Operating Systems, Concurrency and Time

concepts: tasks, threads, processes

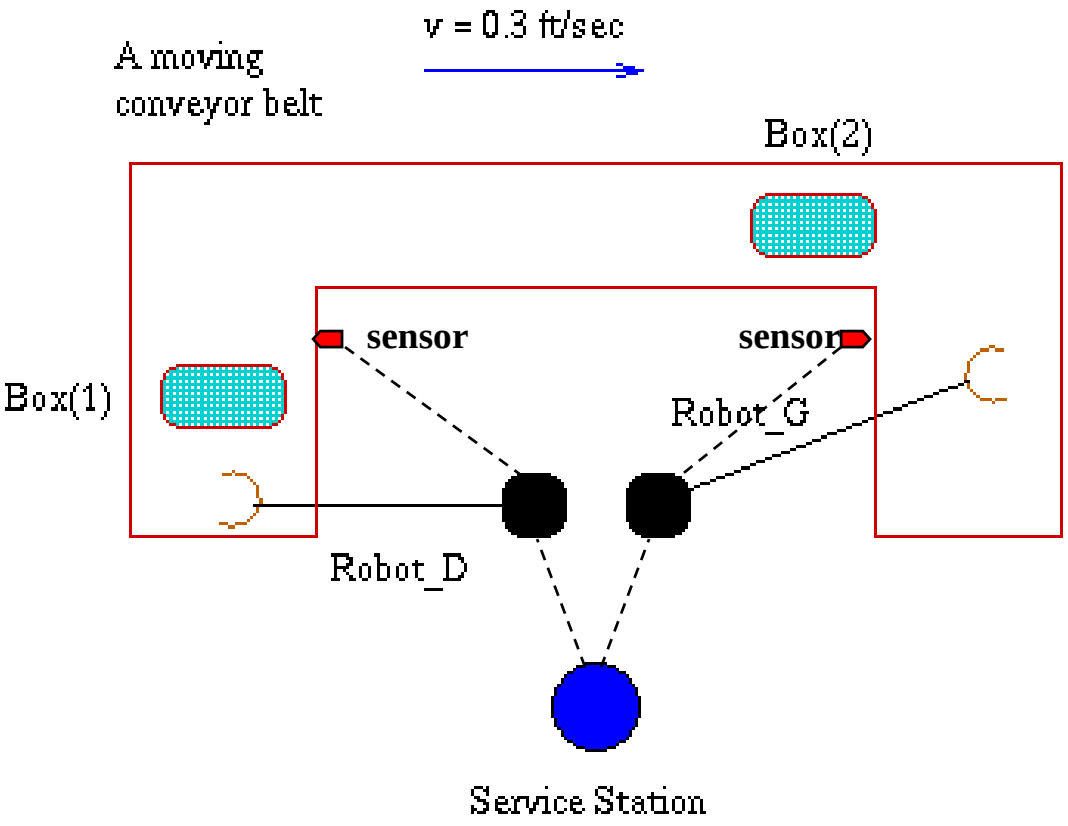
Johan Lukkien

Questions

- What is a real-time system and how is it structured?
- What are examples?
- What are relevant concepts?

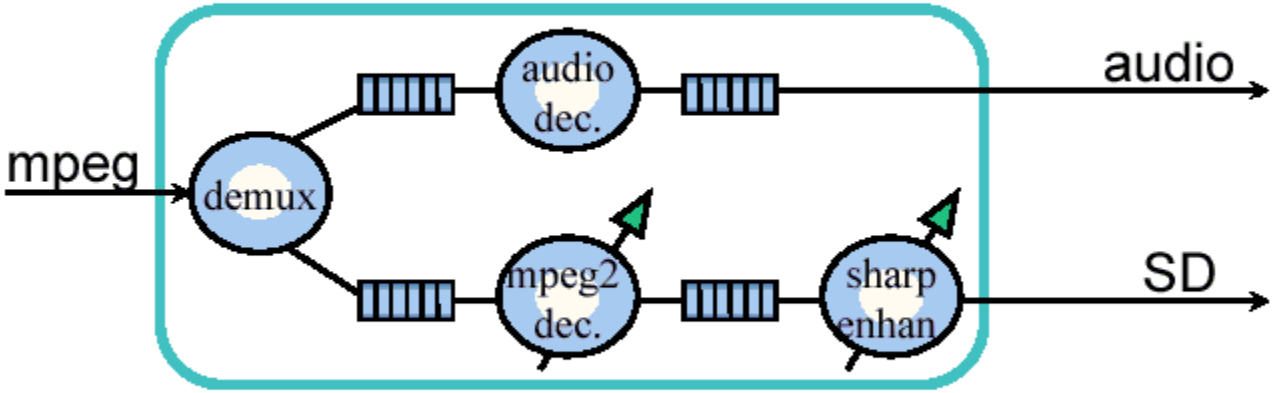
- A real-time system is a system in which correct operation depends not only on logical results of computation of values but also on the time these results are produced
    - *Stankovic in Byte, '92*

Example


v = 0.3 ft/sec
A moving
conveyor belt
Box(2)
sensor
sensor
Box(1)
Robot_G
Robot_D
Service Station

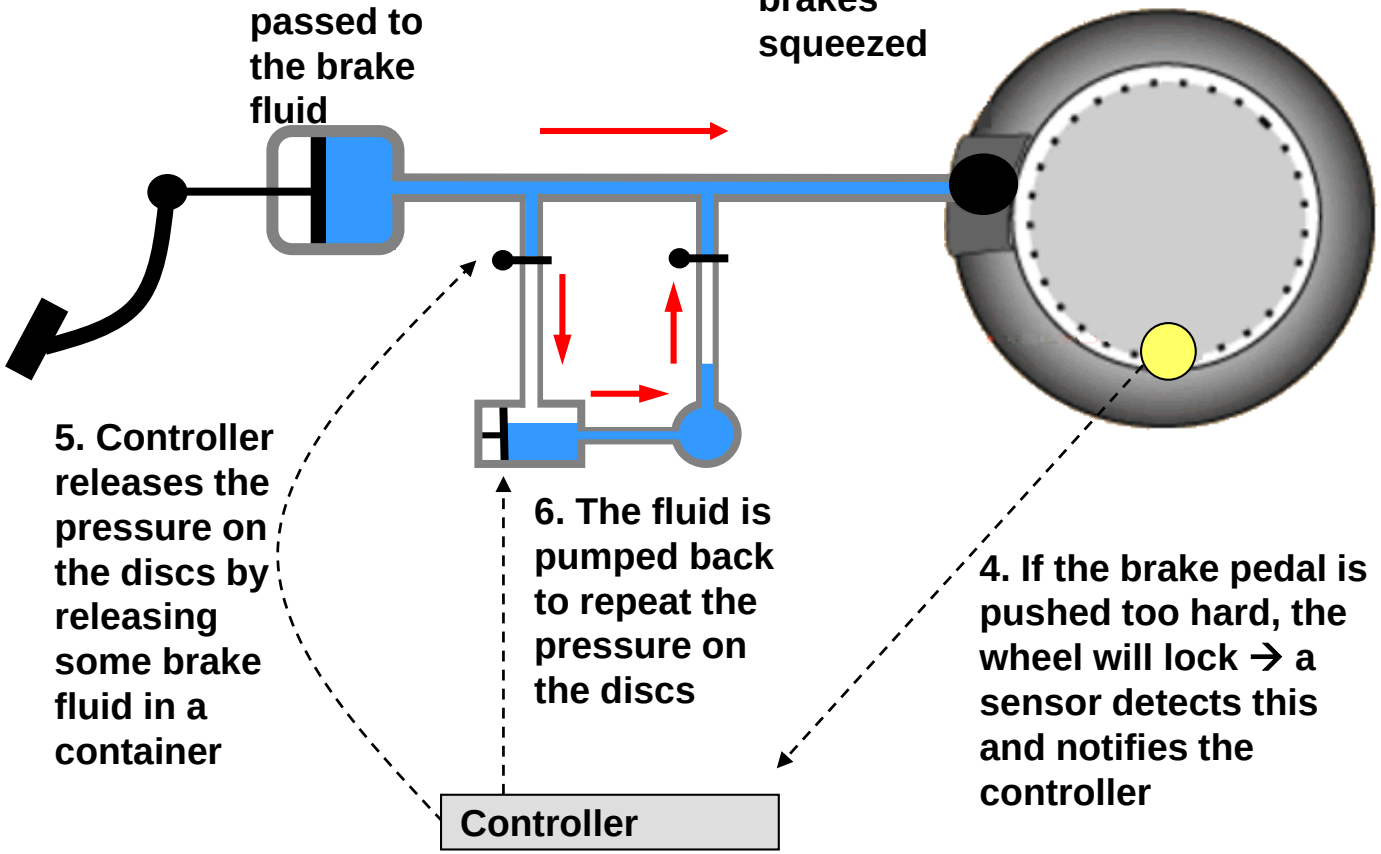
Example: video processing

- Various periodic tasks in a pipeline or tree
- Periods can be different
  - compare audio with video decoding



