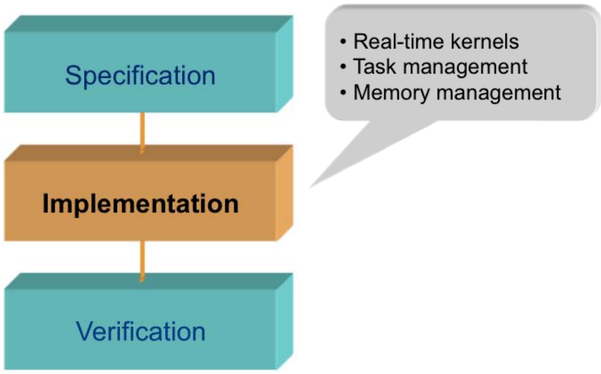
CHALMERS

Real-Time Systems



CHALMERS

Real-time kernels

Most real-time kernels contain the following minimal set of functions:

- Task management
  - create, terminate and switch task
- Synchronization
  - semaphores, mutual exclusion
- Interrupt handling
  - I/O, real-time clock
- Memory management
  - memory mapping, memory protection

CHALMERS

Task management

In the general case, the number of tasks is larger than the number of processors available. This raises the following questions:

1. **How** should the processor be shared?
   - Serial execution (cyclic executive)
   - Pseudo-parallel execution
2. **When** should task switches take place?
   - At natural stops (e.g., at *wait* or *delay* operations)
   - At changed system state (e.g., after *signal* operations)
   - At clock or I/O interrupts
3. **Which** task should execute?
   - Scheduling policy

CHALMERS

Task management

Serial execution: (cyclic executive)

- The system contains a table describing a predetermined (cyclic) execution order for the tasks.
- A task executes until it terminates; then, the next task in the table is started.
- Properties:
  - Works best for independent tasks that can execute in an arbitrary order
  - There is no need for semaphores or other synchronization to guarantee mutual exclusion
  - Requires short task code segments in order to provide short response times for external events

CHALMERS

Task management

Pseudo-parallel execution:

- Multiple executable tasks compete over the processor.
- The execution of a task can be interrupted before it is completed in favor of another task
  - Based on task priorities (real-time kernels)
  - Based on time quanta (time-shared multi-user systems)
- Properties:
  - Works well for dependent as well as independent tasks
  - Semaphores or other synchronization may be needed to to guarantee mutual exclusion
  - Response times for external events become very short

CHALMERS

Task management

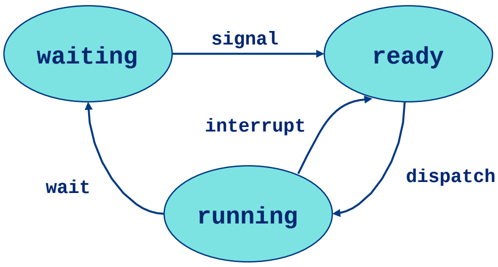
Process context:

- The process context consists of the status information that is stored in the processor, for example:
  - General registers
  - Program counter (PC)
  - Stack pointer (SP)
- In the event of a task switch, the context must be stored so that the current task can continue its execution when it once again gains access to the processor.
- Consequently, a task switch will in practice also involve a context switch.

CHALMERS

Task management

Task states:



Running: Currently executing task
Ready: Task that is available for execution
Waiting: Task that cannot execute because it is needs access to a resource other than the processor

CHALMERS

Task management

Process control block: (PCB)

- A data structure in the real-time kernel that contains information about a task in the system.
- PCB typically contains:
  - Pointer to next PCB (linked list)
  - Task state
  - Task identifier
  - Task priority and/or time quanta
  - Pointer to the task's stack area
  - Pointer to the task's program code

CHALMERS

Task management

Process queue:

- A data structure in the real-time kernel that is used for defining groups of PCBs.
- The following queues should exist in a real-time kernel:
  - One queue for currently-executing task (Running)
  - One queue for the remaining executable tasks (Ready). This queue is sorted according to the chosen scheduling policy.
  - One queue for tasks whose execution should be delayed until a given time instant (Delay).
  - One queue for tasks waiting for an interrupt (Interrupt).
  - One queue per semaphore for tasks waiting for access to that semaphore

CHALMERS

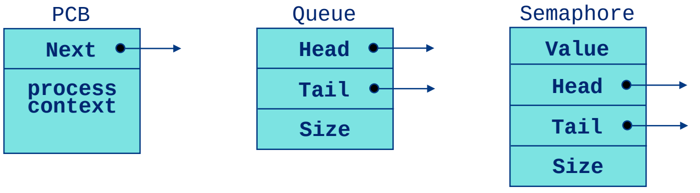
Task management

Process queue:

- The following operations manipulate the queues:

Insert	add an element to the queue
Remove	delete an element from the queue
Dispatch	perform a task switch

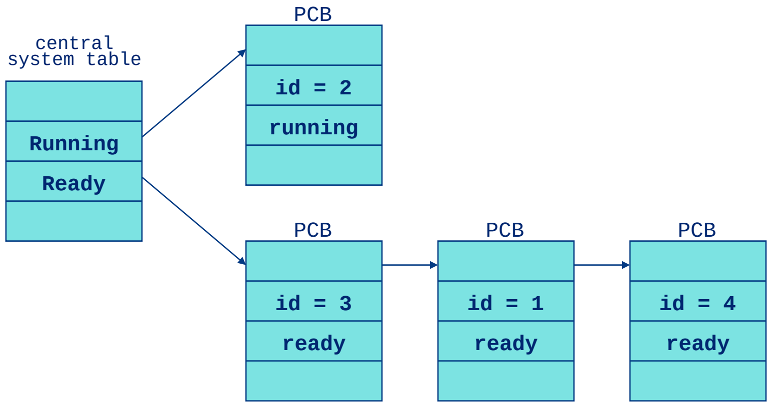
Some examples of queue-based data structures:



CHALMERS

Task management

Examples of process queues:



CHALMERS

Task management

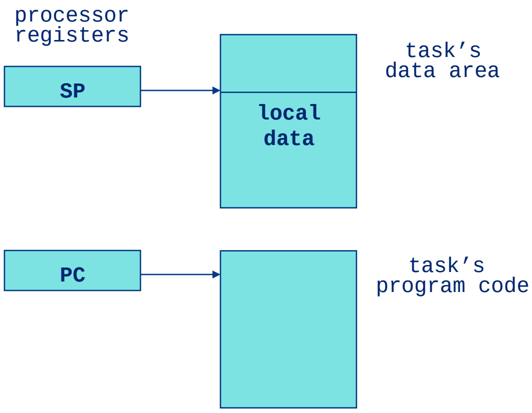
Task switches are typically executed by a special code in the real-time kernel, called the dispatcher, whose functions are:

- Save context:
  - Store the interrupted task's context in its PCB
- Start a new task:
  - a) Select the task that is first in the Ready queue
  - b) Fetch the PCB for that task
  - c) Update the Running queue with the new PCB
  - d) Load the context for the task from its PCB
  - e) Start the execution of the new task

CHALMERS

Task management

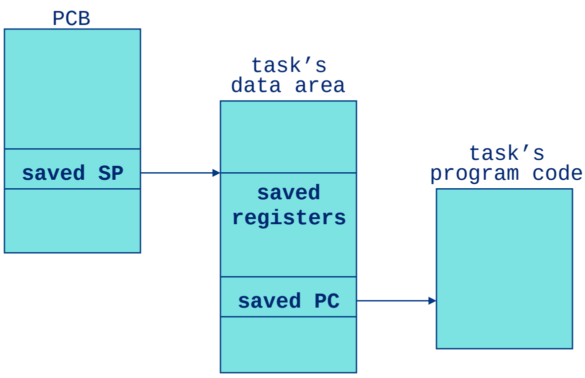
Context for executing task:



CHALMERS

Task management

Context for waiting task:



CHALMERS

Task management

What system events may cause a task switch:

Wait	The executing task is blocked when trying to get access to a semaphore.
Signal	A task with high priority becomes ready for execution because a task with lower priority leaves a critical region.
Delay	A task requests to be delayed.
Clock interrupt	A task with higher priority becomes ready for execution after a delay, or currently executing task has consumed its allotted time quantum.
I/O interrupt	A task with higher priority becomes ready for execution due to an external event.

CHALMERS

Task management

What happens at a call to **Wait**?

1. Interrupts are disabled. (**Wait** is a critical region)
2. The context of the calling task is saved and its PCB is updated.

3a. If semaphore = 0, the calling task's PCB is moved to the wait queue of the semaphore

3b. If semaphore > 0, its value is decreased by one and the calling task's PCB is moved to the Ready queue.

4. Interrupts are enabled.
5. Dispatcher is called to start a new task.

CHALMERS

Task management

What happens at a call to **Signal**?

1. Interrupts are disabled. (**Signal** is a critical region)
2. The context of the calling task is saved and its PCB is updated.

