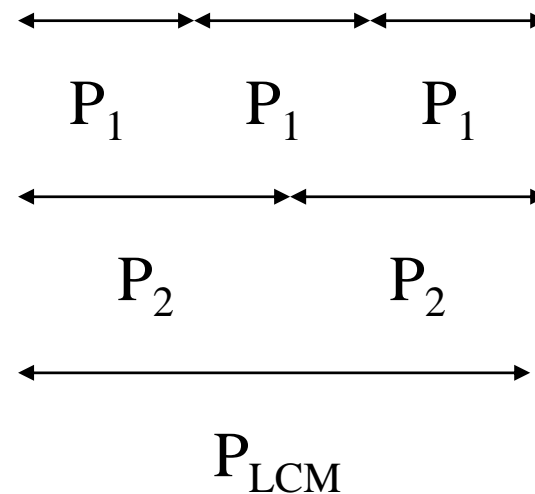


⌘ Time slots may be equal size or unequal.

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 a 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

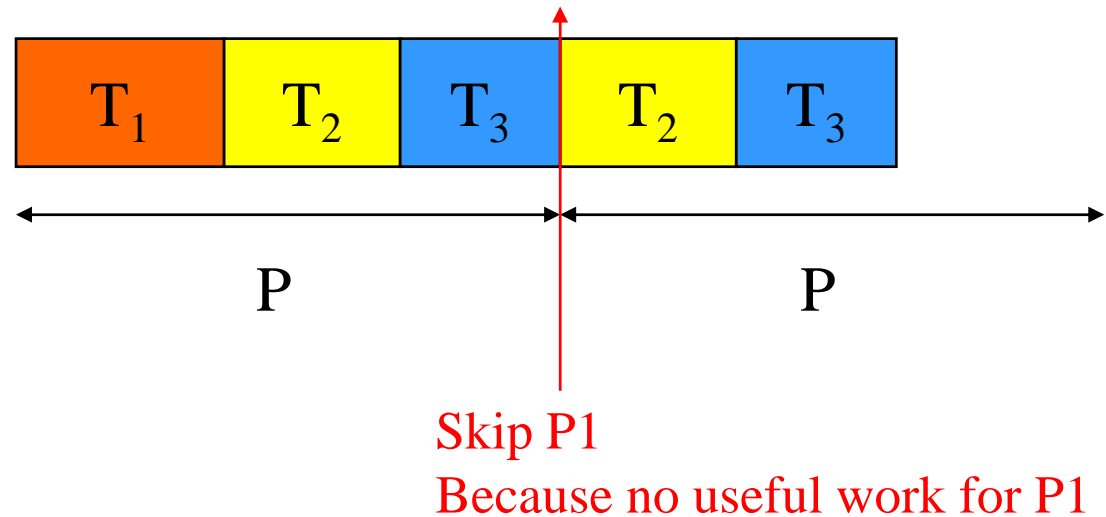
⌘ Schedule process only if ready.

☑ Always test processes in the same order.

⌘ Variations:

☑ 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.

- ☒ Use time slots at end of period.

- ☒ Last slot left empty for aperiodic tasks

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.

☐ All processes should
have the same rate

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

Timed loop implementation

⌘ Encapsulate set of all processes in a single function that implements the task set,.

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

⌘ Use timer to control execution of the task.

⏏ No control over timing of individual processes.

⌘ the timer's interrupt handler : pall()


⌘ If a process is too slow, next iteration start late

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.
- ⌘ We may not have enough timers to support all the rate required in the system

```
void pA(){ /* rate A */  
    p1();  
    p3();  
}  
void B(){ /* 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.

⌘ `p2()` must run at $1/3$ the rate of `p1()`

⌘ Rates should be multiple each other

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

Implementing processes



- ⌘ All of these implementations are inadequate.
 - ☐ When the rates are not related by simple ratio
- ⌘ Need better control over timing.
- ⌘ Need a better mechanism than subroutines.
- ⌘ We need to employ RTOS

Operating systems



⌘ The operating system controls resources:

- ☑ who gets the CPU;
- ☑ when I/O takes place;
- ☑ how much memory is allocated.

⌘ The most important resource is the CPU itself.

- ☑ CPU access controlled by the scheduler.

RTOS



- ⌘ Executes processes based on timing constraints provided by the system designer.
- ⌘ The most reliable way to meet timing constraints accurately is
 - ☑ to build a **preemptive** OS
 - ☑ to use **priorities** to control what process runs at any given time.

Preemption

- ⌘ An alternative to the C function call as a way to control function calls as process execution with a timer.
- ⌘ We want to share across two processes.
 - ⌘ **kernel**: part of OS that determines what process is running, activated periodically by the timer.
 - ⌘ **time quantum**: the timer period, which is the smallest increment in which we can control CPU activity.
- ⌘ How do we switch between processes before the process is done?
 - ⌘ **Context (the set of CPU registers) switching**
 - ⌘ **Process control block**: the data structure that hold the state of the process.

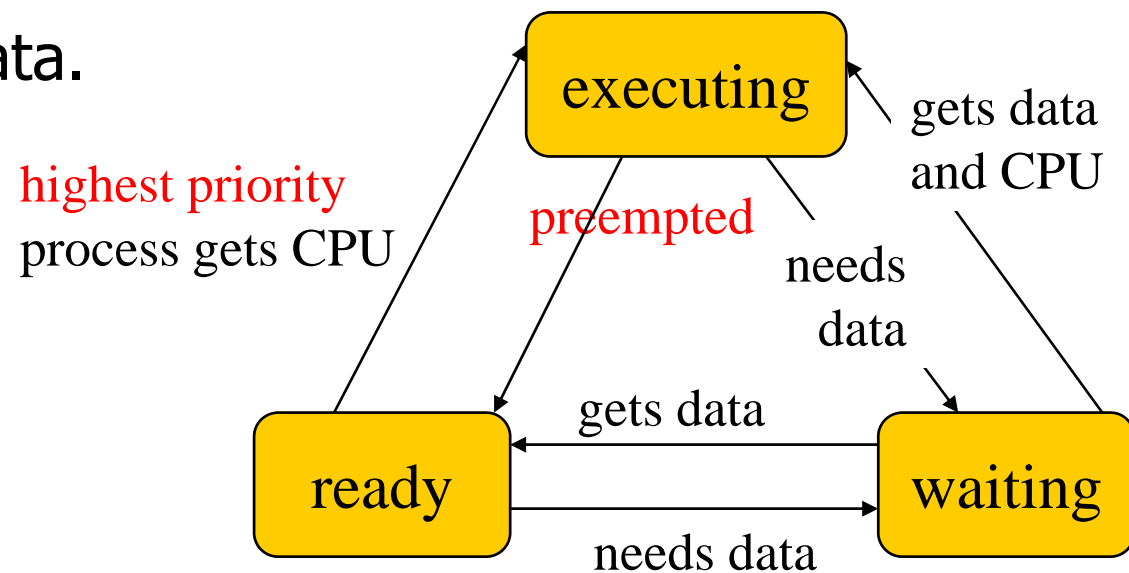
Process state

⌘ A process can be in one of three states:

☑ **executing** on the CPU;

☑ **ready** to run;

☑ **waiting** for data.



Operating system structure




⌘ OS needs to keep track of:

- ☑ process priorities;
- ☑ scheduling state;
