Terms

- A **resource** is any software structure that can be used by a process to advance its execution, e.g. data structure, a set of variables, a main memory area, a file or a set of registers of a peripheral device.

- A resource dedicated to a particular process is said to be **private**, whereas a resource that can be used by more tasks is called a **shared** resource.

- A shared resource protected against concurrent accesses is called an **exclusive resource**.

- A piece of code executed under **mutual exclusion** (to guarantee consistency of data structures in exclusive resources in case of competing tasks) is called a **critical section**. Operating systems typically provide a general synchronization tool called a **semaphore** for building critical sections. A semaphore can be accessed only through kernel primitives called **wait** and **signal**.

- A task waiting for an exclusive resource is said to be **blocked** on that resource. Otherwise, it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the associated resource becomes **free**.

- Waiting state caused by resource constraints:
Terms

- Each exclusive resource $R_i$ must be protected by a different semaphore $S_i$ and each critical section operating on a resource must begin with a `wait($S_i$)` primitive and end with a `signal($S_i$)` primitive.
- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a `wait` on a locked semaphore, it enters a waiting state, until another task executes a `signal` primitive that unlocks the semaphore.
- When a task leaves the waiting state, it goes into the ready state. From all tasks in the ready state, the scheduler selects one for dispatching according to its scheduling policy.

Blocking on an exclusive resource

- Software structure

Priority inversion

- Unavoidable blocking

Priority inversion

- normal execution
- critical section

Unavoidable blocking

J₁ blocked by J₃

Priority inversion can last arbitrarily long

[But97, S.184]
Solutions

- **Disallow preemption** during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked.

- **Construct schedule which is guaranteed to be free of conflicts**
  - (+) simple concept (no semaphores etc.)
  - (-) complex to construct (search procedure)
  - (-) must be changed in case of dynamic tasks
  - (-) no on-line schedule possible

Resource Access Protocols

- **Basic idea:** Modify the priority of those tasks that cause blocking. When a task $J_i$ blocks one or more higher priority tasks, it temporarily assumes a higher priority.

- **Methods:**
  - Priority Inheritance Protocol (PIP), for static priorities
  - Priority Ceiling Protocol (PCP), for static priorities
  - Stack Resource Policy (SRP), for static and dynamic priorities
  - others ...

Priority Inheritance Protocol (PIP)

- **Assumptions:**
  - $n$ periodic tasks which cooperate through $m$ shared resources; fixed priorities, deadlines equal periods, all critical sections on a resource begin with a $wait(S_i)$ and end with a $signal(S_i)$ operation.

- **Basic idea:**
  - When a task $J_i$ blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

- **Terms:**
  - We distinguish a fixed nominal priority $P_i$ and an active priority $p_i$, larger or equal to $P_i$. Jobs $J_1, ..., J_n$ are ordered with respect to nominal priority where $J_i$ has highest priority. Jobs do not suspend themselves. The critical sections used by any task are properly nested; that is, one section is either entirely contained in another one or they are unrelated.
Priority Inheritance Protocol (PIP)

**Algorithm:**
- Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a FCFS discipline.
- When a job $J_i$ tries to enter a critical section and the resource is blocked by a lower priority job, the job $J_i$ is blocked. Otherwise it enters the critical section.
- When a job $J_i$ is blocked, it transmits its active priority to the job $J_k$ that holds the semaphore. $J_i$ resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it inherits the priority of the highest priority of the jobs blocked by it).
- When $J_k$ exits a critical section, it unlocks the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by $J_k$, then $p_k$ is set to $P_k$, otherwise it is set to the highest priority of the jobs blocked by $J_k$.
- Priority inheritance is transitive, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

Priority inheritance protocol (PIP)

**Example:**
- Direct Blocking: higher-priority job tries to acquire a resource held by a lower-priority job
- Push-through Blocking: medium-priority job is blocked by a lower-priority job that has inherited a higher priority form a job it directly blocks

Priority inheritance protocol (PIP)

**Example with nested critical sections:**
- Priority does not change

Priority inheritance protocol (PIP)

**Example of transitive priority inheritance:**
- $J_1$ blocked by $J_2$, $J_2$ blocked by $J_3$. $J_3$ inherits priority from $J_1$ via $J_2$.
Priority inheritance protocol (PIP)

**Schedulability test:** A set of \( n \) periodic tasks using PIP can be scheduled by the rate-monotonic algorithm if

\[
\forall i, \quad 1 \leq i \leq n, \quad \sum_{k=i}^{n} \frac{C_k}{T_k} + \frac{B}{T_i} \leq i \left(2^{1/i} - 1\right)
\]

**Maximum blocking time \( B_i \):**

- Under PIP, a job \( J \) can be blocked for at most the duration of \( \min(n,m) \) critical sections, where \( n \) is the number of lower-priority jobs that could block \( J \) and \( m \) is the number of distinct semaphores that can be used to block \( J \).
- There are algorithms to compute the maximum blocking times \( B_i \).

---

**Problem: Chained Blocking**

- Normal execution
- Critical section

---

**Problem: Deadlock**

- Normal execution
- Critical section

---

[But97, S. 200]