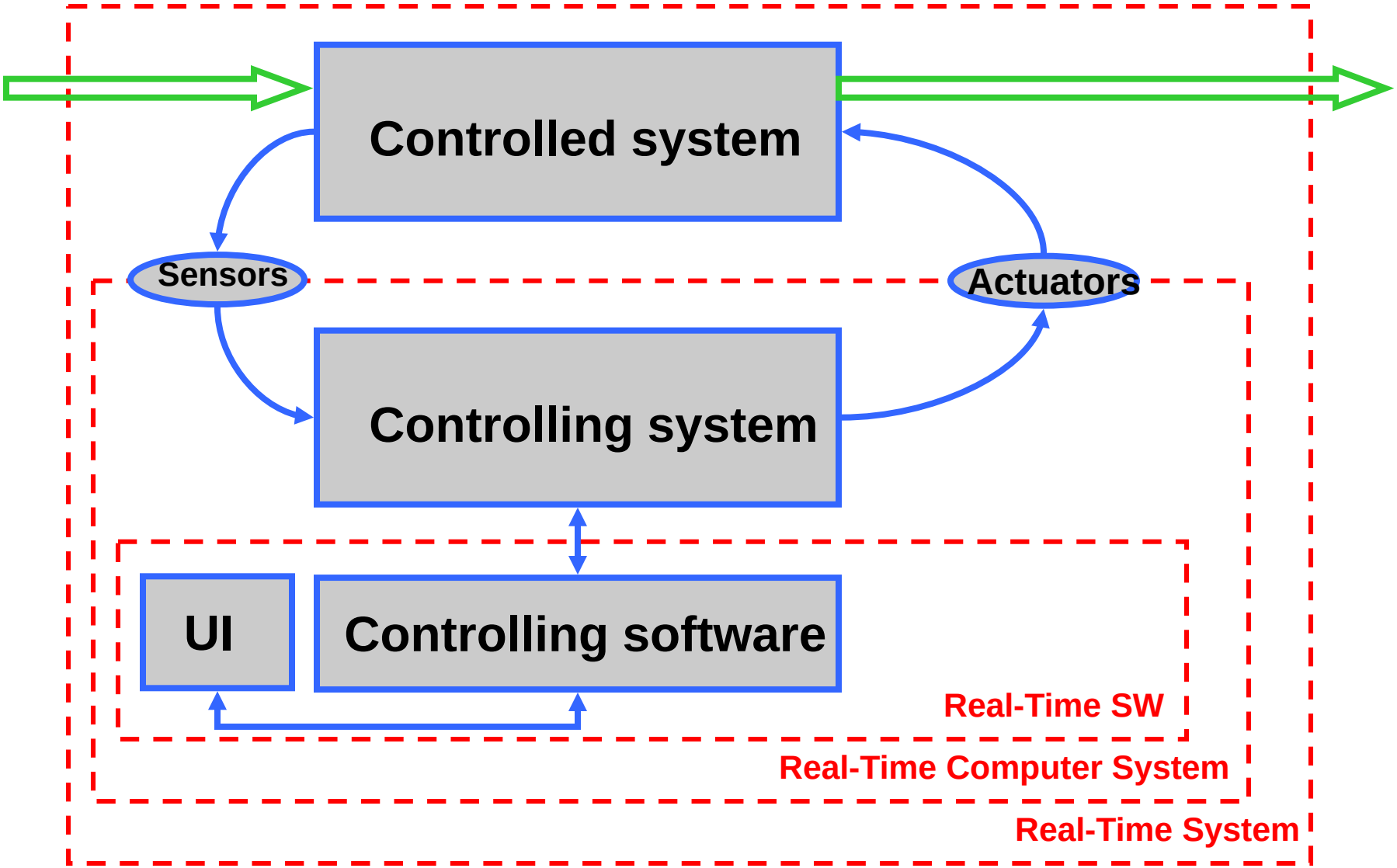
Example: Anti-lock Braking System (ABS)

(by courtesy of Damir Isovic Mälardalen University)



7. Entire process is repeated about 15 times/sec

RT(C)S components



Timeliness requirements: first source

- Real-time constraints that are part of the functional specification of the controlled system (and translate into real-time constraints on controlling system)

- Two flavors:
  - Behavioural aspects of *system* (no hard deadline)
    - "Lift doors close after 5 seconds"
    - "The test alarm sounds at 12 o'clock"
  - Strict (hard deadline)
    - "The alarm sounds within 5 msec. after pressing the button"

Exercise

- Which of the following are behavioral constraints and which are deadline constraints:
  - The class starts at nine o'clock
  - Everybody should be present before five minutes to nine
  - After one hour we should stop for a break
  - The coffee should be ready before the break
  - The instructor should not talk more than 7 hours
  - The course finishes within 8 hours
  - Everybody wants to be home by five o'clock

Timeliness requirements: second source

- *Derived* constraints on *controlling system* (from functional behaviour) – determined by design choices in the *system*
  - "Not more that 5% of the paint is wasted"
    - hence, opening the faucet must be precise enough w.r.t. speed of conveyor belt:
    - speed of conveyor belt is a system design decision following productivity requirements
  - "Variations in temperature are within 0.05K"
    - hence, response must be quick enough w.r.t. possible variations in environment and size and thermal isolation of system
    - size and thermal isolation are system design decisions
  - "K items must be packed in each box"
    - hence response must be quick enough w.r.t. transport speed of items to pick one
    - transport speed is a system design decision

Two classes of systems

- Dependable real-time systems
  - High cost of failure
    - Possibly loss of life on failure
  - Guaranteed dependability (especially timeliness)
  - Over-dimensioning for *worst-case*
  - Example: Industrial control

- High performance real-time systems
  - Low probability of failure
    - Constant quality of service
  - High regularity in performance
  - Aim at *average case*
  - Example: Consumer electronics

Tasks

- Real-time designs commonly result in a collection of tasks, activated by a timer or an event ('trigger')

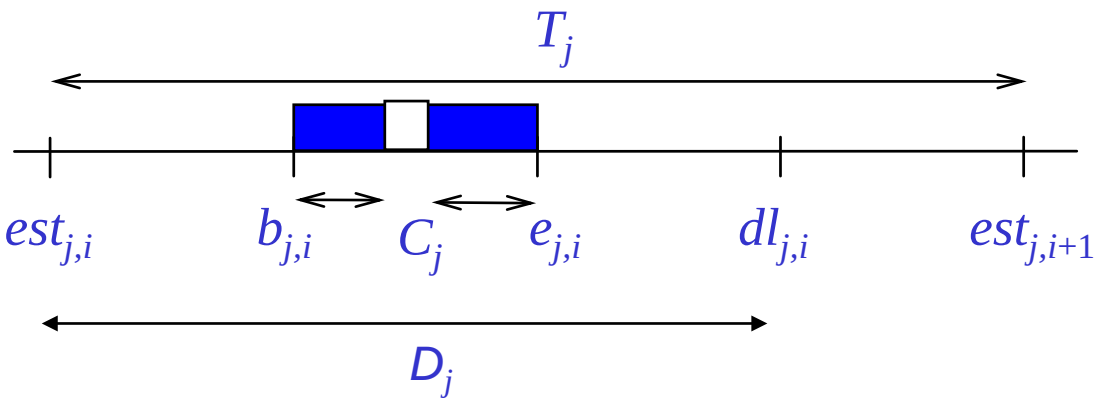
- *A task is a sequence of actions that the software performs to generate a response to an external event or a time event*
  - the complete sequence, possibly performed by several processes and processors together

