DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Models of Parallel Processing

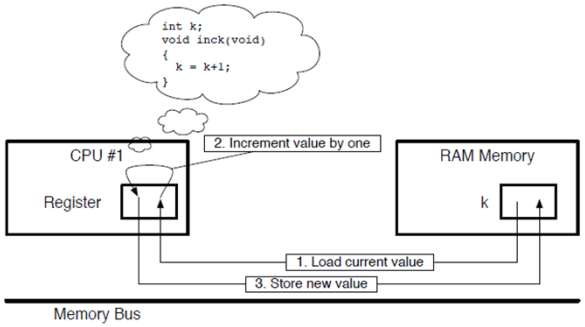
- Race conditions
- Mutual exclusion
- HW mutual exclusion implementation
- SW mutual exclusion implementation
- Semaphores
- Monitors and Condition Synchronization
- Message passing communication

Task Interaction

- In both general-purpose and real-time systems, processes do not live by themselves and are not independent of each other.
- Rather, several processes are brought together to form the application software, and they must therefore cooperate to solve the problem at hand.
- Processes must therefore be able to communicate, that is, exchange information in a meaningful way.
- We have already seen that it is possible to share some memory among processes in a controlled way by making part of their address space refer to the same physical memory region, if separate processes, or share global variables if threads within the same process
- However, this is only part of the story. In order to implement a correct and meaningful data exchange, processes must also synchronize their actions in some ways.
- For instance, they must not try to use a certain data item if it has not been set up properly. Otherwise inconsistent data may be accessed
  - e.g a linked list in shared memory requires proper link setting that may be corrupted when the two tasks try to concatenate a new element concurrently
- Errors due to the unexpected interleaving of actions over shared data structures are often referred to as ***race conditions***

A race condition example

- Suppose a very simple function void **`inck(void)`** that only contains the statement **`k = k+1`**.
- No real-world CPU is actually able to increment k in a single, indivisible step, at least when the code is compiled into ordinary assembly instructions.
- A simplified computer based on the von Neumann architecture will perform a sequence of three distinct steps:
  - 1. Load the value of k from memory into an internal processor register; From the processor's point of view, this is an external operation because it involves both the processor itself and memory The load operation is not destructive, that is, k retains its current value after it has been performed.
  - 2. Increment the value loaded from memory by one. Unlike the previous one, this operation is internal to the processor.
  - 3. Store the new value of k into memory with an external operation involving a memory bus transaction like the first one. It is important to notice that the new value of k can be observed from outside the processor only at this point, not before. In other words, if we look at memory, k retains its original value until this final step has been completed.



Race condition example - cont.

The following sequence of events may occur:

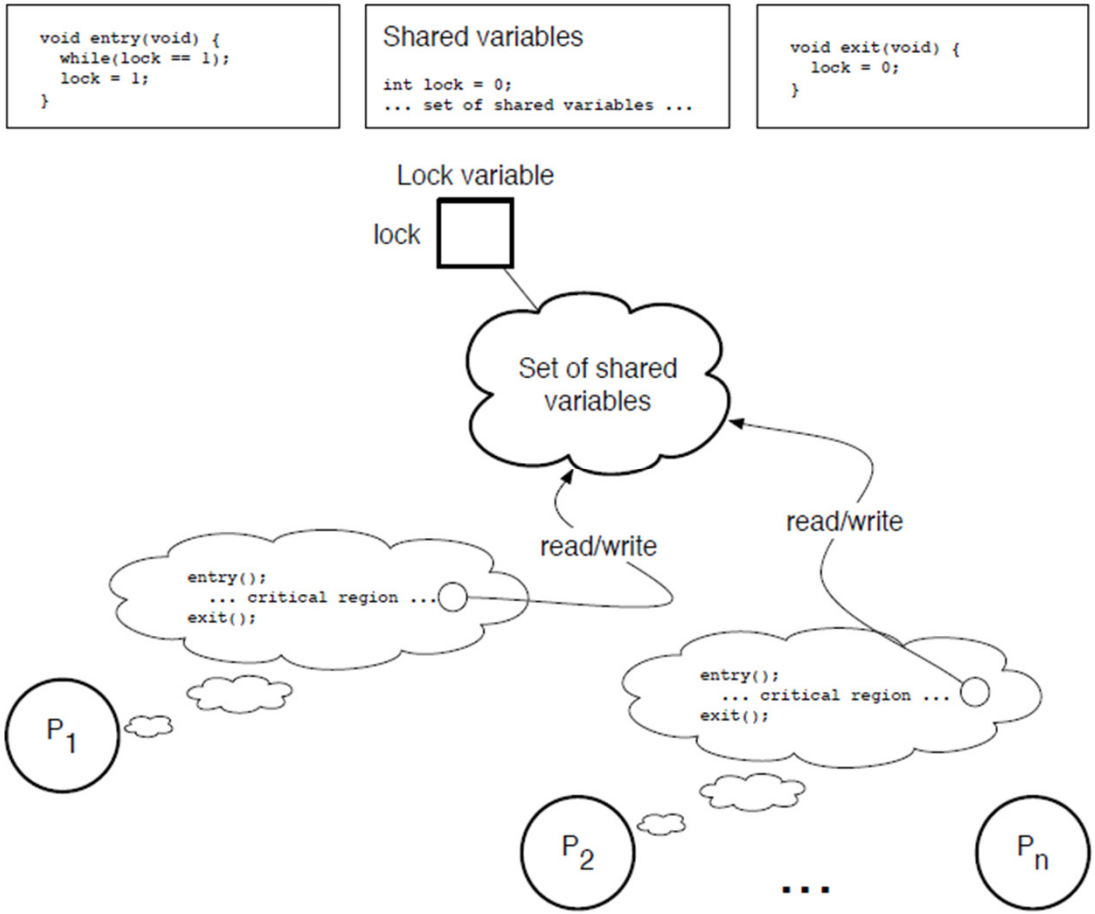
1. CPU #1 loads the value of k from memory and stores it into one of its registers, R1. Since k currently contains 0, R1 will also contain 0.
2. CPU #1 increments its register R1. The new value of R1 is therefore 1.
3. Now CPU #2 takes over, loads the value of k from memory, and stores it into one of its registers, R2. Since CPU #1 has not stored the updated value of R1 back to memory yet, CPU #2 still gets the value 0 from k, and R2 will also contain 0.
4. CPU #2 increments its register R2. The new value of R2 is therefore 1.
5. CPU #2 stores the contents of R2 back to memory in order to update k, that is, it stores 1 into k.
6. CPU #1 does the same: it stores the contents of R1 back to memory, that is, it stores 1 into k.

- Looking closely at the sequence of events just described, it is easy to notice that taking an (obviously correct) piece of sequential code and using it for concurrent programming in a careless way did not work as expected.
    - In the particular example just made, the final value of k is clearly incorrect: the initial value of k was 0, two distinct processes incremented it by one, but its final value is 1 instead of 2, as it should have been.
      Even worse: the result of the concurrent update is not only incorrect but it is incorrect only sometimes.
- Finally, observe that the same problem may arise with two processes/treads on the same processor