- ☑ process activation record.

⌘ Processes may be created:

- ☑ statically before system starts;
- ☑ dynamically during execution.

Embedded vs. general-purpose scheduling



⌘ Workstations try to avoid starving processes of CPU access.

☐ Fairness = access to CPU.

⌘ Embedded systems must meet deadlines.

☐ Low-priority processes may not run for a long time.

Priority-driven scheduling



- ⌘ Each process has a priority.
- ⌘ CPU goes to highest-priority process that is ready.
- ⌘ Priorities determine scheduling policy:
 - ☐ fixed priority;
 - ☐ time-varying priorities.

Example: priority-driven scheduling



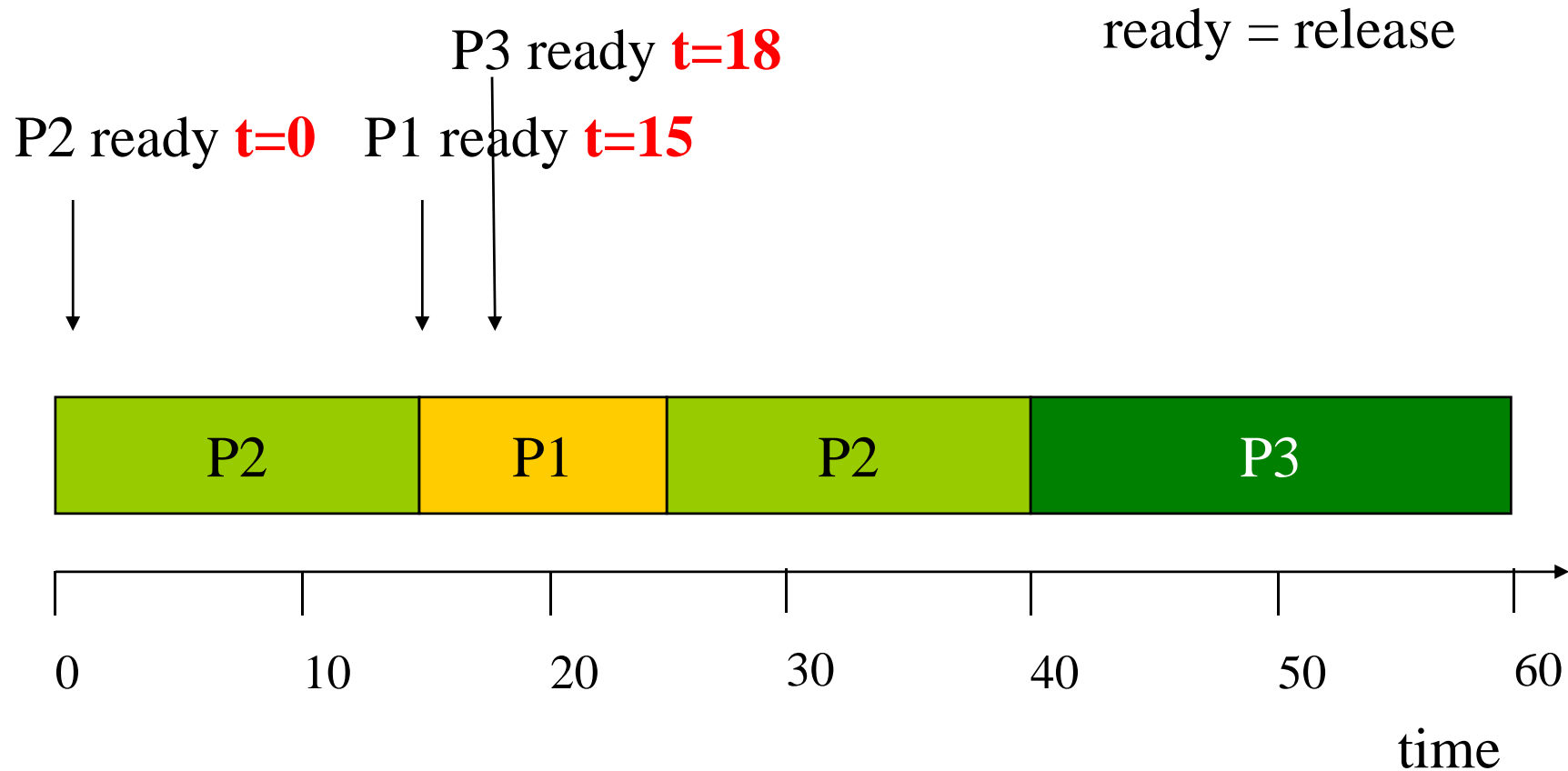
⌘ Rules:

- ☑ each process has a fixed priority (1 highest);
- ☑ highest-priority ready process gets CPU;
- ☑ process continues until done or it is preempted by a higher-priority process.

⌘ Processes

- ☑ P1: priority 1, execution time 10, release at time 15
- ☑ P2: priority 2, execution time 30, release at time 0
- ☑ P3: priority 3, execution time 20, release at time 18

Example: priority-driven scheduling



The 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?

Process initiation disciplines



- ⌘ **Periodic process**: executes on (almost) every period.
- ⌘ **Aperiodic process**: executes on demand.
- ⌘ Analyzing aperiodic process sets is harder--must consider worst-case combinations of process activations.

Timing requirements on processes



- ⌘ **Period**: interval between process activations.
- ⌘ **Initiation interval**: time difference between process starting; reciprocal of period.
- ⌘ **Initiation time**: time at which process becomes ready.
- ⌘ **Deadline**: time at which process must finish.

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.

Interprocess communication



⌘ **Interprocess communication (IPC)**: OS provides mechanisms so that processes can pass data.

⌘ Two types of semantics:

☐ **blocking**: sending process waits for response;

☐ **non-blocking**: sending process continues.

IPC styles



⌘ Shared memory:

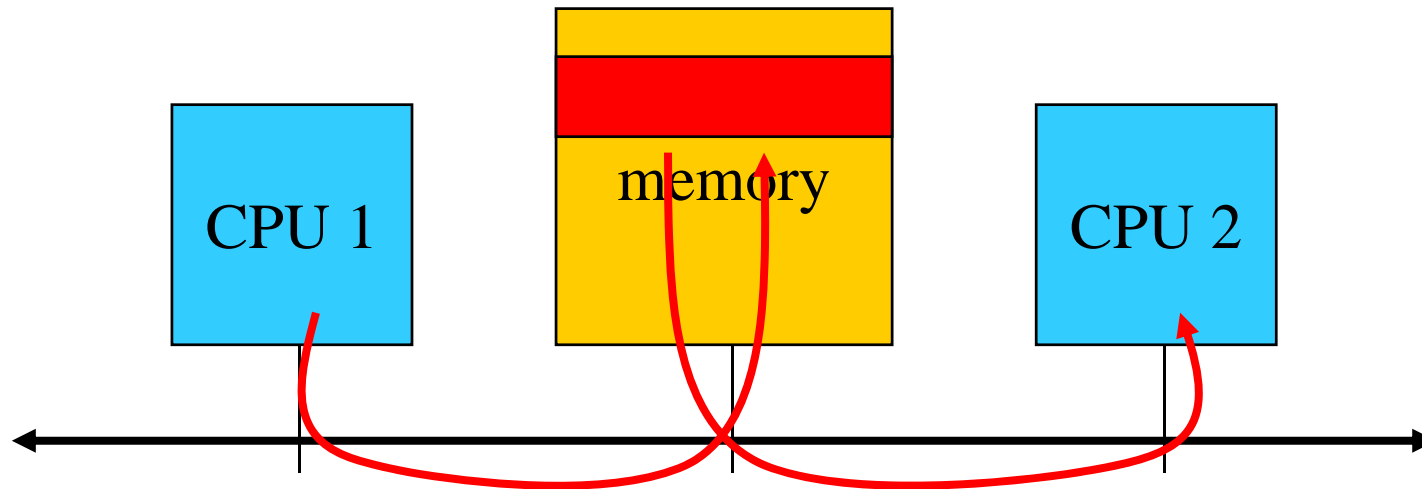
- ☐ processes have some memory in common;
- ☐ must cooperate to avoid destroying/missing messages.

⌘ Message passing:

- ☐ processes send messages along a communication channel---no common address space.

Shared memory

⌘ Shared memory on a bus:



⌘ The fag (additional shared data location)

⏏ 0 if the data (memory) is not in use;

⏏ 1 if the memory is in use.

Race condition in shared memory



- ⌘ Problem when two CPUs try to write the same location:
 1. CPU 1 reads flag and sees 0.
 2. CPU 2 reads flag and sees 0.
 3. CPU 1 sets flag to one and writes location.
 4. CPU 2 sets flag to one and overwrites location.
- ⌘ A critical timing race between 3 and 4 steps.
- ⌘ To avoid this timing race, the microprocessor must support an **atomic test-and-set** operation

Atomic test-and-set



⌘ Problem can be solved with an atomic test-and-set:

☑ single bus operation reads memory location, tests it, writes it.

⌘ ARM test-and-set provided by SWP:

```
ADR r0,SEMAPHORE
LDR r1,#1
GETFLAG SWP r1,r1,[r0]
BNZ GETFLAG
```

Critical regions



⌘ **Critical region**: section of code that cannot be interrupted by another process.

⌘ Examples:

- ☑ writing shared memory;
- ☑ accessing I/O device.

Semaphores

⌘ **Semaphore**: OS primitive for controlling access to critical regions.

⌘ Protocol:

1. **P()**; //Get access to semaphore
2. Perform critical region operations.
3. **V()**; //Release semaphore

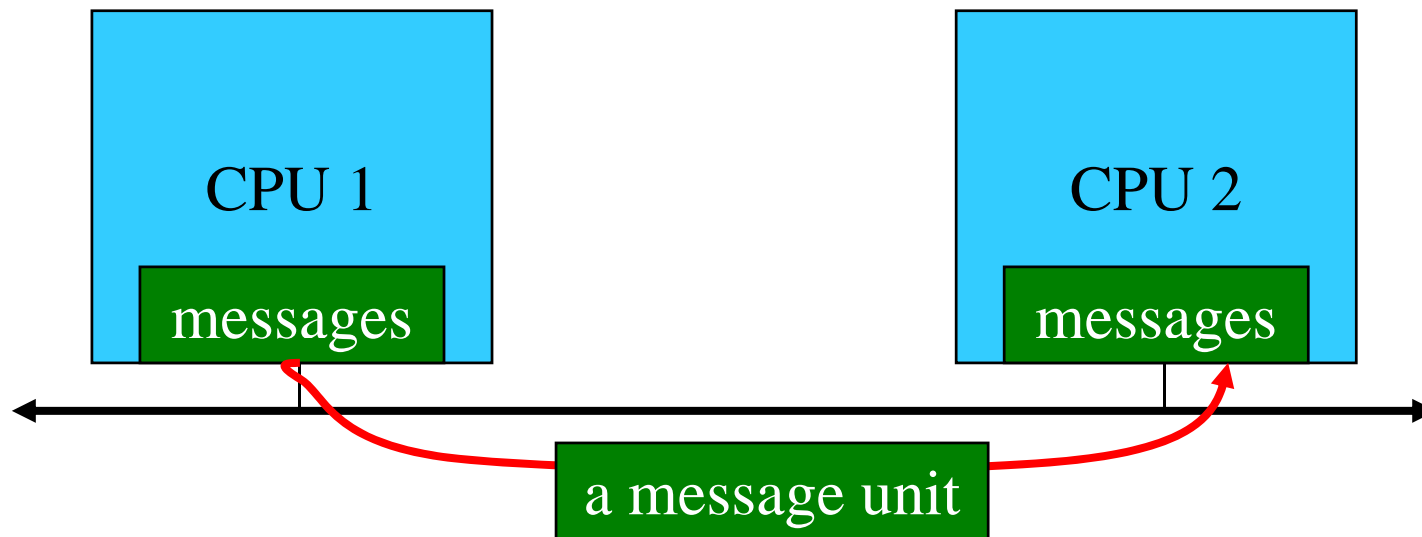
☐ **P()**: use a **test-and-set** to repeatedly test a location that holds a lock on the memory block. access to semaphore

☐ **V()**: reset the lock

Message passing

⌘ Message passing on a network:

- ☑ Messages are stored in the senders/receivers at the end of the link



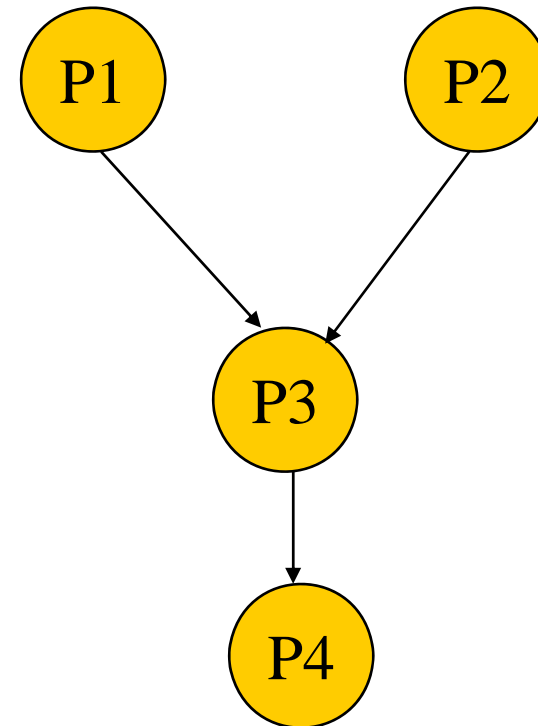
Signals



- ⌘ Another form of interprocess communication commonly used in Unix
- ⌘ A **signal**: simple because it does not pass data beyond the existence of the signal itself.
- ⌘ It is analogous to an interrupt, but it is entirely a software creation. It is generated by a process and transmitted to another process by the OS.

Process data dependencies

- ⌘ One process may not be able to start until another finishes.
- ⌘ Data dependencies defined in a **task graph**.
- ⌘ All processes in one task run at the same rate.



Other OS functions



- ⌘ Date/time.
- ⌘ File system.
- ⌘ Networking.
- ⌘ Security.

Processes and OS



⌘ Scheduling policies:

☐ RMS;

☐ EDF.

⌘ Scheduling modeling assumptions.

Metrics



⌘ How do we evaluate a scheduling policy:

- ☑ Ability to satisfy all deadlines.
- ☑ CPU utilization---percentage of time devoted to useful work.
- ☑ Scheduling overhead---time required to make scheduling decision.

Rate monotonic scheduling



- ⌘ **Static** scheduling policy
- ⌘ **RMS** (Liu and Layland): widely-used, analyzable scheduling policy.
- ⌘ Analysis is known as **Rate Monotonic Analysis (RMA)**.

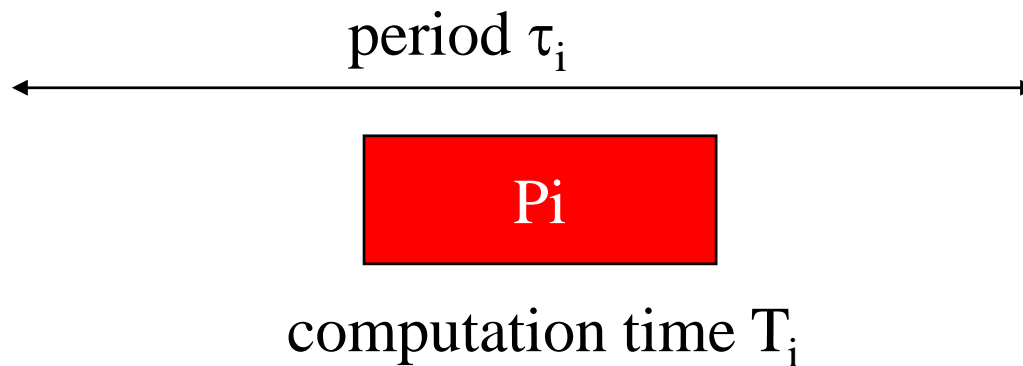
RMA model



- ⌘ All process run on a single CPU.
- ⌘ Zero context switch time.
- ⌘ No data dependencies between processes.
- ⌘ Process execution time is constant.
- ⌘ Deadline is at end of period.
- ⌘ Highest-priority ready process runs.

Process parameters

⌘ T_i is computation time of process i ; τ_i is period of process i .