- An execution of a task is a *job*
  - task: 'procedure'
  - job: 'call', the sequence of actions to respond to a *particular* trigger

- A task starts in the specification domain. It needs
  - implementation (code) and mapping to a platform
  - verification that this implementation satisfies the timing properties of the task

Task Attributes

- A task has static attributes
  - a name (the j th task) τ_j
  - a (worst case, platform dependent) execution time C_j
  - a period (periodic and sporadic tasks) T_j
  - a phasing (periodic tasks: earliest start time of job 0) ϕ_j
  - a relative deadline D_j
- A job i has dynamic (runtime defined) attributes :
  - earliest-start-time (or release time or arrival time) $est_{j,i}$
  - an absolute deadline $dl_{j,i}$
  - a start time (or beginning time) $b_{j,i}$
  - an end time $e_{j,i}$



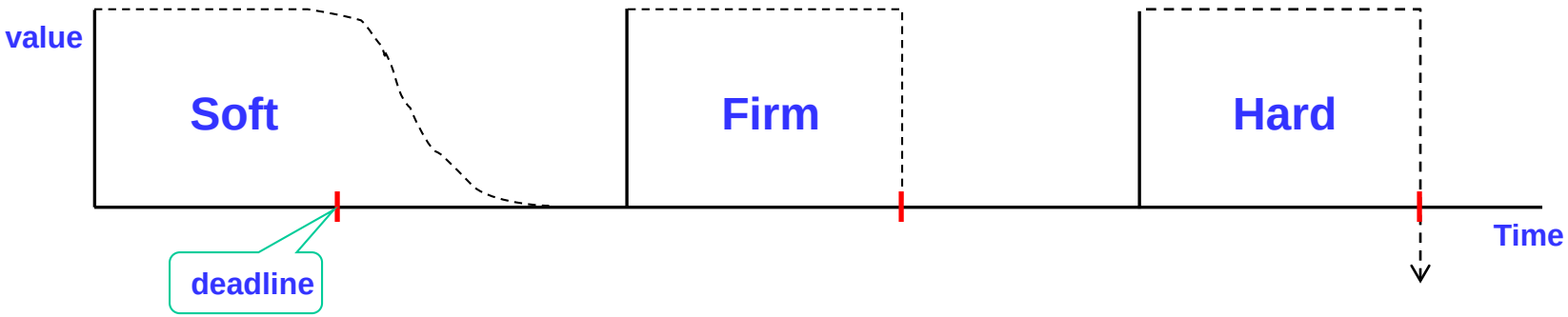
Derived attributes

- $U_j = C_j / T_j$ — utilization of task *j*

- $L_{j,i} = e_{j,i} - dl_{j,i}$ — lateness
- $E_{j,i} = \mathbf{max}\ \{0, L_{j,i}\}$ — tardiness (active after deadline)
- $X_{j,i} = dl_{j,i} - est_{j,i} - C_j$ — slack time
- $R_{j,i} = e_{j,i} - est_{j,i}$ — response time, turnaround time

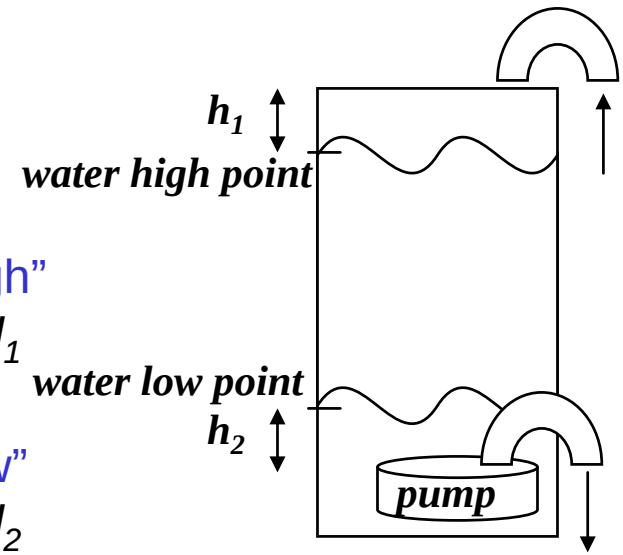
Value after deadline

- **Soft**
  - A response is still valuable after the deadline, but value decreases steadily after that.
    - Example: interaction with human users. People get impatient.
    - Typically: deadline miss must not happen often, and not be extreme
- **Firm**
  - A response has no value after the deadline.
    - Example: a video frame that cannot be shown in time can be skipped.
- **Hard**
  - Damage is done if a response does not come in time.



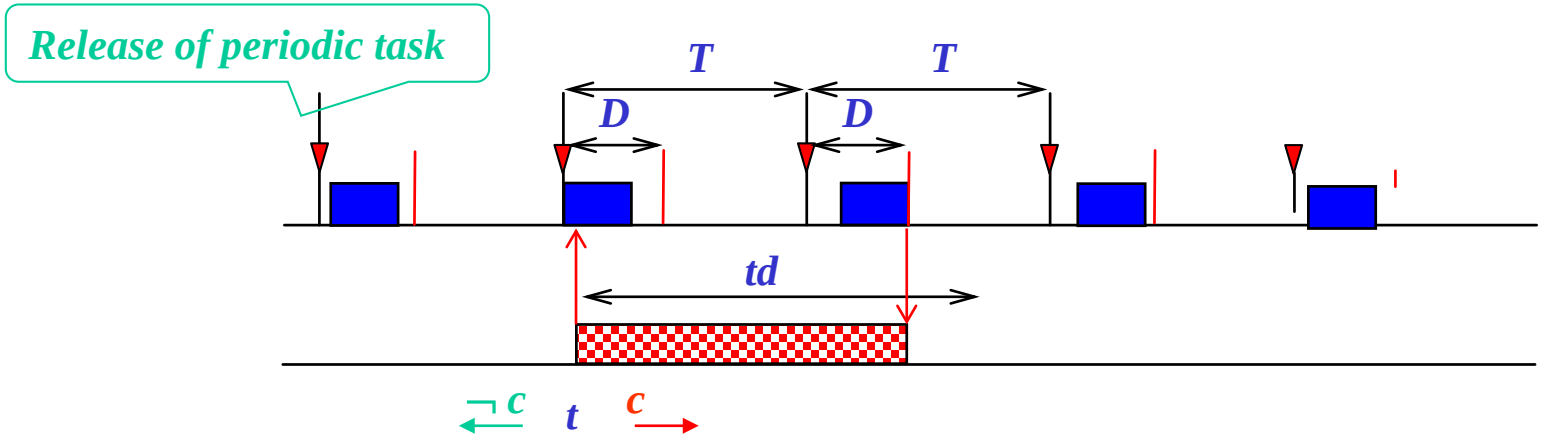
Example: water vessel

- Requirements:
  - Vessel may not overflow
  - Pump may not run dry
- Properties (environment model)
  - Water level has a maximum rate of change (up and down)
  - Sensor positions are chosen
- Derived SW requirements
  - Response deadline can be calculated
    - The state "pump is off and the water is high" may not persist longer than a time span td_1
      - **only then risk of overflow**
    - The state "pump is on and the water is low" may not persist longer than a time span td_2
      - **only then risk of underflow**