The need of mutual exclusion

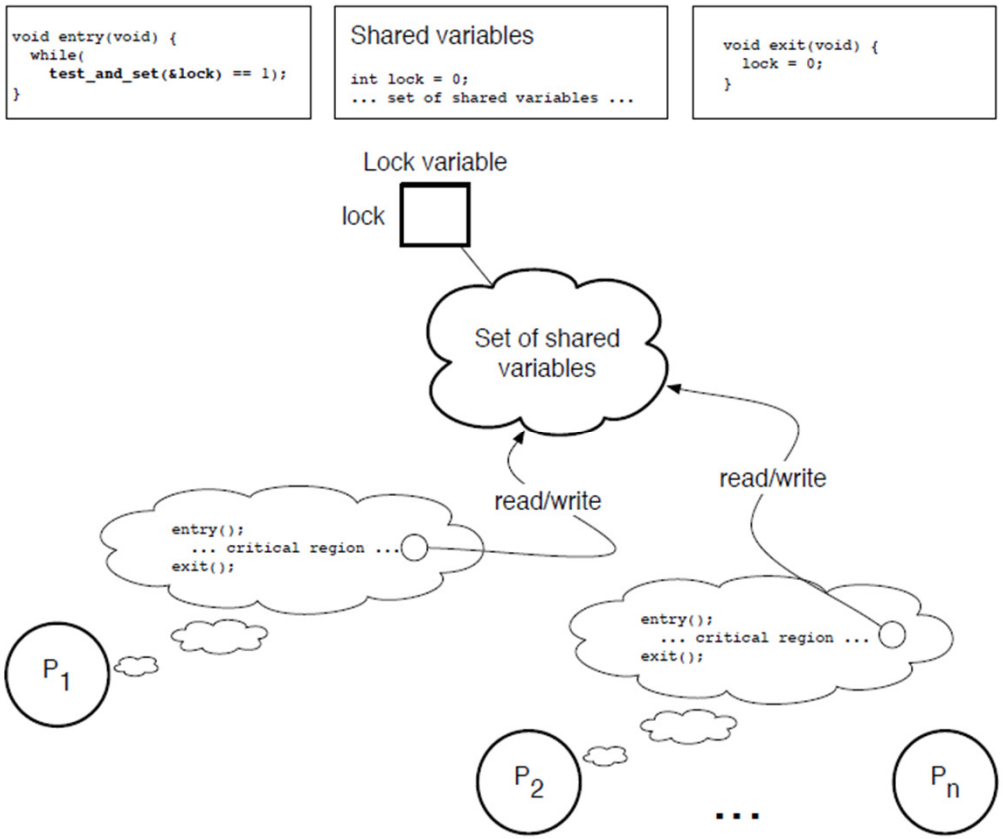
- A general solution is to protect **critical sections**, i.e. sections of code accessing shared information by means of **locks**, thus ensuring that only a task at a time is allowed to execute a section of code 'protected' by that lock
- A task that wants to access a shared object, by means of a certain critical region, must perform the following sequence of steps:
  - Acquire some sort of lock associated with the shared object and wait if it is not immediately available.
  - Use the shared object.
  - Release the lock, so that other processes can acquire it and be able to access the same object in the future
- In any case, four conditions must be satisfied in order to have an acceptable solution:
  - 1. It must really work, that is, it must prevent any two processes from simultaneously executing code within critical regions pertaining to he same shared object.
  - 2. Any process that is busy doing internal operations, that is, is not currently executing within a critical region, must not prevent other processes from entering their critical regions, if they so decide.
  - 3. If a process wants to enter a critical region, it must not have to wait forever to do so. This condition guarantees that the process will eventually make progress in its execution.
  - 4. It must work regardless of any low-level details about the hardware or software architecture. For example, the correctness of the solution must not depend on the number of processes in the system, the number of physical processors, or their relative speed.

A Naive Implementation



Hardware assisted solution

- Several processor provides a **`test_and_set`** instruction.
- This instruction has the address p of a memory word as argument and atomically performs the following three steps:
  - 1. It reads the value v of the memory word pointed by p.
  - 2. It stores 1 into the memory word pointed by p.
  - 3. It puts v into a register.
- An alternative solution uses XCHG instruction, which atomically exchanges the contents of a register with the contents of a memory word
- Both Instruction ensure atomicity even in a multicore architecture because they internally lock the memory bus



Software Solution: the Peterson algorithm

```
//Global Variables
int flag[2] = {0, 0};
int turn=0;

//Enter critical section
void entry(int pid) {
    int other = 1-pid;
    flag[pid] = 1;
    turn = pid;
    while(flag[other] == 1
      && turn == pid);
}

//Exit critical section
void exit(int pid) {
    flag[pid] = 0;
}
```

Lemma: ***pid in critical section => flag[pid] == 1***

Proof: pid in critical section => pid in critical section **or** pid in entry after instruction flag[pid] = 1 => flag[pid] == 1 (Invariant)

Theorem: At most one task is in critical section

Proof: At the time the first task just enters in critical section, the other task is either:

1. not calling entry() or in any case before instruction turn = pid
2. calling entry and after instruction turn = pid