3a. If there are tasks in the wait queue of the semaphore, the first task in that queue is moved to the Ready queue.

3b. If no tasks are waiting for the semaphore, the value of the semaphore is increased by one.

4. The calling task's PCB is moved to the Ready queue.
5. Interrupts are enabled.
6. Dispatcher is called to start a new task.

CHALMERS

Task management

What happens at an I/O interrupt?

1. The processor's interrupt mechanism automatically stores selected parts of the interrupted task's context, and its PCB is updated.
2. The I/O unit that requested the interrupt is served.
3. The interrupt service routine checks whether any task in the Interrupt queue has become ready for execution. If so, that task's PCB is moved to the Ready queue.
4. The interrupted task's PCB is moved to the Ready queue.
5. Dispatcher is called to start a new task.

CHALMERS

Task management

What happens at a clock interrupt?

1. The processor's interrupt mechanism automatically stores selected parts of the interrupted task's context, and its PCB is updated.
2. The variables that represent calendar time is updated.
3. The interrupt service routine checks whether any task in the Delay queue has become ready for execution. If so, that task's PCB is moved to the Ready queue.
4. The interrupted task's PCB is moved to the Ready queue.
5. Dispatcher is called to start a new task.

CHALMERS

Task management

What happens at a clock interrupt? (comments)

- All real-time systems have a real-time clock that generates an interrupt at regular intervals, e.g., each 10 ms.
- The real-time clock is used for:
  - Keeping track of how long a task has executed. This function is often used in ”watchdogs” whose purpose is to abort tasks that do not behave as expected.
  - Scheduling periodic tasks.
  - Keep track of the delay time for tasks that has called **delay**.
  - Keep track of calendar time.

CHALMERS

Memory management

In a real-time system it is useful to have large flexibility as regards the utilization of the primary memory.

- The real-time kernel should be able to decide the addresses in which the code and data of user tasks are placed.

It is also useful to have a protective interface (firewall) between the real-time kernel and the user tasks.

- A faulty or malicious user task should not be able to write to and possibly corrupt the data structures in the real-time kernel, e.g., queues and PCBs.

These system properties can be achieved with the aid of memory mapping and memory protection.

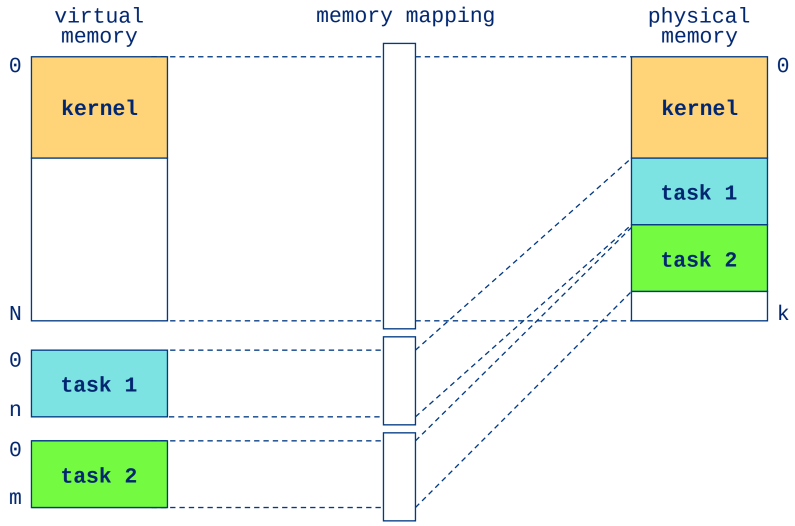
CHALMERS

Memory mapping

- Memory mapping requires special hardware in the form of a memory management unit (MMU).
- The MMU translates the addresses issued by the user tasks (virtual addresses) to real (physical) addresses in primary memory.
- Through memory mapping, a user task can only access the part of the primary memory that it has been assigned by the real-time kernel.
- The real-time kernel itself resides in the physical address space, and is therefore protected from the user tasks.

CHALMERS

Memory mapping



CHALMERS

Memory protection

- The processor has a privileged state (kernel mode) and one non-privileged state (user mode).
- The real-time kernel executes in *kernel mode*, and user tasks in *user mode*. The memory mapping hardware can only be manipulated in kernel mode.
- Before the dispatcher starts a user task, it configures the MMU so that the user task can only access its assigned part of the primary memory.
- Kernel mode can only be entered via hardware interrupts or trap instructions (software interrupts).
  - The services of the real-time kernel is then called via trap instructions (or via subroutine calls for systems without memory protection)