Polling task

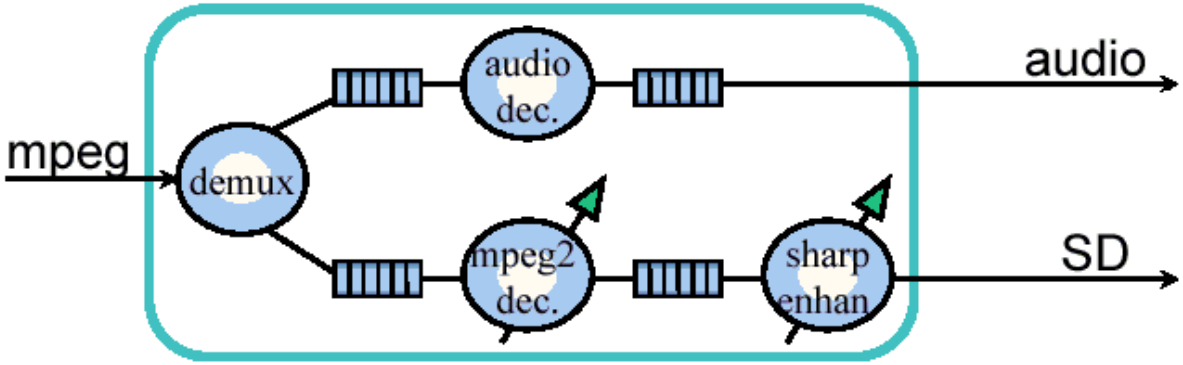
- Critical state c should not exist longer than a time span td without response that cancels this critical state
  - critical state c: water above/below sensor and pump off/on
- Periodic task is released with period T and satisfies deadline D within this period.
  - If water at (around) low sensor: Task stops pump
  - If water at (around) high sensor: Task starts pump
- Schedulability conditions (see diagram): $T + D < td$
  - If the task may finish anywhere within the period ($D = T$) : $2T < td$



Example: video processing

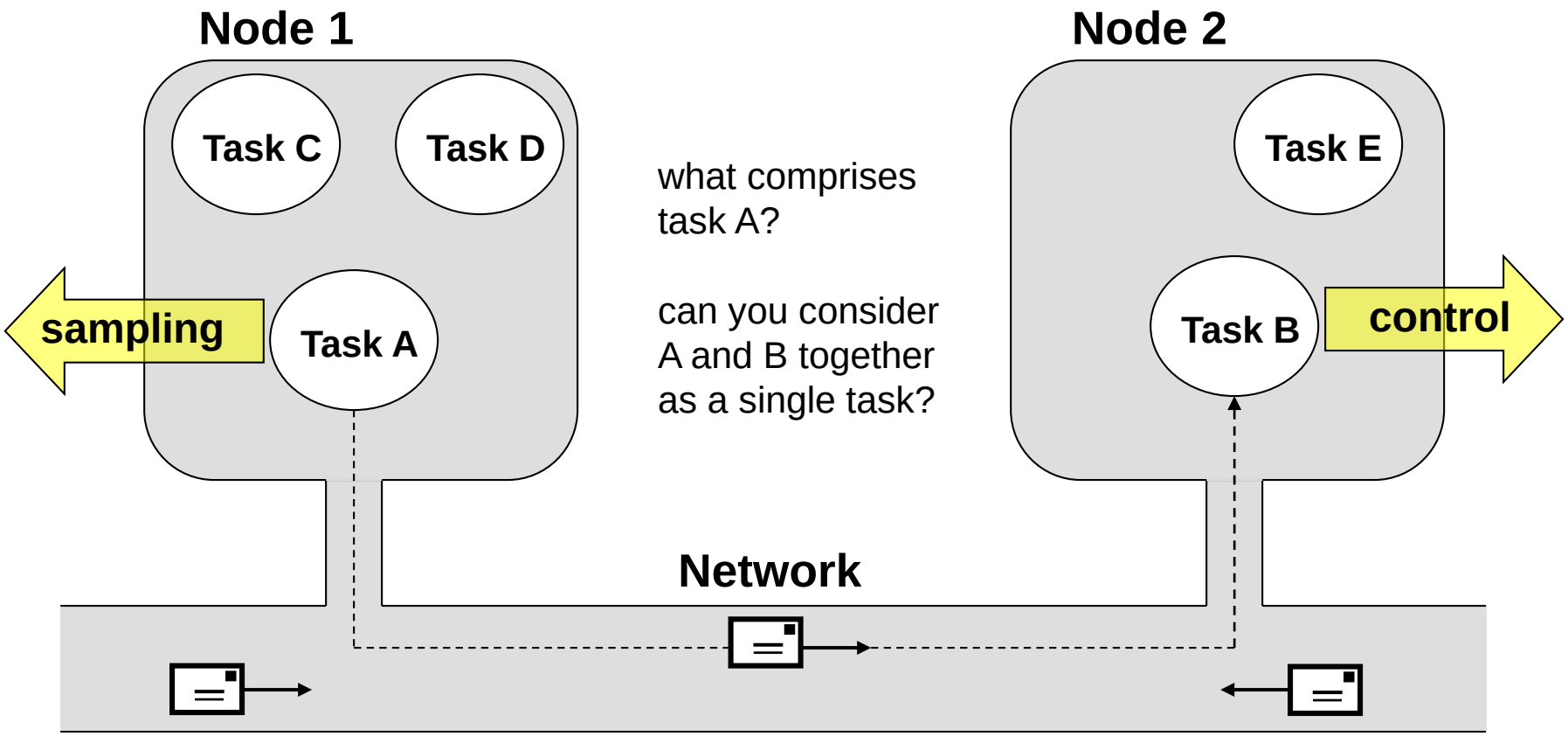
- System timing requirements include the following.
  - Task: full frame processing until delivery
  - Activation: timer or frame arrival
  - A basetime is set when the first frame is output. Deadlines of subsequent jobs equal:

basetime + jobnumber x frametime (e.g. 40msec @ 25fps)



  - Audio is sync'ed with the video at playout

Execution of tasks can be distributed



what comprises task A?

can you consider A and B together as a single task?

Courtesy of Damir Isovic

Exercise

- Form teams of two, for the duration of the course. Maintain a notebook for experiments and exercises. Always make a few slides of your results.

- Look at the project / system you are working on
  - what are timing concerns?
  - specify tasks and how they are activated.
  - what are relevant attributes?
  - does it fit the RTCS model, can you recognize the components?

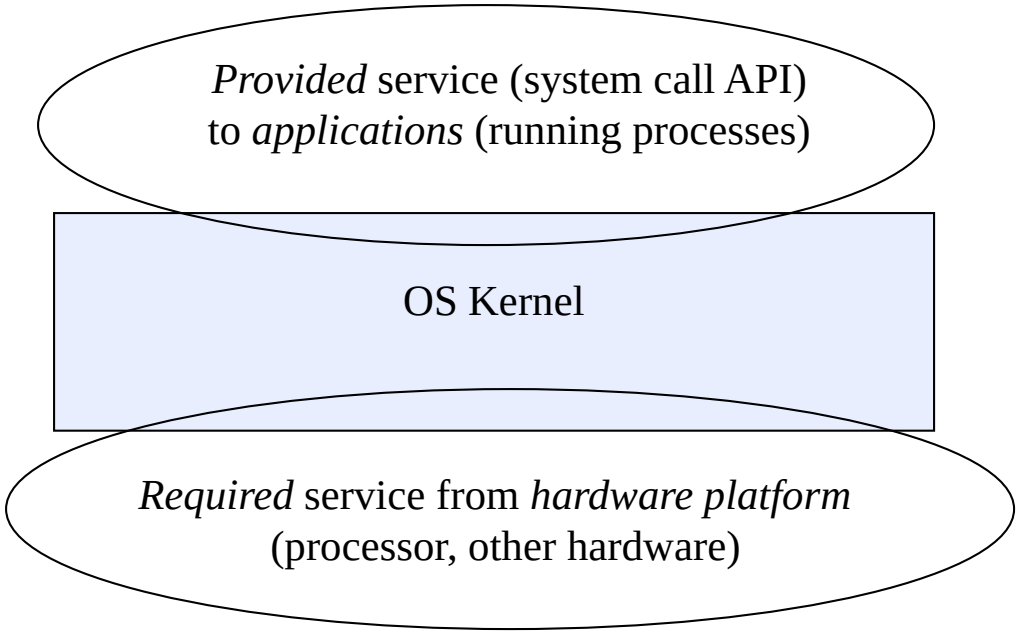
- **Note:** tasks are not threads or processes. Tasks are mapped onto those eventually.