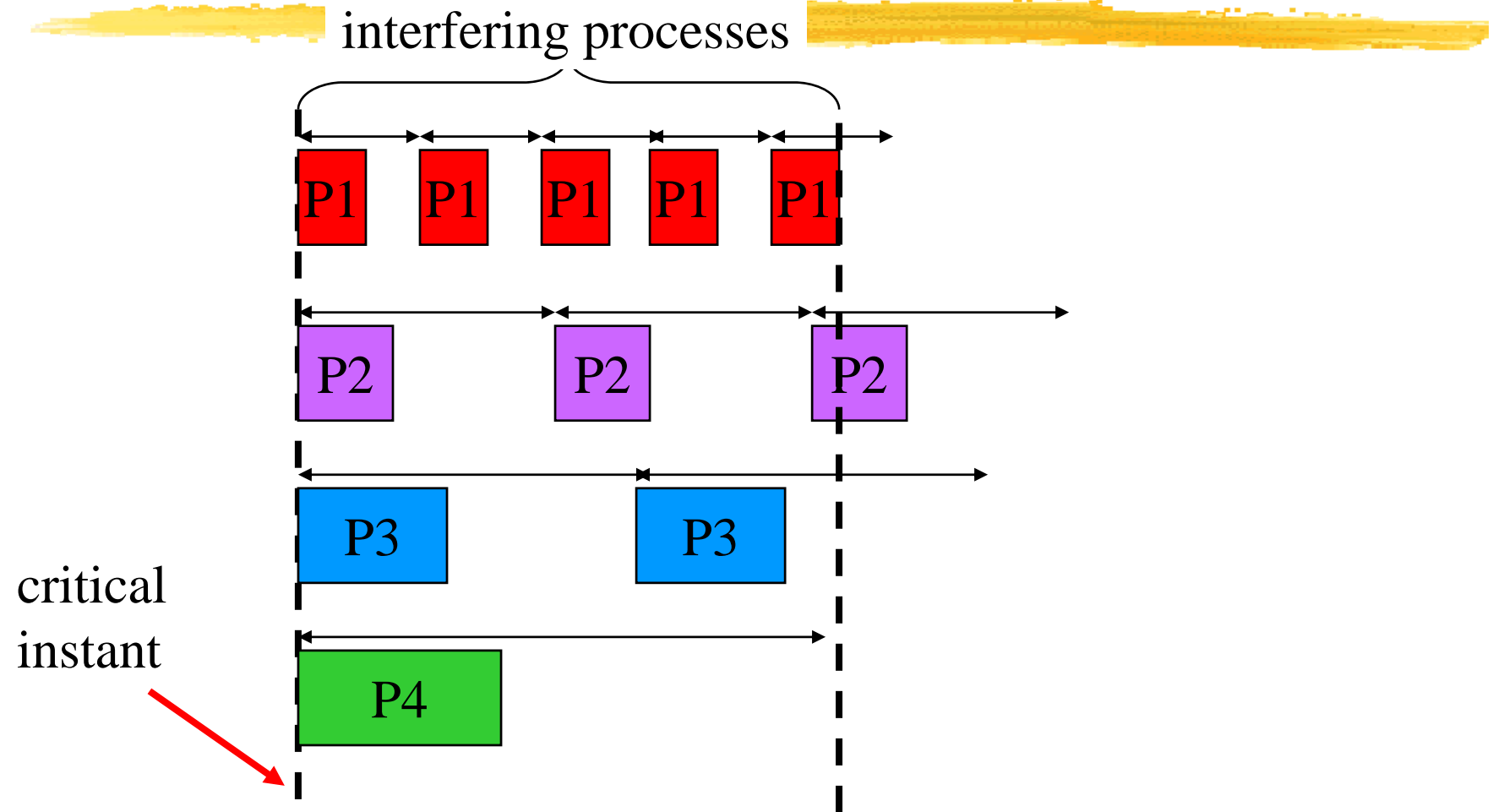


Rate-monotonic analysis



- ⌘ **Response time**: time required to finish process.
- ⌘ **Critical instant**: scheduling state that gives worst response time.
- ⌘ Critical instant occurs when all higher-priority processes are ready to execute.

Critical instant



RMS priorities

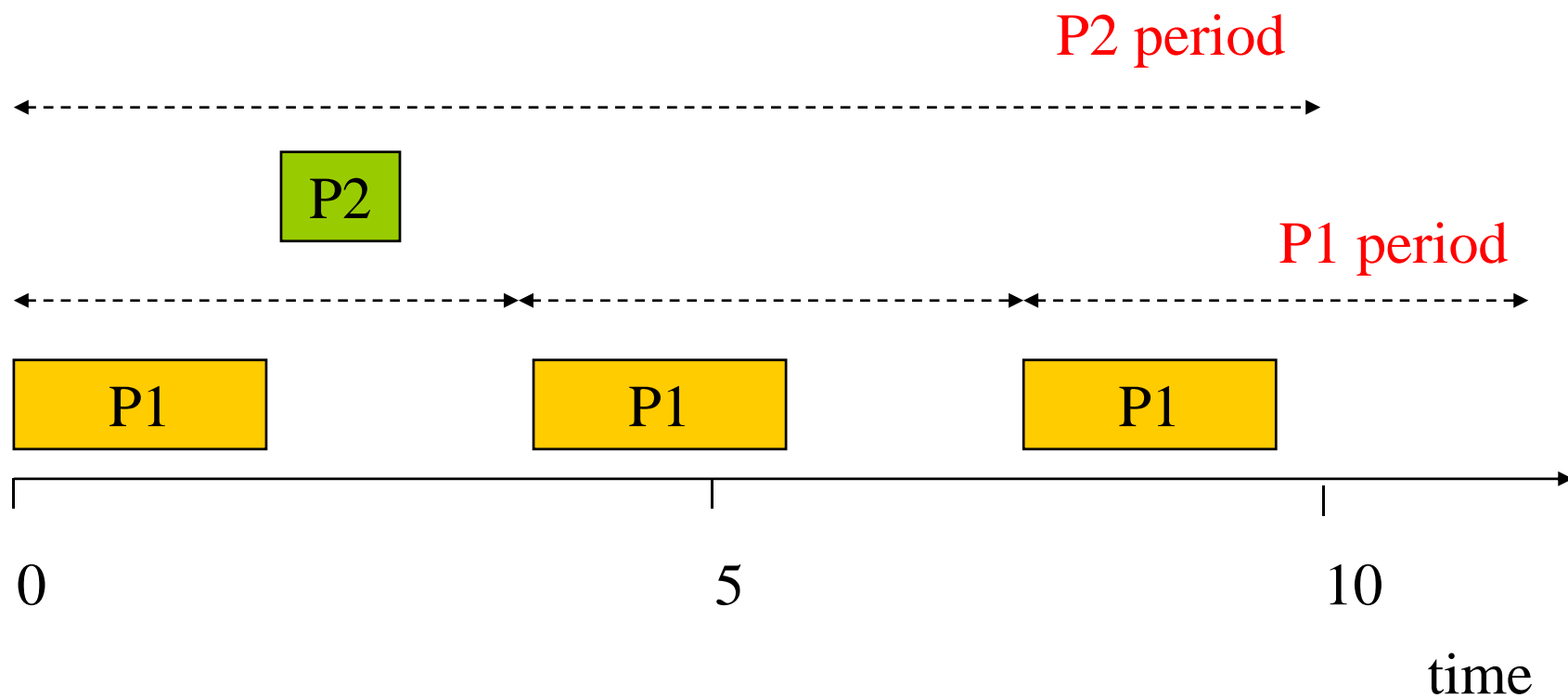


⌘ Optimal (fixed) priority assignment:

- ☑ shortest-period process gets highest priority;
- ☑ priority inversely proportional to period;
- ☑ break ties arbitrarily.

⌘ No fixed-priority scheme does better.

RMS example



RMS CPU utilization



⌘ Utilization for n processes is

$$\boxed{\wedge} \sum_i T_i / \tau_i$$

⌘ $U = m (2^{1/m} - 1)$ for $m = \#$ of tasks

⌘ As number of tasks approaches infinity, maximum utilization approaches 69%.

RMS CPU utilization



- ⌘ RMS cannot use 100% of CPU, even with zero context switch overhead.
- ⌘ Must keep idle cycles available to handle worst-case scenario.
- ⌘ However, RMS guarantees all processes will always meet their deadlines.

RMS implementation



⌘ Efficient implementation: $O(n)$

- ☑ An RMS scheduler runs at the OS's timer interrupt
- ☑ The scheduler scans thru the list of processes;
- ☑ It chooses the highest-priority active process.

Earliest-deadline-first (EDF)



- ⌘ **dynamic** priority scheduling scheme.
- ⌘ Process closest to its deadline has highest priority.
- ⌘ Requires recalculating processes at every timer interrupt.

EDF analysis



- ⌘ EDF can use 100% of CPU.
- ⌘ But EDF may fail to miss a deadline.

EDF implementation



⌘ On each timer interrupt:

- ⏏ compute time to deadline;

- ⏏ choose process closest to deadline.

⌘ Generally considered too expensive to use in practice.

Fixing scheduling problems



⌘ What if your set of processes is unschedulable?

- ☑ Change deadlines in requirements.
- ☑ Reduce execution times of processes.
- ☑ Get a faster CPU.

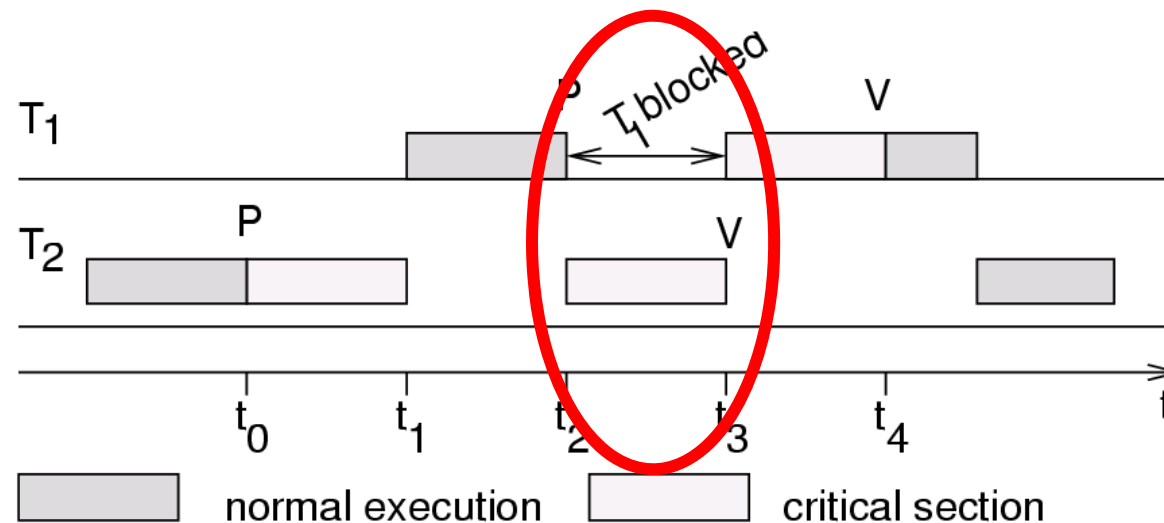
Priority inversion



- ⌘ **Priority inversion**: low-priority process keeps high-priority process from running.
- ⌘ Improper use of system resources can cause scheduling problems:
 - ☐ Low-priority process grabs I/O device.
 - ☐ High-priority device needs I/O device, but can't get it until low-priority process is done.
- ⌘ Can cause deadlock.

Priority inversion

