What is a Task

- A task is a type of job or set of activities to be done repeatedly (in periodic, aperiodic or sporadic intervals) when triggered.
- The implementation of a job is described by a task specific program operating on job specific data.
- Several tasks can by help of a control program run concurrently on a processor.
- A more advanced control program (operating system) enables tasks to be individually started, stopped and resumed.

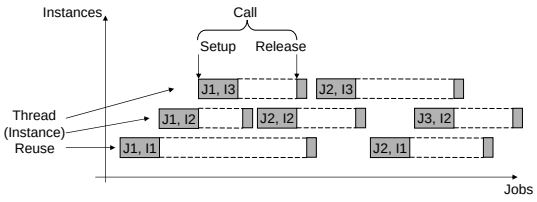
What is a Task, cont.

- A task executes in a unique execution context; incl. program-, stack- and data-pointers defining its control and data state.
- The data needed to handle the execution context of a task instance is maintained in a record often called task control block (TCB).
- Each instance of a task type needs memory space for its individual data and stack content.

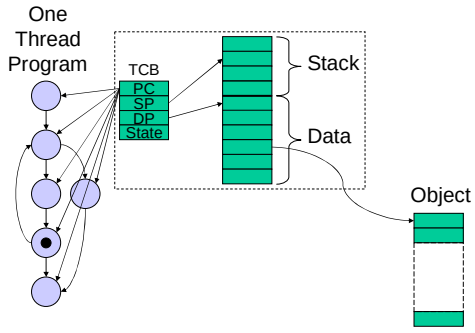
Task threads, instances and jobs

Same type of work (e.g., phone call setup and release) done:

- concurrently (several instances in space)
- repeatedly (several instances after each other in time)
- reusing tread data structures



Task implementation



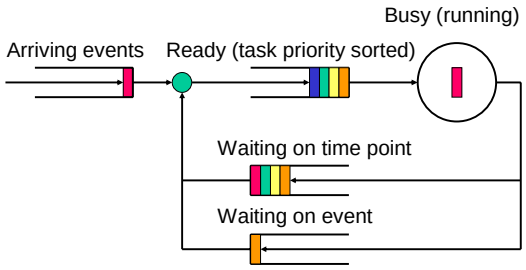
Control Program / Operating System

- To support the handling of programs and associated execution state changes a control program provides several services.
- Larger control programs are often called operating systems (OS).
- The core of an OS is often called the kernel.
- Scheduling of programs with state (tasks) is a basic service that an OS kernel provide.

Operating system (OS) services

- A task can be controlled by and react to OS services such as: start(task), stop(task), wait_on(event), resume(task), etc.
- A task can issue a wait_on(X), to wait for an event X to occur (e.g., an external condition or a clock time event).
- The task can then resume its execution (performing some more actions) and then typically return to the “wait on event” state.

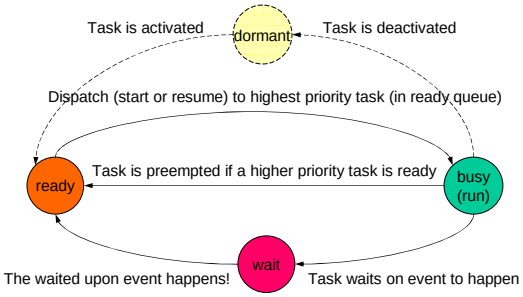
Task queue handling



Task type and task instance

- The program part of a task is shared among all task instances running on the same processor.
- The data part of a task defines the state (incl. PC, SP, DP, stack, heap and run_state) of an instance of the task.
- A task can thus exist in several instances with their respective control state and data.

Task instance control states



Job

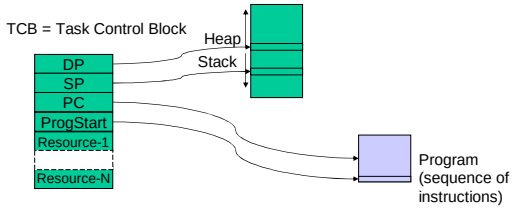
- A job is released) periodically (controlled by a clock driven schema), sporadic or aperiodic when messages arrives or when (external or internal) events occurs.
- A job is finished when all activities specified by the program implementing the task has been executed.
- A larger job or external service is often carried out by a set of cooperating smaller jobs needed to perform the whole job.

Job, cont.

- To do a job, as any other work, energy is required. Some energy is needed each time a task instance is executed by a computer.
- The term job, as used in computing, origins from the time of “batch” processing.
- A batch of punched cards described a job.
- A computer typically executed jobs one at a time from a FIFO ordered job queue.

Task control block

- In many applications there is only need for one data instance and simultaneous invocation (thread or copy) of each task



Task (instance) control block

A typical TCB may contain information about:

- Program Pointer (program start address)
- Program Counter (PC) (resume address)
- Stack Pointer (SP)
- Max size of stack
- Max size of heap
- Task state
- Priority
- Resource state
- ….