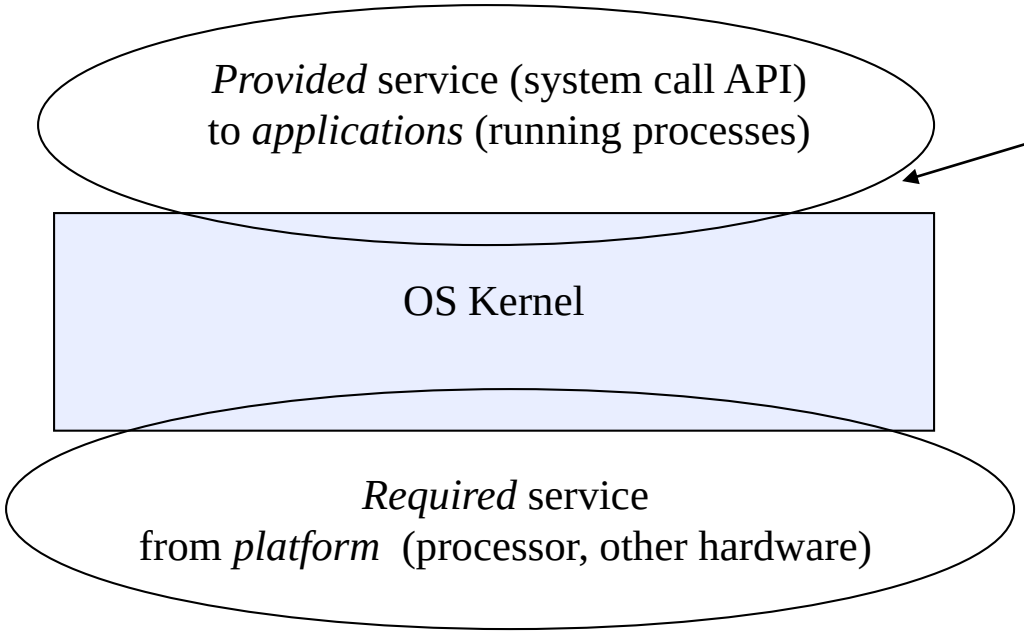
Processes and threads

Questions

- How is this task model mapped onto a platform(= OS and hardware)
  - what are effects of variations in this hardware?
  - what interface, what concepts does the OS provide?
  - how does this all influence timing?

- Do our codes really have this structure?
  - what are tasks, activations and deadlines then?
  - which timing properties are important for our code?
  - most often:
    - code is a single application that works as fast as possible
    - or code is a collection of communicating applications, possibly distributed over several machines
    - in addition, there are few guarantees from the platform
    - so, we test and run it until we are satisfied


Provided service (system call API)
to *applications* (running processes)
OS Kernel
Required service from *hardware platform*
(processor, other hardware)

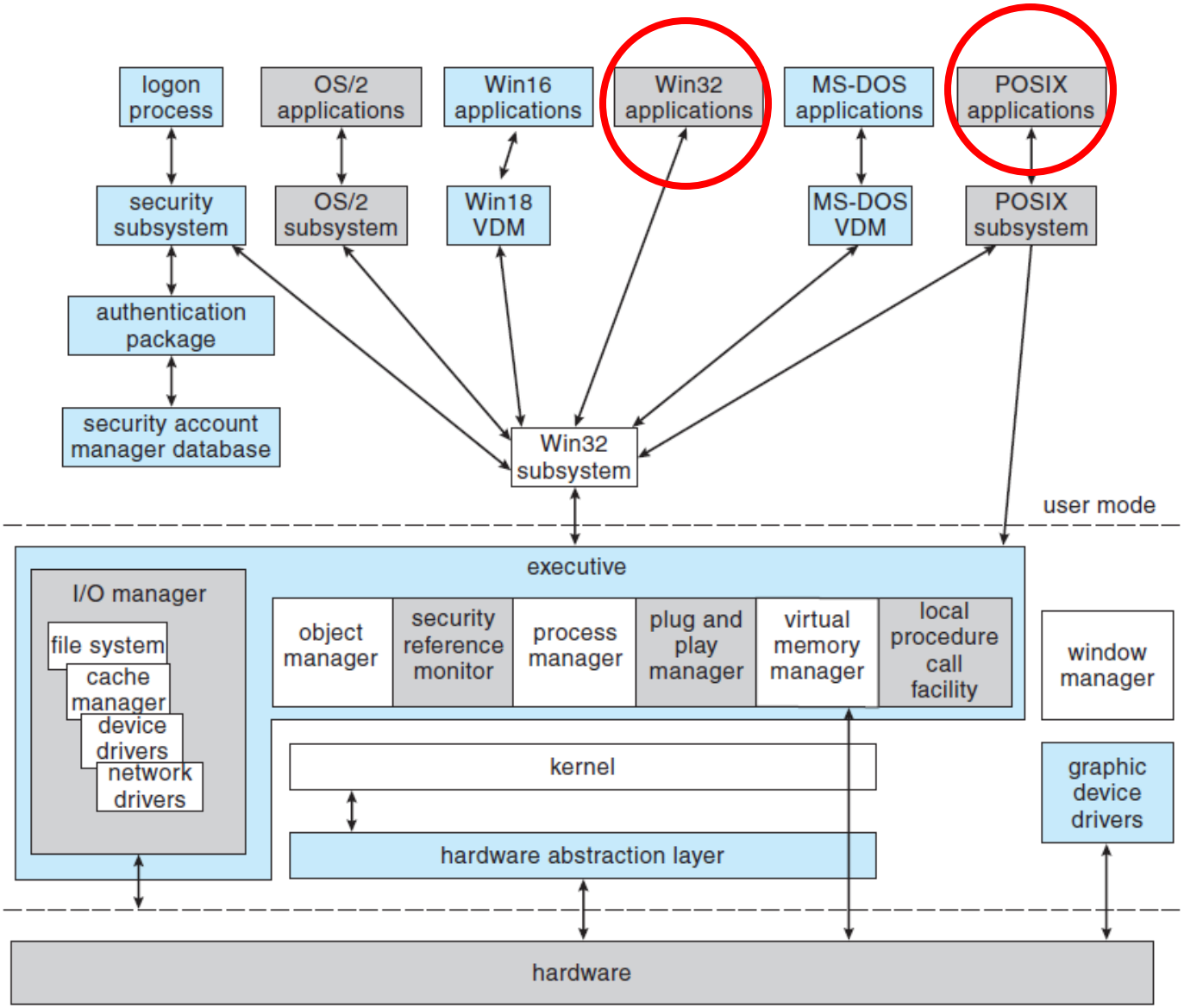


Process management
- *fork(), exec(), exit(), wait()*

Thread management
- *pthread_create(), pthread_join(), ...*

Synchronization
- *semaphores, condition variables,...*

Communication
- *pipes, sockets, ...*



Windows 7 block diagram, from Operating Systems Concepts by Silberschatz, Galvin, Gagne

The OS API: common API chapters

- Process & thread management
- Memory management
- File management
- General I/O
- Miscellaneous (e.g timing and resource use)

- For an RTOS (Real-Time OS)
  - add system calls to deal with time and scheduling
  - add timing (e.g. timeout) to existing API

API standardization: POSIX

- Portable Operating System Interface
  - UNIX-like

- Goal: source-code portability of applications
  - in practice: just reduce portability effort

- Standard, IEEE, ANSI, ISO, developed in API chapters:
  - mandatory ('base') & optional parts, per chapter
  - the entire set of chapters covers 'everything'

- Versioning: Posix 1003.1x-year, where x is the chapter
  - x=a, or no x: basic set of systems calls, like UNIX; 1003.1a-2004
  - x=b: real-time extensions (also: POSIX.4); 1003.1b-1993
  - x=c: multi-threading support (also: POSIX.4a); 1003.1c-1995
  - x=d: additional real-time extensions; 1003.1d-1999
  - x=j: advanced real-time extensions; 1003.1j-2000