- ⌘ Priority T_1 assumed to be **higher** than priority of T_2 .
- ⌘ If T_2 requests exclusive access first (at t_0), T_1 has to wait until T_2 releases the resource (time t_3), thus inverting the priority:



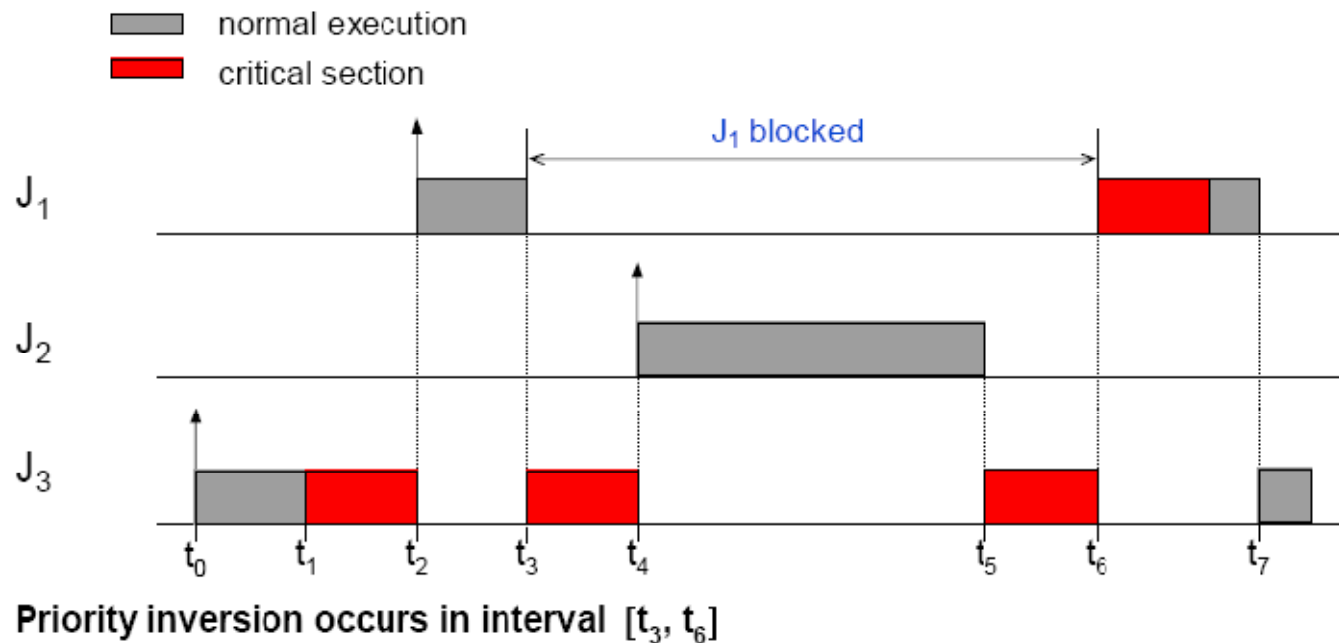
In this example:
duration of inversion bounded by length of critical section of T_2 .

Duration of priority inversion with >2 tasks

maximum blocking time of J_1 = duration of J_2 in critical section

→ unavoidable due to semantics of critical section

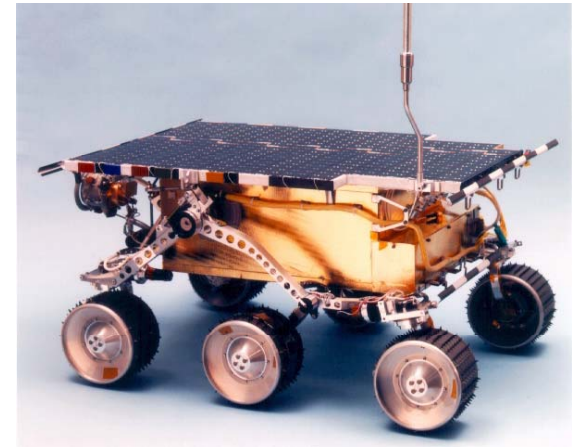
However: blocking time may be unbounded if there are tasks with intermediate priority:



can exceed the length of any critical section!

The MARS Pathfinder problem

⌘ “But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once".” ...



The MARS Pathfinder problem



⌘ “VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”

⌘ “Pathfinder contained an “**information bus**”, which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

- ☒ A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus.

- ☒ Access to the bus was synchronized with mutual exclusion locks (mutexes).”

The MARS Pathfinder problem

- ☒ The meteorological data gathering task ran as an infrequent, low priority thread, ... When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. ..
- ☒ The spacecraft also contained a communications task that ran with medium priority.”



High priority: retrieval of data from shared memory
Medium priority: long-running communications task
Low priority: thread collecting meteorological data

The MARS Pathfinder problem

⌘ “Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, **a watchdog timer would go off**, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion.”

Priority inheritance protocol

- ☒ A task is scheduled according to its active priority.
Tasks with the same priorities are scheduled **FCFS**.
- A task inherits the highest priority from the tasks it blocks. (PIP)
- If task T1 executes **P(S)** but its exclusive access was granted to T2, then T1 will be blocked.
- If $\text{priority}(T2) < \text{priority}(T1)$, then T2 inherits the priority of T1 so that T2 can release the shared resource earlier by preventing medium-priority tasks from preempting T2 and prolonging the blocking period..