In case 1 if the other task enters the while loop it finds the condition false: flag[other] = 1 (lemma) and turn == pid (turn is not changed by the other task in critical section)

In case 2 turn == pid, otherwise the first task would not have entered the critical section. For the same reason as before, the while loop finds the condition false

- Peterson algorithm is shown here for 2 tasks, but a similar algorithm is valid for N tasks
- Unlike previous HW solution, fairness is guaranteed here. A slower task is guaranteed to eventually enter the critical section (because of the variables **`flags`** and **`turn`**)

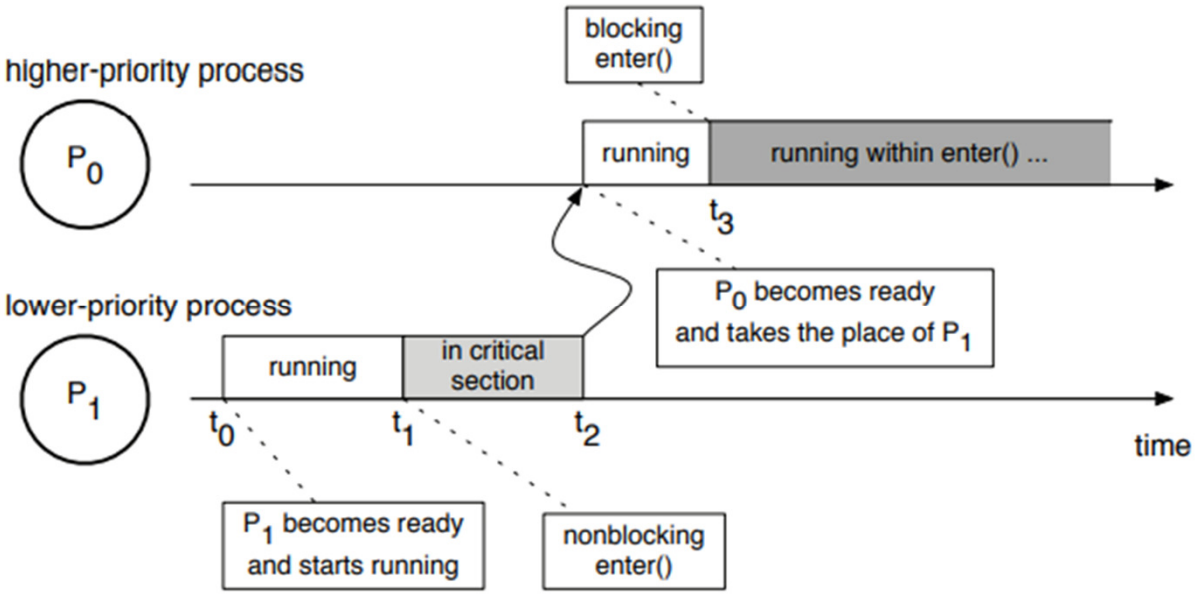
The Peterson algorithm for more than two processes

```
//Shared Variables
int level[N]
int last_to_enter[N-1]

void lock(pid)  //pid from 0 to N-1
{
        int h;
        for(h = 0; h < N-1; h++)
        {
                level[pid] = h;
                last_to_enter[h] = pid;
                while(last_to_enter[h] == pid &&  there exists k ≠ pid, such that
level[k] ≥ h);
        }
}

void unlock(int pid) {level[pid]= -1;}
```