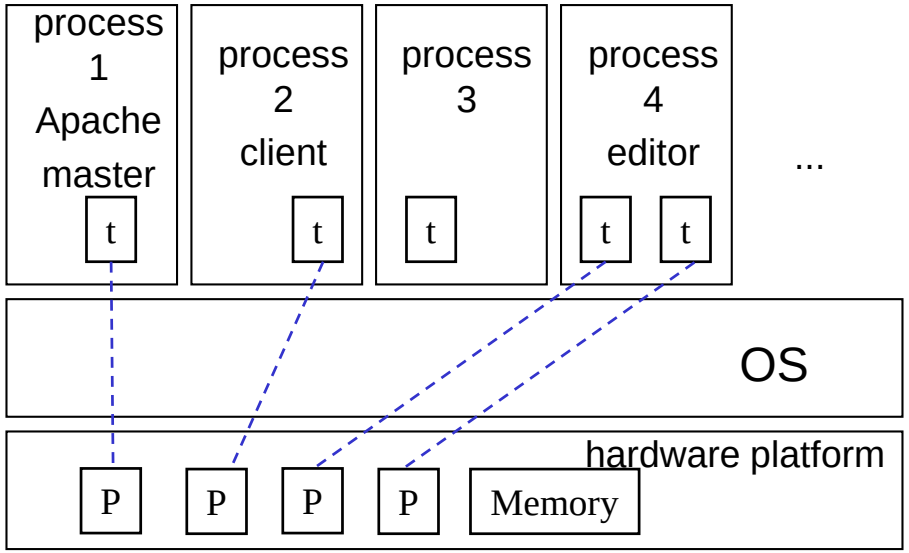
Concepts: Process, Thread

- Process ("program in execution")
  - defines a data space (virtualization of the memory)
  - defines ownership of resources
  - has at least one associated thread
  - unit of deployment (distribution)
  - unit of fault containment

- Thread
  - unit of concurrency (virtualization of the processor)
  - unit of scheduling
    - though in case of one thread per process, the *process* is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated *execution state*
    - place where it is in the code, stack image and return addresses of function calls, values of processor registers, priority, …

Motivation for using processes

- Introduce safety boundaries
  - separate users
  - separate distinct, independent tasks
- Follow solution structure
  - e.g. (older) organization of the Apache web server
    - a managing process
    - client process per connection
- Increase concurrency level
- Allow for distribution

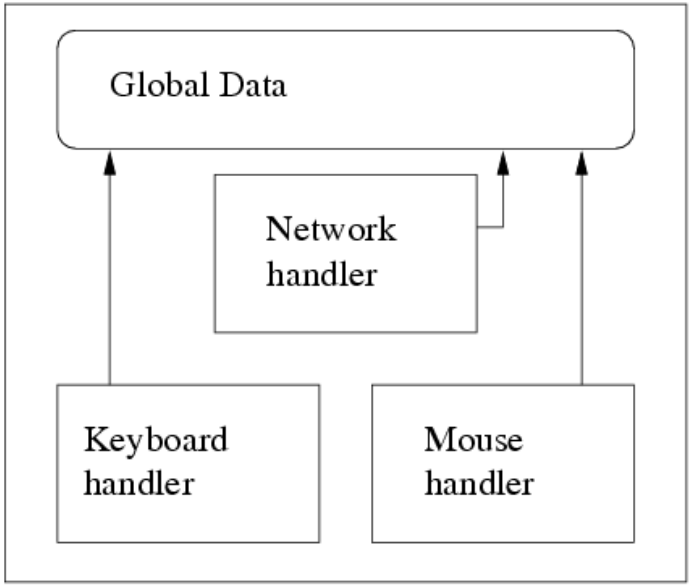


Reasons to introduce threads

- Increase concurrency level
  - performance
    - hide latency
    - increase responsiveness
    - exploit platform concurrency
  - discriminate importance levels in activities
    - e.g. interrupt routines

- Deal with the natural concurrency
  - natural organization, structure ... e.g.
    - thread per event
    - thread per resource
    - thread per (active) external interaction sequence
      - e.g. in user interfaces

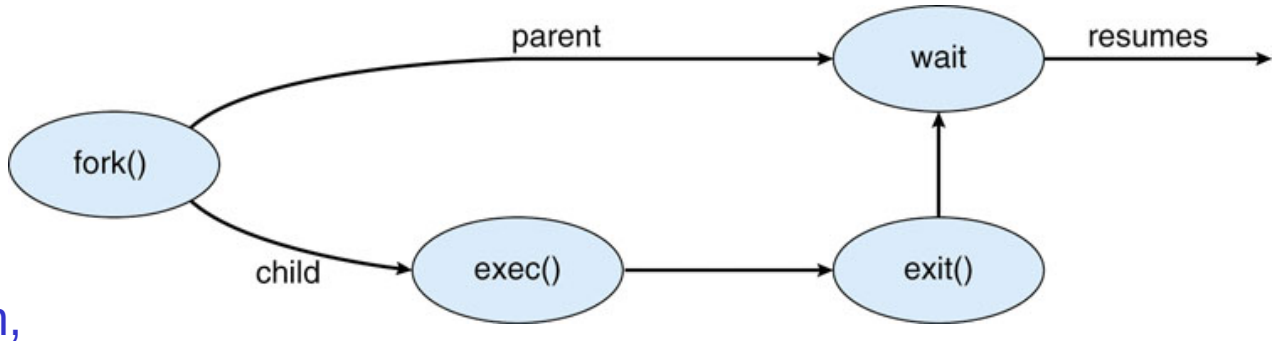
single process



POSIX processes (1003.1)

- Starting point is an already running process…
  - shell (command interpreter) or other program in the GUI (windows, Xwindows)
  - … that was started through a login procedure

- POSIX 1003.1 API: *fork() – exec() – exit() – wait()*

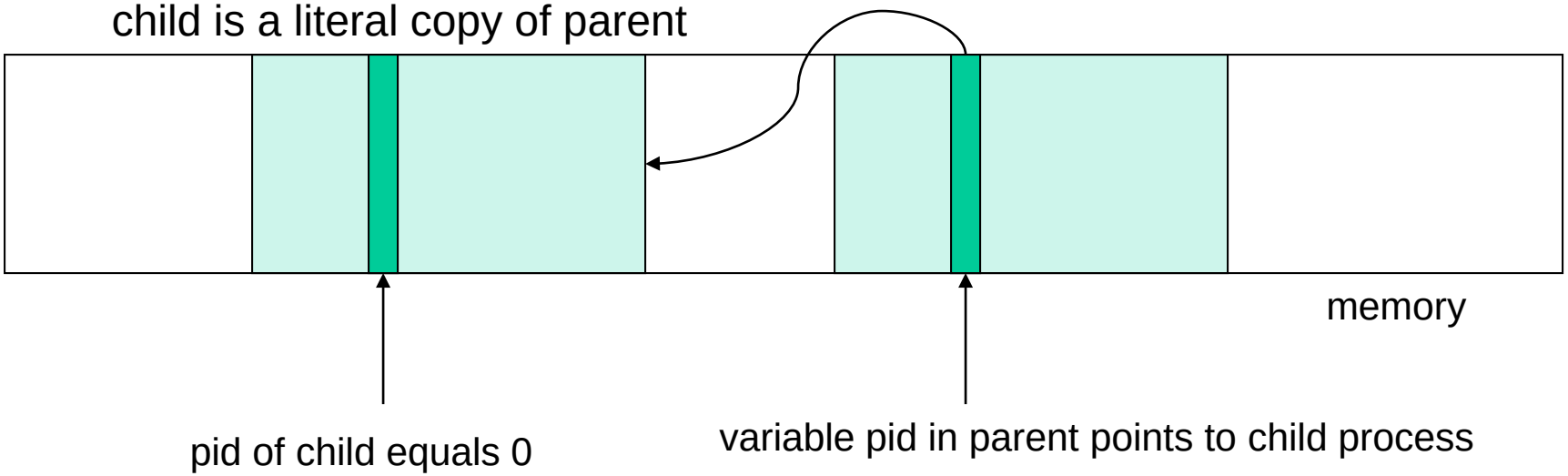


- POSIX process:
  - an executing program
  - unit of distribution, concurrency, *scheduling*
  - single address space: memory management
  - includes one 'natural' execution thread

Before and after "*pid = fork()*"

Parent process about to perform *pid = fork()*

Parent and child after *pid = fork()*
child is a literal copy of parent



Create new process: code fragment

- *fork()* creates an identical copy of the caller in a separate memory space; only the return value stored in variable '*pid*' differs between the two: 0 for the child, child- process-id for the parent
- *execlp()* overwrites the calling process with its argument (file containing an executable program)
- This means that code following *execlp* is never reached
- *perror()* is a generic error printing routine

```
pid_t pid;

......
pid= fork();
if (pid<0) /* error occurred */  { perror ("fork"); exit (-1); }

if (pid== 0) /* the child */ {
      execlp ("child program", "own name", arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror ("execlp"); exit (-1);
}
else /* the parent; pid == process id of child */ {
      /* do whatever you want, e.g., just return from this routine */
}
......
```