- When T2 executes **V(S)**, its original priority at the point of entry of the critical section is restored.

Priority inheritance protocol

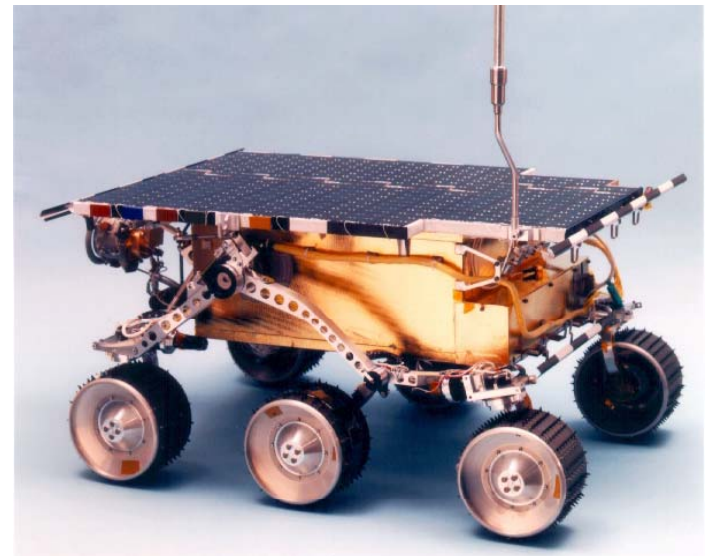


- Priority inheritance is transitive
- Assuming that $\text{priority}(T1) > \text{priority}(T2) > \text{priority}(T3)$
- If $T3$ blocks $T2$ and $T2$ blocks $T1$, then $T3$ inherits the priority of $T1$.

Priority inversion on Mars

- ⌘ Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].



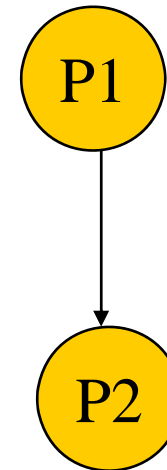
Remarks on priority inheritance protocol



- ⌘ Possible large number of tasks with high priority.
- ⌘ Possible deadlocks.
- ⌘ More sophisticated protocol: priority ceiling protocol.

Data dependencies

- ⌘ Data dependencies allow us to improve utilization.
 - ☑ Restrict combination of processes that can run simultaneously.
- ⌘ P1 and P2 can't run simultaneously.



Context-switching time



- ⌘ Non-zero context switch time can push limits of a tight schedule.
- ⌘ Hard to calculate effects---depends on order of context switches.
- ⌘ In practice, OS context switch overhead is small (hundreds of clock cycles) relative to many common task periods (ms – μ s).

Evaluating RTOS performance

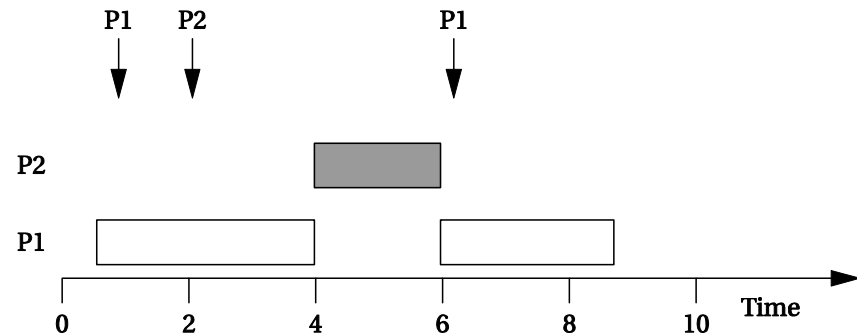


⌘ Simplifying assumptions:

- ☑ Context switch costs no CPU time,.
- ☑ We know the exact execution time of processes.
- ☑ WCET/BCET don't depend on context switches.

Scheduling and context switch overhead

Process	Execution time	deadline
P1	3	5
P2	3	10



With context switch overhead of 1, no feasible schedule.

$$2TP1 + TP2 = 2*(1+3)+(3)=11$$

Process execution time



- ⌘ Process execution time is not constant.
- ⌘ Extra CPU time can be good.
- ⌘ Extra CPU time can also be bad:
 - ☐ Next process runs earlier, causing new preemption.

Processes and caches



⌘ Processes can cause additional caching problems.

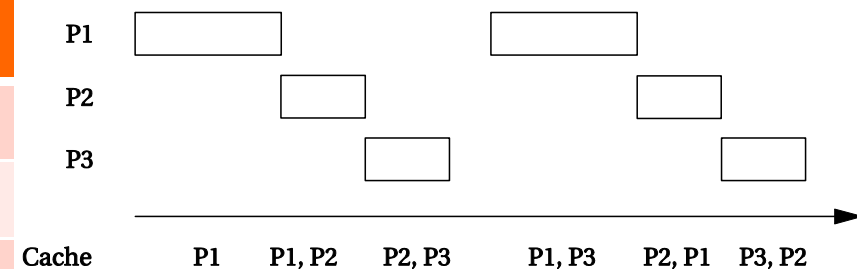
☐ Even if individual processes are well-behaved, processes may interfere with each other.

⌘ Worst-case execution time with bad behavior is usually much worse than execution time with good cache behavior.

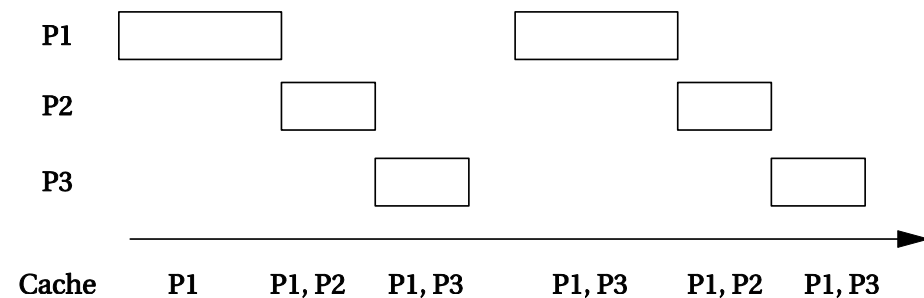
Effects of scheduling on the cache

Process	WCET	Avg. CPU time
P1	8	6
P2	4	3
P3	4	3

Schedule 1 (LRU cache):



Schedule 2 (half of cache reserved for P1):



- ⌘ Each process uses half the cache