From Active to Passive Wait

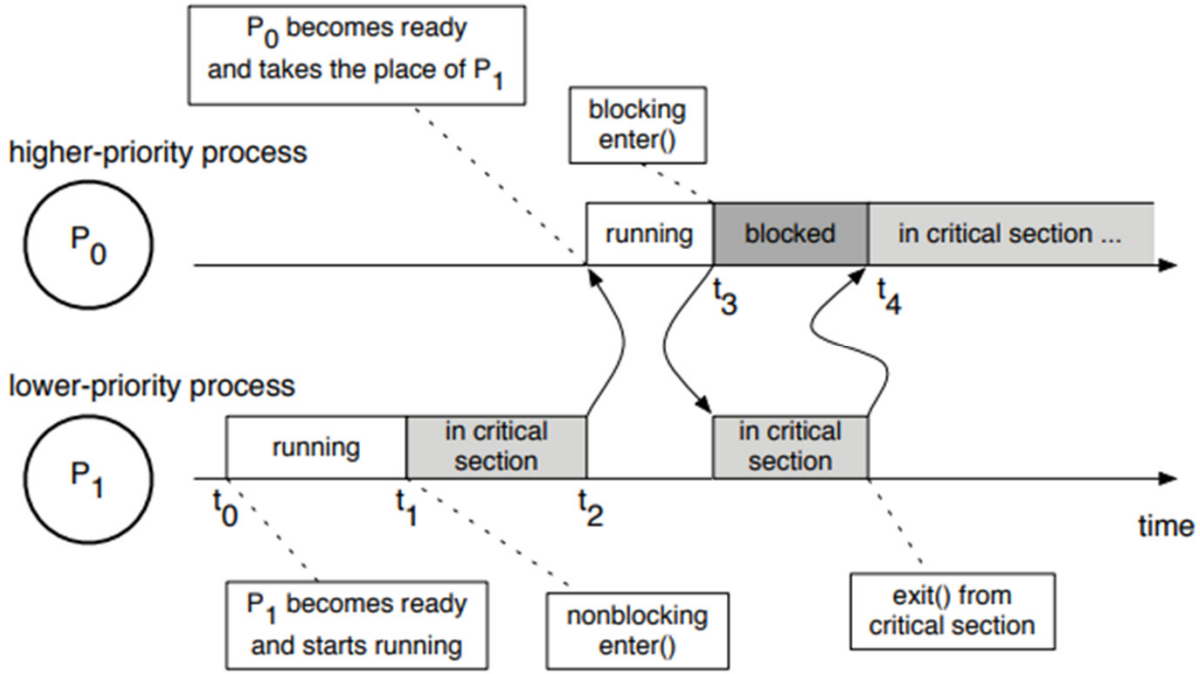
- The solutions seen so far imply a repeated check of s shared variable.
- This mechanism is similar to the polling I/O synchronization, but may waste processor cycles
  - Moreover, when the tasks share the same processor, the wait loop may affect the execution of the critical section leading to unbounded access time to the critical section
- A better solution would imply the OS removing the processor from a task trying to access a critical section already locked until it becomes free
- This implies making a request to the OS for entering and exiting a critical section
- In a single processor system OS data structures will replace the shared variables, and any task (process or thread) willing to access a critical section will be put in wait state if another task already gained access. When a task exits the critical section, the OS shall be informed, and tasks waiting for that critical section will be made ready again and eventually they will try again to enter the critical section.
- The implementation of the OS code for multicore systems is more complicated because the OS code itself is distributed
  - In this case HW specific locking instructions such as atomic exchanges shall be used internally in the OS as well as polling (***spinlock***)

Busy Wait for two processes with fixed priority



Indefinite wait (deadlock) may happen if the two processes share the same processor

Passive Wait for two processes with fixed priority



In this case dedlock does not occur

Semaphores

- A semaphore is an object that comprises two abstract items of information:
  - 1. a nonnegative, integer value;
  - 2. a queue of processes passively waiting on the semaphore.

- Upon initialization, a semaphore acquires an initial value specified by the programmer, and its queue is initially empty.

- Neither the value nor the queue associated with a semaphore can be read or written directly after initialization.

- On the contrary, the only way to interact with a semaphore is through the following two primitives that are assumed to be executed atomically:
  - 1. P(s), when invoked on semaphore s, checks whether the value of the semaphore is (strictly) greater than zero. If this is the case, it decrements the value by one and returns to the caller without blocking. Otherwise, it puts the calling process into the queue associated with the semaphore and blocks it by moving it into the Wait (Blocked) state of the process state diagram.
  - 2. V(s), when invoked on semaphore s, checks whether the queue associated with that semaphore is empty or not. If the queue is empty, it increments the value of the semaphore by one. Otherwise, it picks one of the blocked processes found in the queue and makes it ready for execution again by moving it into the Ready state of the process state diagram.

Using Semaphores to protect a critical section