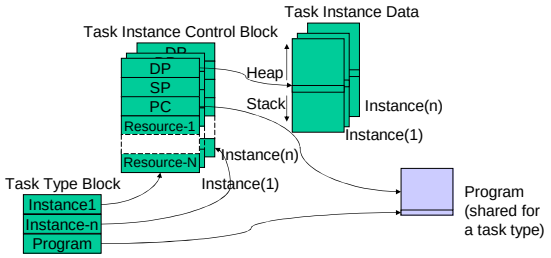
Task instance states

A task instance can be in a number of states:

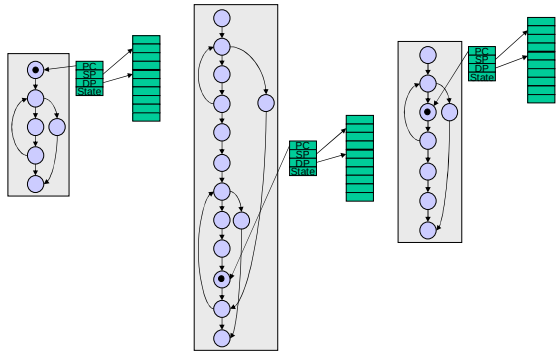
- **dormant** – the task instance is not yet activated or put aside to offload the scheduler
- **ready** – a new job or waited for event or message has arrived and a task instance is activated waiting for its turn to execute
- **busy** – the task instance is currently executing on the processor to perform a job
- **wait** – the task instance waits on an event, message, time or a resource (to become free)

Task instance control

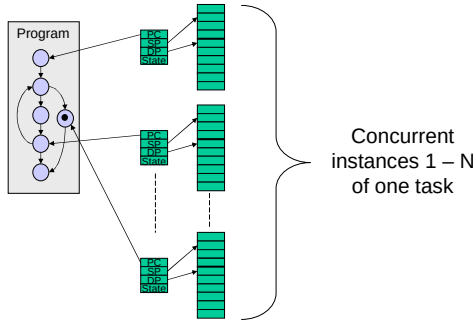
- In other applications some types of tasks must run as several concurrent instances



Different tasks



Several instances of a task type



Resources

Types of resources in a typical distributed real time system:

- Sensors
- Actuators
- Processors
- Memories
- Communication channels (time slots, frequencies, codes, …)
- Data (environment conditions, state of resources, parameters, etc.)

Some resources are:

- Private
- Shared
- Pooled (many replaceable)
- Unique (“one of a kind” not replaceable)
- Globally available
- Locally available

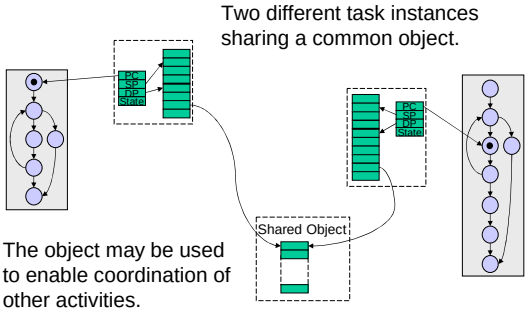
Some tasks require reservation of more than one resource:

- Sequentially (one after another but not at the same time or overlapping)
- Concurrently (more than one in parallel or overlapping in time)

Resource utilisation

- The OS provide basic services for efficient utilisation of the processor, memory, I/O and other resources.
- Flags, locks, priority inheritance/ceiling and other similar mechanisms helps to coordinate use of scarce shared resources.
- Efficient utilization of resources are up to system designers to care for by properly designed programs (tasks).

Shared objects (resources)

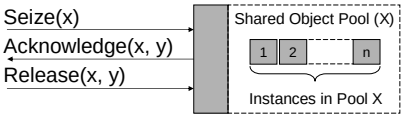


Mutual exclusion

Resolve resource sharing conflicts by use of:

- **Test-and-set** instruction operating on flag
- **Critical section** (preemption protected code)
- Resource reservation handling **protocols** or **procedures** and associated reservation data structures such as **semaphores**
- Messages/events (**seize**, **acknowledge**, **release**) for communication with resource handling task

Resource reservation, example



1. Request resource of type x using **seize**(x)
2. Get **acknowledge**(x, y) in return
3. Use the resource instance y of type x
4. Release resource instance y of type x using **release**(x, y)

Optimize on system level

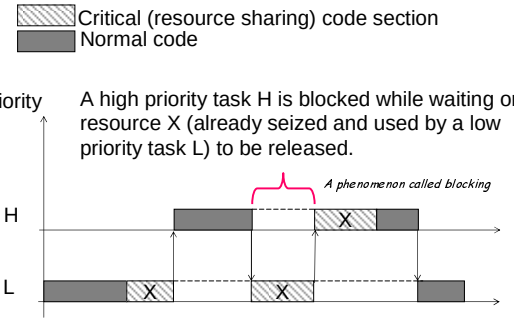
Potential gains:

- efficient use of controlled resources or
- efficient control of real physical processes (often very valuable)

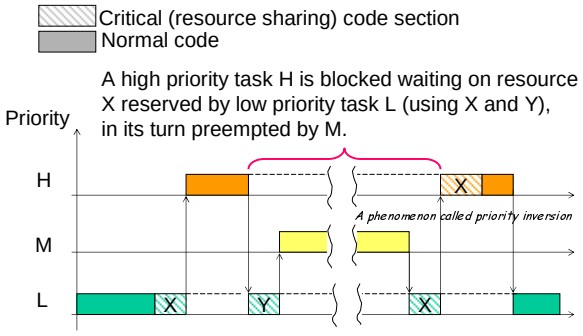
Such improvements can often motivate:

- a powerful computer system and
- the design of complex programs

High priority task blocked waiting



Priority inversion



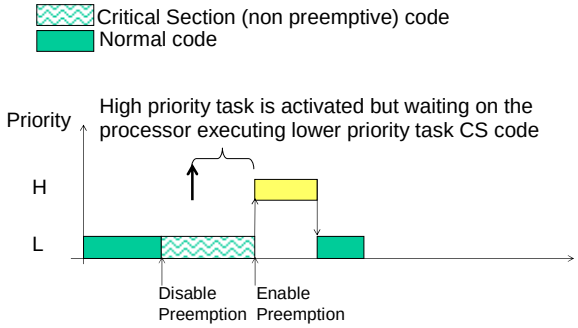
Critical section

- Sensitive code that for example access a shared resource (data structure or device)
- Used for handling of resources that only can be used in a safe way by one task at a time
- The term critical section (CS) has two interpretations:
  - Sensitive resource sharing code
  - Code that is not allowed to be preempted

Non preemptive critical section

- The risk for priority inversion and dead lock can be avoided if CS code can't be preempted
- However, such CS should be kept short, not to block higher priority tasks longer than necessary

Non preemptive critical section



Means to avoid priority inversion

- Non-preemptive Critical Sections
- Priority Inheritance
- Priority Ceiling

Problems:

- the first method can block higher priority jobs;
- the second can cause problems with:
  - chained blocking
  - deadlock

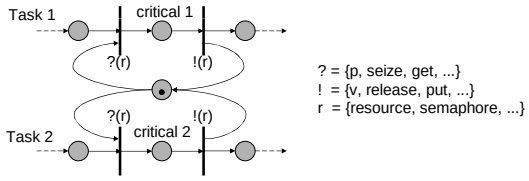
Thus the third is most free from problems.

Priority ceiling

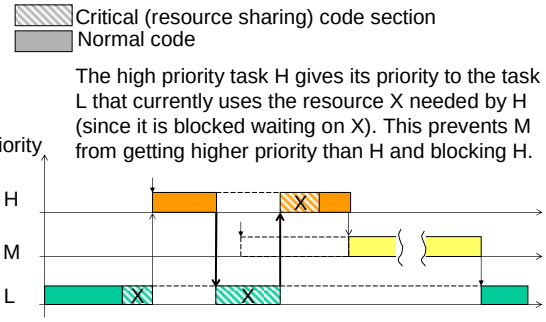
- A job that reserves a resource (marked by a flag or semaphore) will get its priority dynamically raised to the level of the highest priority job that may need to use the same resource but only when the higher priority job attempts to reserve (lock) this resource.
- Philosophy – the lower priority job is empowered to finish and release a scarce resource as soon as possible, when the higher priority job needs it.
- This solution is made to avoid deadlock and to improve resource utilization.
- No job can take a reserved resource from another job until it is released.

Resources and priority handling

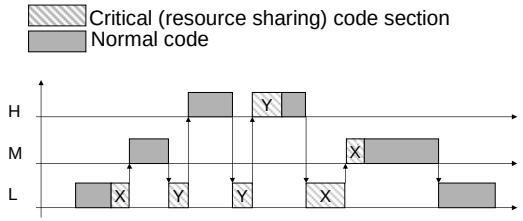
- In the priority ceiling protocol the priority of a resource is lifted to the level of the highest priority process that may need to use it.
- The resource use and priority changing information is communicated via the resource reservation mechanism (flag, semaphore, monitor or resource handler).



Priority ceiling



Priority ceiling, cont.



While task L uses X the task M, that also needs X, tries to reserve X the task L gets its priority raised to the level of M and latter when L then uses Y and H tries to lock Y, L gets its priority raised to the level of H.

Sensors and Actuators

Sensors

- Mechanical (e.g., gauge)
- Electro mechanical (e.g., microphone)
- Optical or light
- Chemical
- Biological

Actuators

- Displays
- Loudspeakers
- Linear and rotating motors (engines)
- Electrically controlled hydraulic and pneumatic valves

Transducers

- Translates from one form of signal to another form of signal (often via electrical).
- Analogue electrical signal to digital via A/D converter and vice versa via D/A converter.
- Can include a combination of sensors and actuators.

Control system – Controlled system