- ⌘ Schedule 1: P1 is the worst case for the 2nd iteration
- ⌘ Schedule 2: P1 is the average case for the 2nd iteration

Power optimization



- ⌘ **Power management**: determining how system resources are scheduled/used to control power consumption.
- ⌘ OS can manage for power just as it manages for time.
- ⌘ OS reduces power by shutting down units.
 - ☑ May have partial shutdown modes.

Simple power management policies



⌘ **Request-driven**: power up once request is received. Adds delay to response.

⌘ **Predictive shutdown**: try to predict how long you have before next request.

☐ May start up in advance of request in anticipation of a new request.

☐ If you predict wrong, you will incur additional delay while starting up.

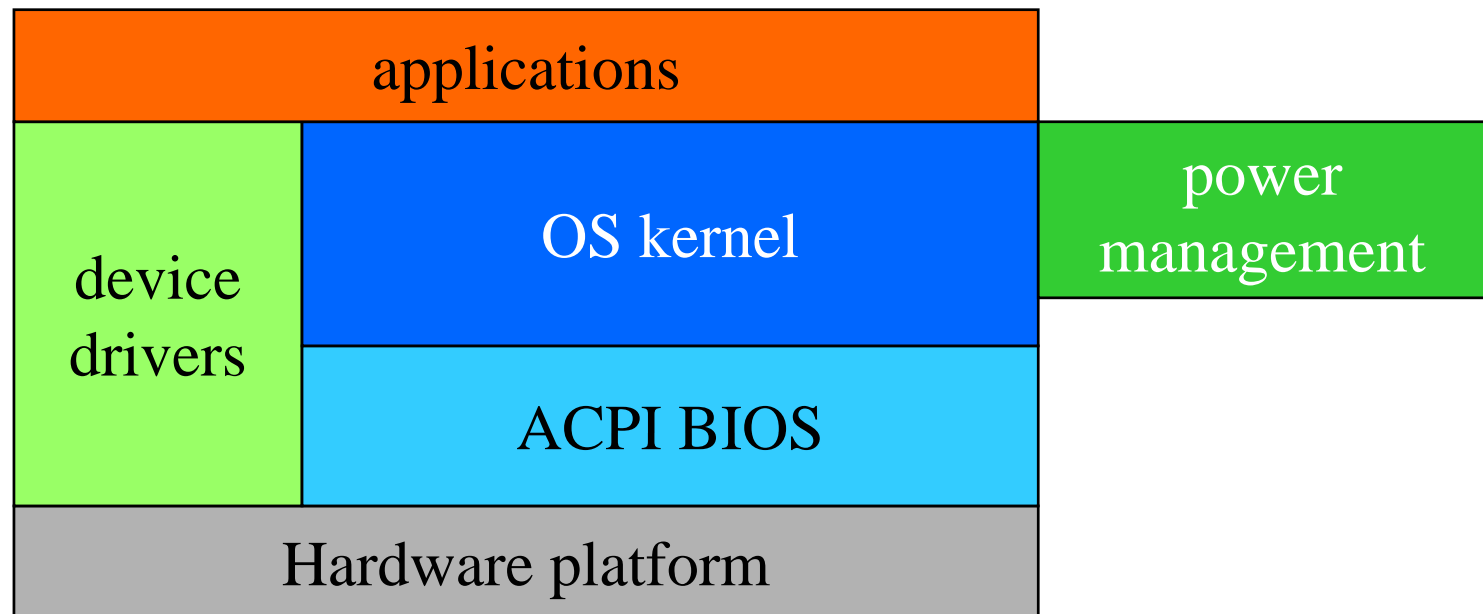
Probabilistic shutdown



- ⌘ Assume service requests are probabilistic.
- ⌘ Optimize expected values:
 - ☐ power consumption;
 - ☐ response time.
- ⌘ Simple probabilistic: shut down after time T_{on} , turn back on after waiting for T_{off} .

Advanced Configuration and Power Interface

⌘ **ACPI**: open standard for power management services.



ACPI global power states



⌘ G3: mechanical off

⌘ G2: soft off

☒ S1: low wake-up latency with no loss of context

☒ S2: low latency with loss of CPU/cache state

☒ S3: low latency with loss of all state except memory

☒ S4: lowest-power state with all devices off

⌘ G1: sleeping state

⌘ G0: working state

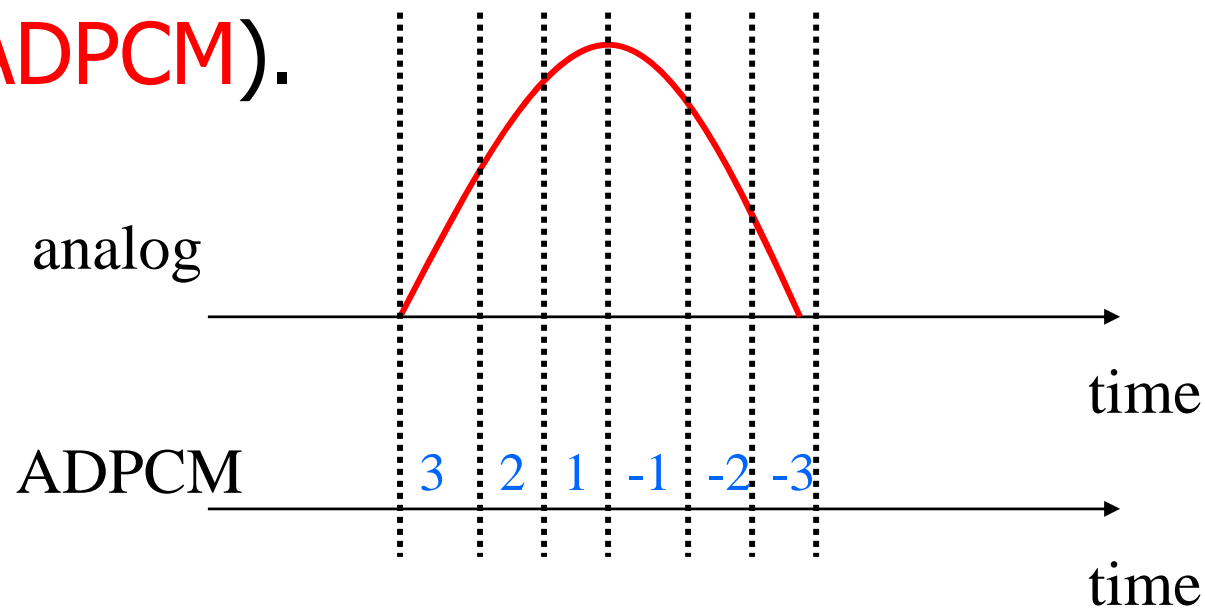
Processes and operating systems



⌘ Telephone answering machine.

Theory of operation

⌘ Compress audio using **adaptive differential pulse code modulation (ADPCM)**.



ADPCM coding

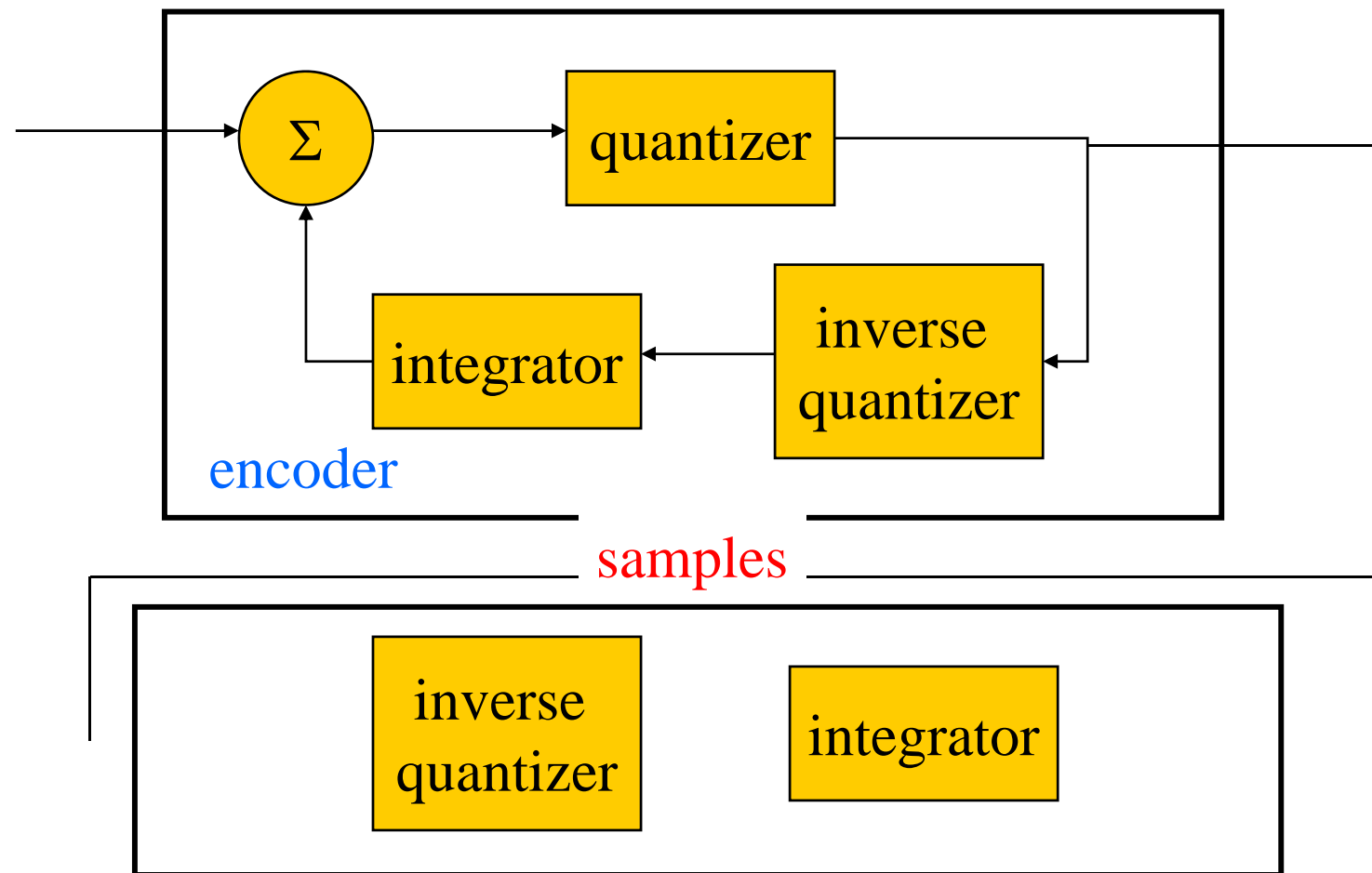


⌘ Coded in a small alphabet with positive and negative values.

☐ $\{-3, -2, -1, 1, 2, 3\}$

⌘ Minimize error between predicted value and actual signal value.

ADPCM compression system



Telephone system terms




⌘ **Subscriber line**: line to phone.

⌘ **Central office**: telephone switching system.

⌘ **Off-hook**: phone active.

⌘ **On-hook**: phone inactive.

Real and simulated subscriber line



⌘ Real subscriber line:

- ☒ 90V RMS ringing signal;
- ☒ companded analog signals;
- ☒ lightning protection, etc.

⌘ Simulated subscriber line:

- ☒ microphone input;
- ☒ speaker output;
- ☒ switches for ring, off-hook, etc.

