chapter 5 Date No

Data inconsistency: may occur in the case of current access to shared data

race condition ⟶ several Processes access and manipulate the same data concurrently

↳ order of access changes outcome

Solution: orderly execution of cooperating Processes that share a logical address space (serial execution)

* The critical-section problem

- Critical section: Segment of code in which a Process may be changing common variables, updating tables, ...
- each Process must request Permission to enter it's critical section

code implementing request ⟶ entry section

exiting critical section ⟶ exit section

~~~ remainder section

Solution ⟶ 1. Mutual exclusion: if a Process is executing in it's critical section, no other Processes can be executing in their critical section

2. Progress: if some Processes wish to enter their critical section, only the Processes that are not in their remainder section can decide which will enter it's critical section next and it can not be Postponed indefinetely

3. there exits a limit on the number of times that other Processes are allowed to enter their critical section before another Process in it's entry section can execute it's critical section.

- no assumption concerning relative speed of n Processes

- Kernel data structures are prone to race conditions
~~~

- non-preemptive kernel → free from race condition
- preemptive kernel → more responsive than non-preemptive
  - → more suitable for real-time programming

* Peterson's solution → no guarantees that it will work on modern computers
  - → software based

```
do {
    flag[i] = true;          } shared variables
    turn = j                 } entry section
    while( flag[j] && turn == j);
        critical section
    flag[i] = false;         } exit section
        remainder section
} while(true);
```

* Hardware solutions ⇒ inaccesible to application programmers
- locking : Protecting critical regions through the use of locks
- atomic : uninterruptible unit

Hardware → uniprocessor → could disable interrupts
- → no preemption
- → boo inefficient

→ multiprocessor → provide special atomic hardware instructions
- → test memory word and set value
- → swap contents of memory words

executed atomically

* OS solutions (mutex locks) → Simplest
- mutex lock → software solution
  - → Mutual Exclusion
- the Process acquires a mutex lock when entering critical section ⇒ lock is released while exiting critical section ⇒ release()

acquire()

Date No

no context switch

acquire() {
    while (!available); → busy waiting → spinlock → waste CPU cycles
    available = false;
}

release() {
    available = true;
}

→ if a process acquires lock which is not available, that process is blocked

→ call to these functions must be atomic => implemented using hardware mechanism

* semaphore → more sophisticated than mutex locks

semaphore S → integer

└→ only accessed via wait() (P) and signal() (V)

wait(S) {
    while(S <= 0) { ; }
    S--;
}

signal(S) {
    S++;
}

busy waiting

improve

~~no interrupts~~ <= atomic *

typedef struct {
    int value;
    struct process *list;
} semaphore;

wakeup()

FIFO queue for waiting processes

add link field to PCB

- semaphore → binary → s s{0,1} → similar to mutex locks
  - └→ counting → range over an unrestricted domain
  - └→ can be used to control access to instances of a resource
- moved busy waiting from entry section to critical section
- % atomic → uniprocessors → disabling interrupts
  - └→ SMP → // → bad performance effect
    - └→ spinlock & compare & swap
      - test & set

semaphore)

- LIFO queue instead of FIFO ⇒ Starvation (Process waiting for
- Processes waiting for each other ⇒ deadlock

* Priority inversion
  - occurs in systems with more than two Priorities
  - two Priorities are not sufficient for general Purpose O.S.
  - solved by Priority-inheritance Protocol

* classic Problems of synchronization
  - bounded buffer Problem
  - dining-Philosophers Problem
  - reader_writer Problem

* Monitor → high level synchronization construct (abstract data type)

wrong semaphore → timing errors → hard to detect

→
- signal() before wait() ⇒ ~~mutual exclusion~~
- wait() instead of signal() ⇒ deadlock