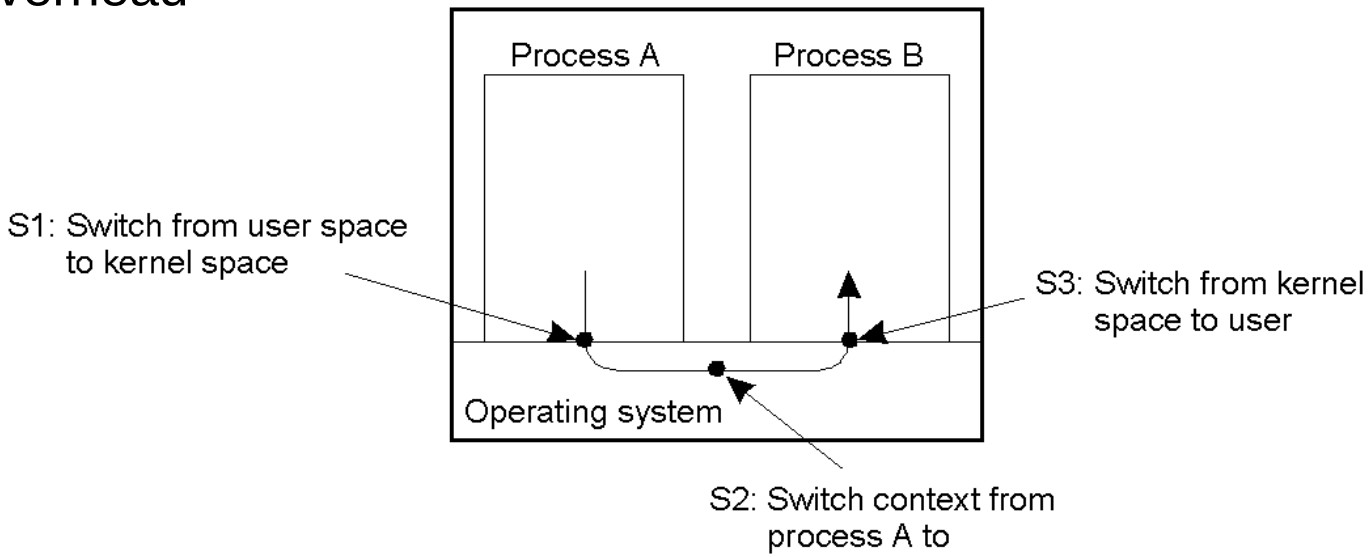
Termination of children: fragment

- Use *exit(status)* to terminate
- Need to wait for children to free child's resources
  - on many systems at least
- Functions *wait(), waitpid()*
  - *wait()* blocks until a child returns; *waitpid()* until a particular child returns.
  - blocking is avoided here (WNOHANG)
- Alternative: (not shown here): use asynchronous notification: signals

```
child:
      ..... exit (23);

parent:
      pid_t pid, terminated; int status;

     /* blocking wait */
     while (pid != wait (&status)) /* nothing */;

    /* or polling wait */
     terminated = (pid_t) 0;
     while (terminated != pid) {
         terminated = waitpid (pid, &status, WNOHANG);
         /* other useful activities */
     }
     /* both cases: status == 23 */
```

Process shortcomings

- Creation overhead



- Switching overhead: *context switch*
  - switch for each Inter-Process Communication (IPC)
  - state save/restore
    - CPU state
    - memory
      - TLB, MMU, cache

Threads

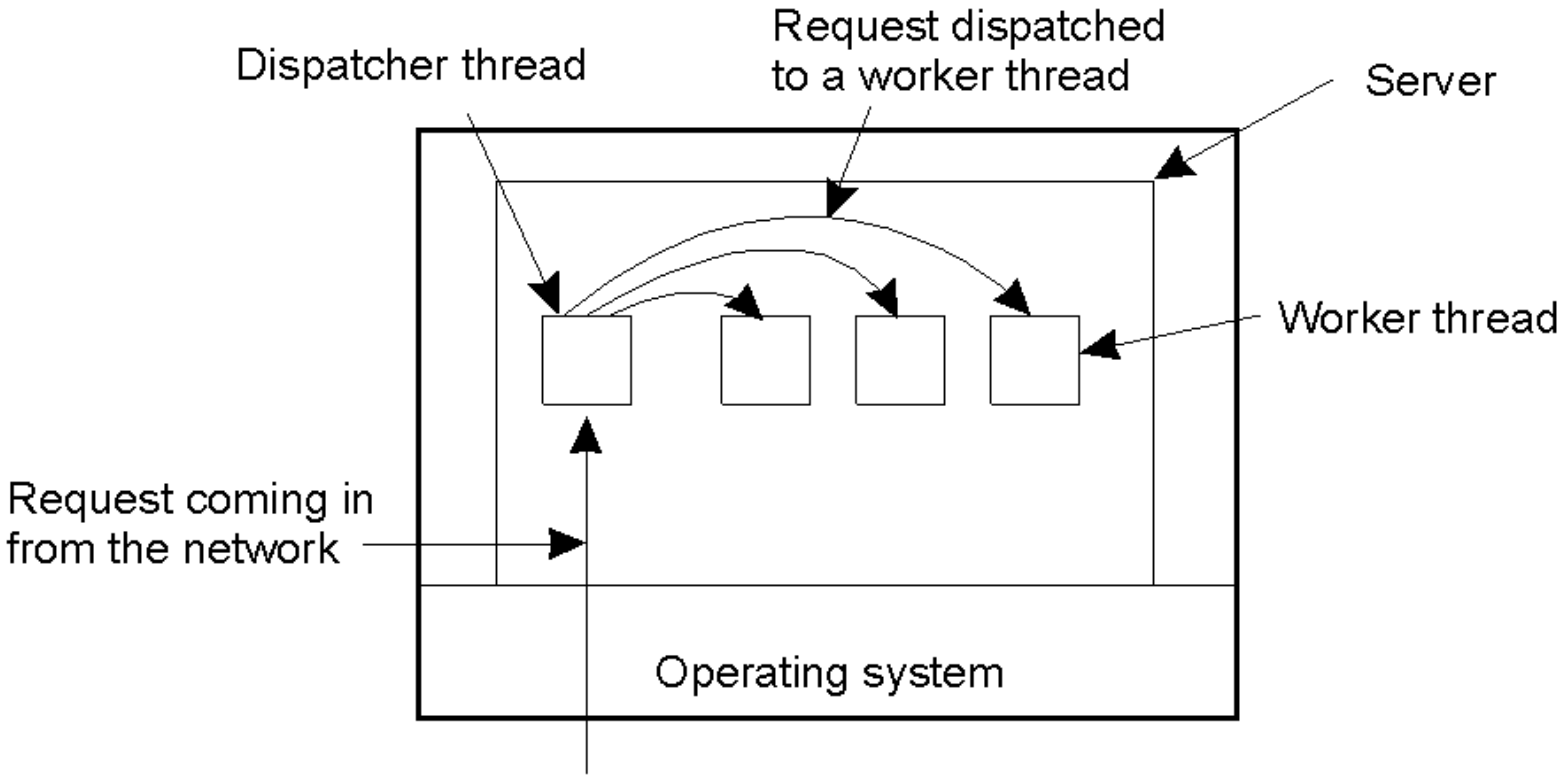
- Concurrency on shared memory
  - no protection agains other threads (faults, memory sharing)
  - no expensive IPCs between threads
  - easier to construct programs (?)
    - *exploit platform concurrency*: multiple processors will run multiple threads
    - *latency hiding*: while waiting do something useful in another thread

- State per thread
  - control state ('context'): CPU registers, masks
  - stack

- Generally, little overhead for switching between threads of same process
  - depending on underlying thread execution model