Requirements

Inputs

Telephone: voice samples, ring.
User interface: microphone, play messages button, record OGM button.

Outputs

Telephone: voice samples, on-hook/off-hook command.
User interface: speaker, # messages indicator, message light.

Functions

Default mode: detects ring, signals off-hook, plays OGM, records ICM
Playback: play all messages, wait 5 seconds for new playback.
OGM editing: OGM up to 10 sec.
About 30 minutes voice (@ 8kHz).

Performance

Manufacturing cost

Consumer product range (\$50)

Power

AC plug

Physical

size/weight

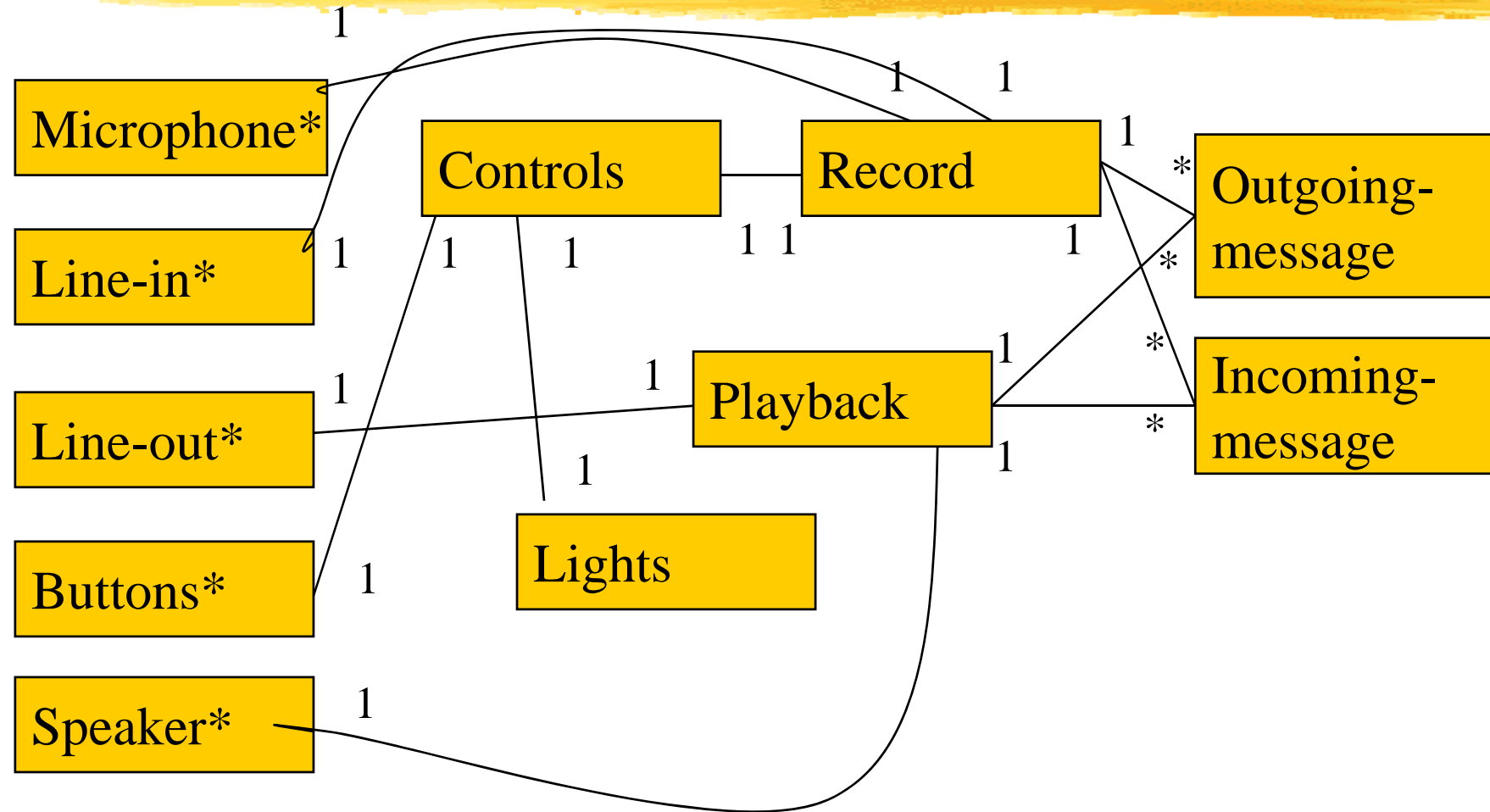
Comparable to desk phone.

Comments on analysis

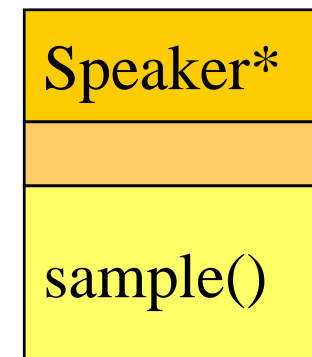
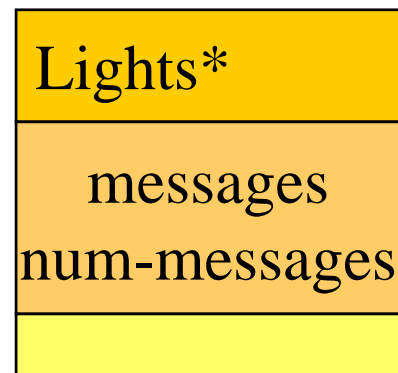
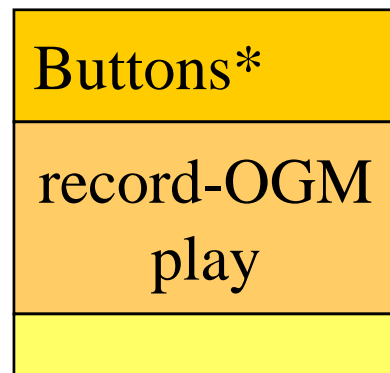
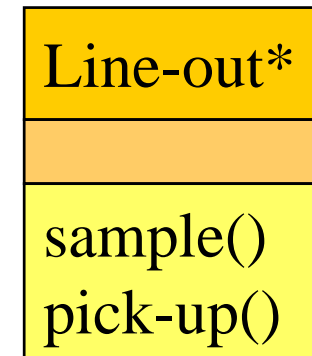
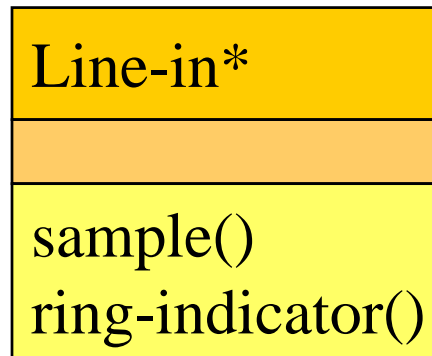
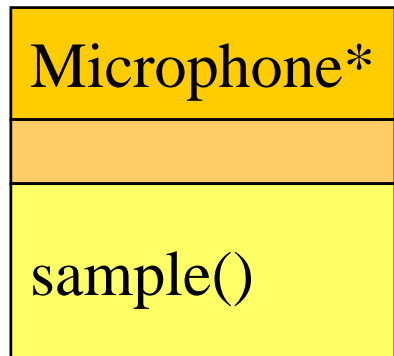


- ⌘ DRAM requirement influenced by DRAM price.
- ⌘ Details of user interface protocol could be tested on a PC-based prototype.

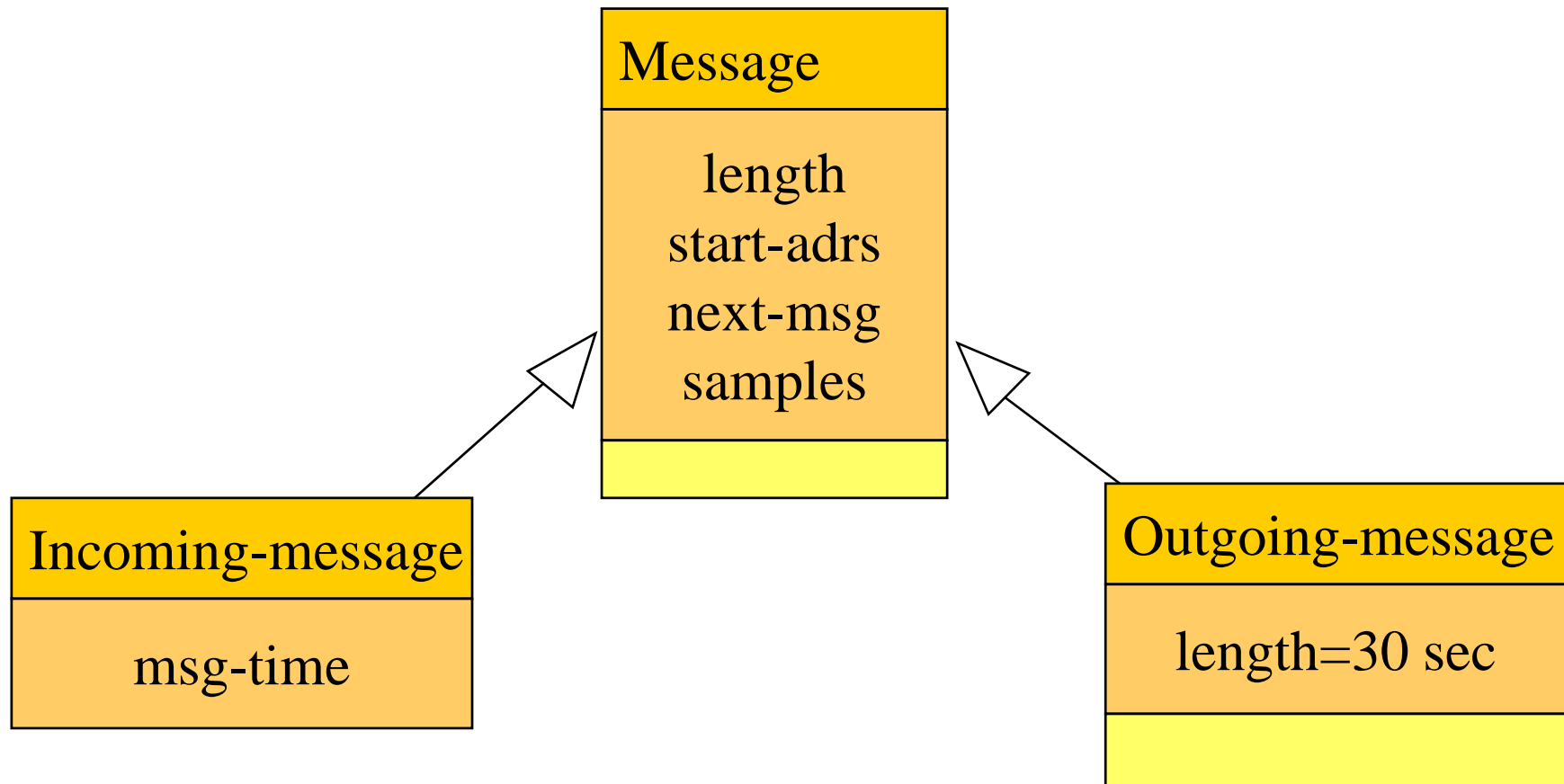
Answering machine class diagram



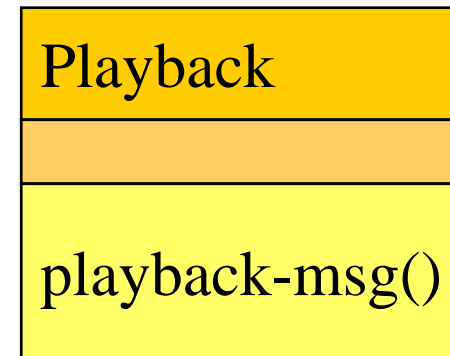
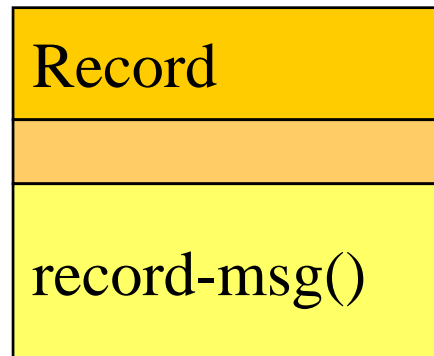
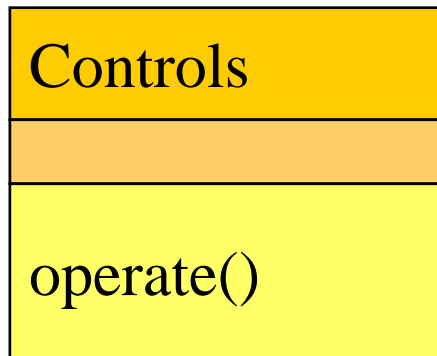
Physical interface classes



Message classes



Operational classes

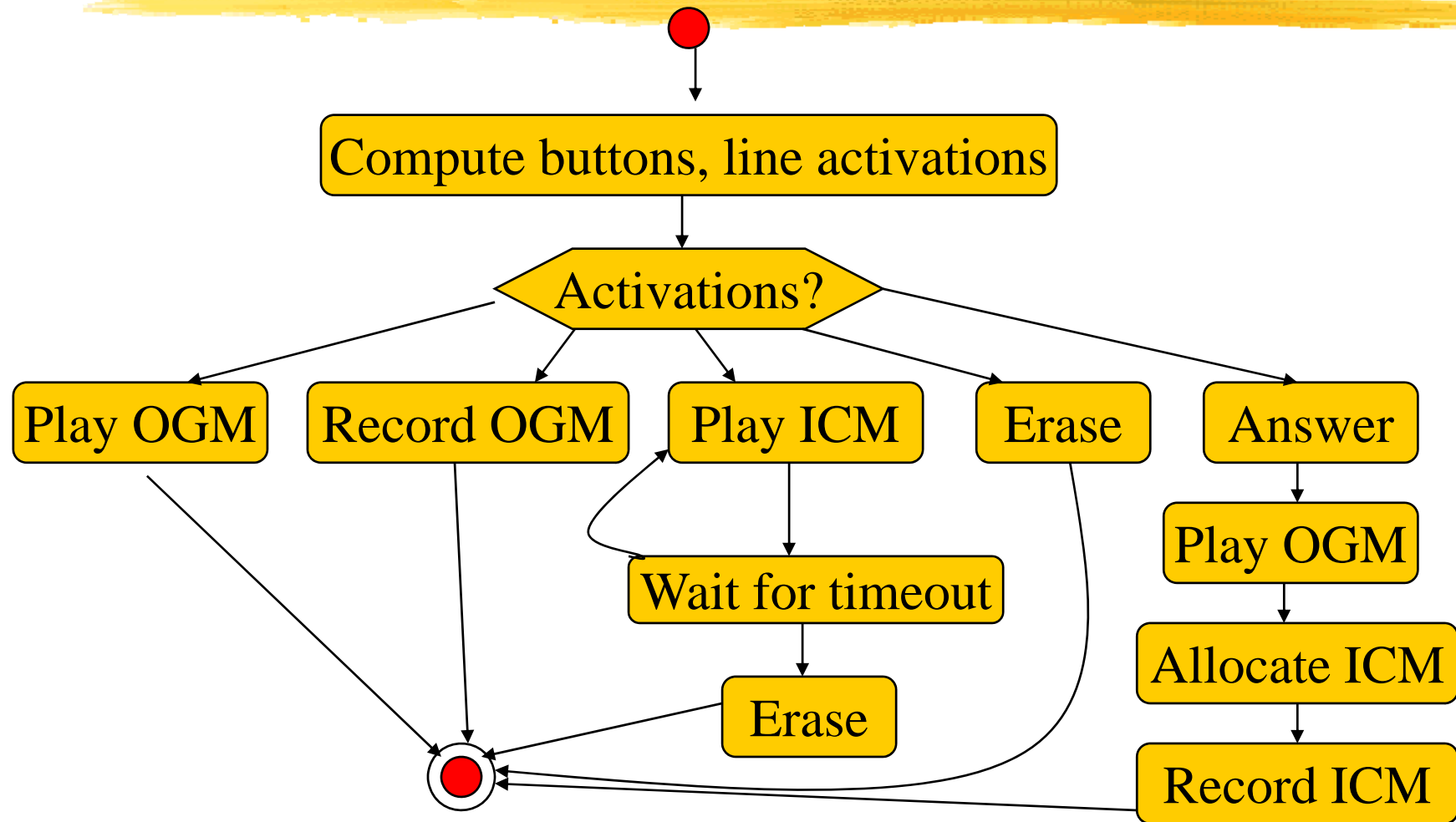


Software components

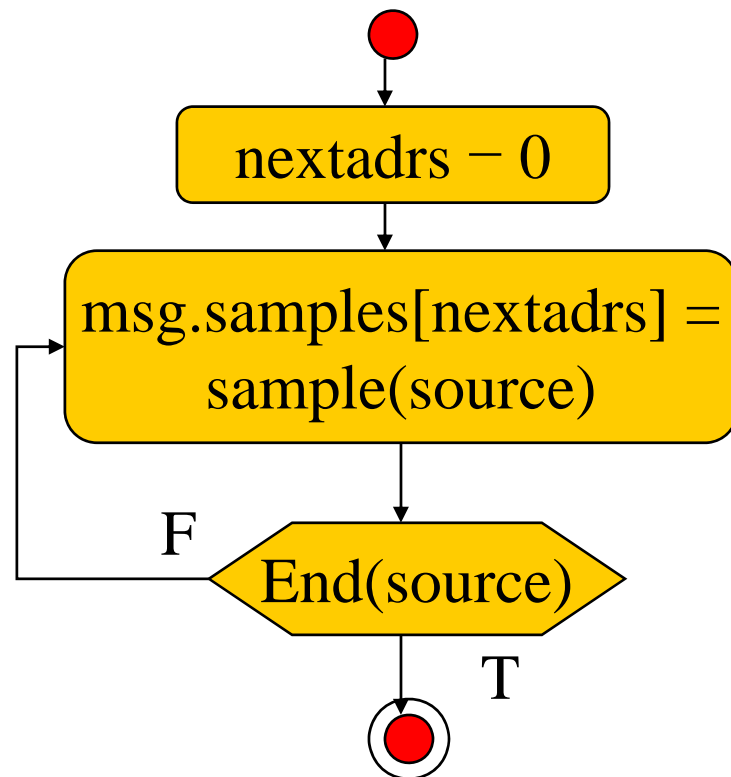


- ⌘ Front panel module.
- ⌘ Speaker module.
- ⌘ Telephone line module.
- ⌘ Telephone input and output modules.
- ⌘ Compression module.
- ⌘ Decompression module.

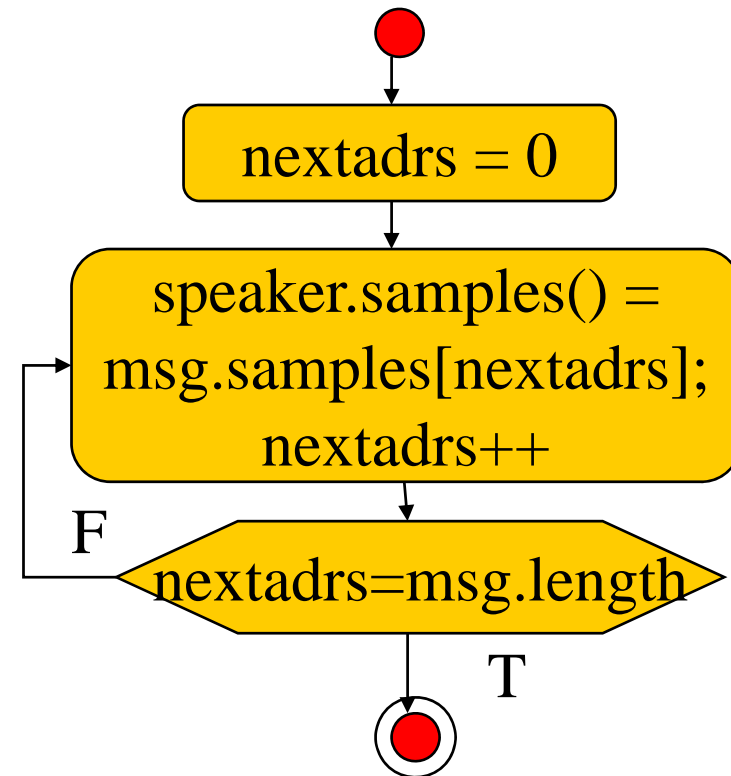
Controls activate behavior



Record-msg/playback-msg behaviors



record-msg



playback-msg

Hardware platform



- ⌘ CPU.

- ⌘ Memory.

- ⌘ Front panel.

- ⌘ 2 A/Ds:

 - ☑ subscriber line, microphone.

- ⌘ 2 D/A:

 - ☑ subscriber line, speaker.

Component design and testing



⌘ Must test performance as well as testing.

- ☑ Compression time shouldn't dominate other tasks.

⌘ Test for error conditions:

- ☑ memory overflow;

- ☑ try to delete empty message set, etc.

System integration and testing



- ⌘ Can test partial integration on host platform; full testing requires integration on target platform.
- ⌘ Simulate phone line for tests:
 - ☑ it's legal;
 - ☑ easier to produce test conditions.