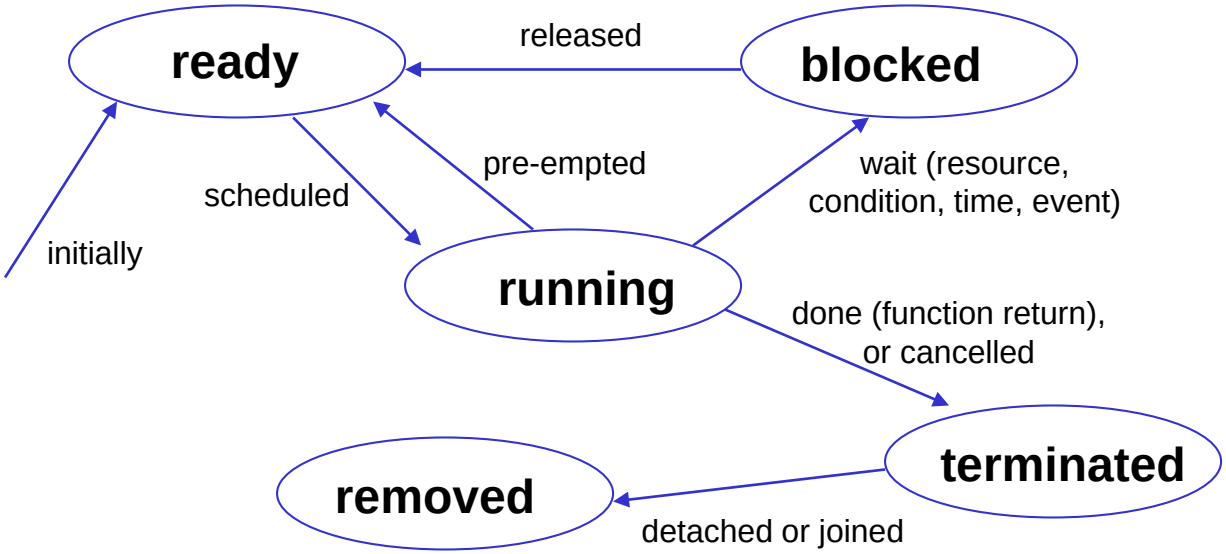
Example: multithreaded server

- Dispatch/worker model



The life of a thread

- Main program (process): own thread
- Additional threads:
  - created on demand or on receipt of signal
  - thread code: a function in the program
  - behavior & limitations: determined by startup attributes



POSIX 1003.1c: Thread interface

- Start a *function* as a new thread
  - setting attributes, e.g. stacksize;
  - thread terminates when this function returns

- Extensive interface
  - e.g. detach, join, cancel

```
pthread_t thread_id;

status = pthread_create (&thread_id, attr, func, arg_of_func);
/* func(arg_of_func) is started as a new thread; when attr == NULL some
 * internal defaults are chose */

status = pthread_join (thread_id, &result);
/* wait for thread to terminate */
```

Example program

```
#include <stdio.h>
#include <pthread.h>

int x;

void *Count_100 ()
{
  int i;
  for (i = 0; i < 100; i ++) {
    x = x+1;
  }
  return (NULL);
}

int main ()
{
  pthread_t thread_id;

  x = 0;
  pthread_create (&thread_id, NULL, Count_100, NULL);
  Count_100 ();
  pthread_join (thread_id, NULL);
  printf ("x = %d\n", x);
}
```

- How many threads are here?
- How many processes?
- What are *possible* final values of *x?*

Resource Monitor

File Monitor Help

Overview | CPU | Memory | Disk | Network

Processes — 47% CPU Usage — 56% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average C...
svchost.exe (netsvcs)	492	Host Proc...	Running	61	24	24.39
chrome.exe	21396	Google Ch...	Running	10	14	4.25
chrome.exe	21340	Google Ch...	Running	10	1	1.79
chrome.exe	7068	Google Ch...	Running	10	1	1.78
perfmon.exe	4556	Resource ...	Running	18	1	3.60
taskmgr.exe	29556	Windows ...	Running	5	1	1.54
System	4	NT Kernel ...	Running	154	0	5.12
dwm.exe	2388	Desktop ...	Running	5	0	0.50

Services — 25% CPU Usage

Name	PID	Description	Status	Group	CPU	Average C...
NlaSvc	1348	Network L...	Running	NetworkS...	25	24.31
VMAuthdService	2688	VMware A...	Running		0	2.21
WSearch	4904	Windows ...	Running		0	0.14
SSDPSRV	3380	SSDP Disc...	Running	LocalServi...	0	0.08
BFE	1640	Base Filteri...	Running	LocalServi...	0	0.07
SysMain	19080	Superfetch	Running	LocalSyste...	0	0.06
MpsSvc	1640	Windows F...	Running	LocalServi...	0	0.06
McAfeeFramework	2032	McAfee Fr...	Running		0	0.05

Associated Handles — Search Handles

Associated Modules

CPU 0 100% 0%

CPU 1 - Parked 100% 0%

CPU 2 100% 0%

CPU 3 - Parked 100% 0%

Physical Memory (MB)	
Total	8107
Cached	3612
Available	5368
Free	1884

Kernel Memory (MB)	
Paged	365
Nonpaged	90

System	
Handles	29434
Threads	1263
Processes	95
Up Time	0:01:14:14
Commit (GB)	2 / 15

Resource Monitor...

Concerns with processes and threads

- *Introduction of concurrency*: multiple activities sharing resources
  - simulated (1 processor) or real, with same effect:
    - interleaving of atomic actions (machine instructions)
  - non-deterministic behavior that is difficult to foresee or reproduce
    - debugging is difficult
- *Concurrency control*
  - manage access to shared resources like devices, memory locations, the OS(!)
    - synchronization, communication, scheduling
    - competition on resources
- *Overhead and delay*
  - switching
    - excessive use of threads in an application may cost performance
  - administration
  - waiting (on others)

Exercise

- Examine the project you are working on
  - how do you use (multiple) processes, threads?
  - for what purpose is this?
  - in which way do you control their execution and scheduling?
  - which synchronization and communication aspects do you have?

Exercises

For this exercise, assume that pre-emption of tasks is not possible. Draw Gantt charts to express timing.

- **C.1** Given is a system with 3 tasks, with (period, computation time) pairs of (4,1), (8,1), (16,1).
  - How often should a timer fire (which period should it have) for the case of the tasks having the same phasing.
  - Determine response times – worst case for this phasing.
  - Try to find an optimal phasing.

- **C.2** Consider a system with two tasks, one with a period of 2 and a computation time of 1 and the other with a period of 20 and a computation time of 5.
  - Give timer interrupt frequency for the case of the tasks having the same phasing.
  - Is there a problem? Solve it by proposing subtasks (i.e., smaller tasks)
  - Determine response times – worst case for this phasing.
  - Try to find an optimal phasing.

- **C.3** Discuss C.2 again but now with task 1 having a period of 6. Is there still a need for subtasks?

Discussion: mapping of tasks

- A task is the response to an event (timer, other)
  - a task can be associated with a thread, a process or just a call
  - however, this is a *mapping* issue, a matter of design choice

- Option 1: each task mapped onto a process
  - and, by result, onto a thread
  - then tasks can share resources only through the OS
  - or they approach them using message passing
    - the resource is maintained by a separate process
    - and access is through communication

- Option 2: each task mapped onto a thread
  - newly created or repeatedly restarted
  - resources are accessed within the shared space
    - proper exclusion mechanisms are needed

Event driven:

```
while true do
  SemWait (Start);
  task();
od
```

Time driven:

```
while true do
  task();
  Suspend until next period
od
```

Multiplexing

- Several threads (or processes) per task
  - then the actions of a task are distributed across several threads
  - a logical consequence of having *active objects*
    - active objects: data + operations + independent, private behavior
      - “object with private thread”
    - simplest form: message passing process
    - happens in pipelined processing

- Several tasks per thread
  - consequence of the above
  - efficiency concern in the mapping, e.g. when the tasks exclude each other
  - in the extreme case: just one thread executing all tasks as calls

- However, mind the effect of blocking (both cases) – time interference

Consequences for scheduling

- Assume that each *task* has a fixed priority
  - Option 1 and 2: direct mapping task on priority of thread/process
    - OS must support fixed priorities for threads and processes
  - one thread executes several tasks completely
    - OS must support dynamic priority assignment to threads
    - priority setting derived from the task the thread is executing
  - one task is executed by several threads
    - the threads act as resources to be acquired by the task
    - dynamic priority assignment required
      - the priority of a thread is again determined by the task it is working for