Mainpoints: describes experiences wiht using monitors in a real operating system and the design changes to monitor that had to be made.

concurrent programming:

* local concurrent programming: an application be divided into co-running processes
* global resource sharing, e.g. processor
* replacing interrupts

two implementation methods: 1) shared memory(monitors) 2) message;

the reasons for not adopting a more simpler shared memory method: nonpreemption scheduler: 1) no restriction on the single processor setting; 2) separate from the scheduling method; 3) hard to process some preempted situations such as page faults.

the reason for not adopting semaphores: too little strcuturing discipline

=> \*\*monitors\*\*

enclasped, to apply lock implicitly

## Processes

wait and join of processes and subprocesses.

detach(p) -> no return(void), no wait(no join)

exception handler(attached to each block)

## monitors

the monitor machinism:

a process to operate the protected data by calling a monitor procedure.

monitor procedures: external procedures and internal procedures(can only be called inside a monitor procedure)

At most one process is in one monitor.

\delta: like a multi-process-safe function

### monitor modules

components:

* data
* entry procedures
* internal procedures
* external procedures(they donot have the requirement for only one process accessing at one time)

### deadlock

* programming bug in a monitor to cause two processes do a wait at the same time. No signals or awakening;
* cyclic calling patterns. solution: resource partial order
* M -> N -> waiting for a lock -> which another M’ call N release -> M’ wait for M to release

### monitored objects

each object a monitor to achieve the highest concurrency

problem: cost in duplicating the information

\tip: solution: **monitored record**: is just the instance of one monitor class, which provides with different granularities of lock

\tip: dead-lock prevention->priority resource allocation

dead-lock avoidance->banker’s algorithm[Banker's algorithm - Wikipedia](https://en.wikipedia.org/wiki/Banker%27s_algorithm)

Lock on monitor class > lock on monitor record (for granularity)

### unwind

the father procedure can send unwind exception to other children procedures when an exception occurs in the last children.

and those abandoned procedures have to restore the variables as well as release the lock.

## condition variables

Mesa’s treatment of notification

signal in the Mesa’s monitor: just for a notification for a process can be run in the queue, and do not immediately awake a process in the queue to run.

\tip: **reason**: 1) remove the scheduling out of the monitor. 2) less context switch(the one doing notification does not need to be switched out immediately, rather be switched out when there is waiting situation).

benefits:

* leave flexibility for schedule outside the monitor.

ways to subtittule notification:

* abort
* timeout
* broadcast

### naked notify: it’s like an atom primitive, which is never blocked.

### priority problem

the conflict between the priority and the monitor lock.(priority inversion)

(Java doen’t worry about that, just wait, the problem is important for some critical systems)

solution1 - priority inherit -> temporarily assign higher priority to those who hold resources.

solution:

* assign monitor priority
  + e.g. A has priority 1, B has priority 3, C has priority 2, A requires M and N, B requires M and N, B already gets M, then although A is more privileged than B, A cannot grab N, thus B is assigned higher priority as A to let M resource release more quickly.
* interrupt

## Implementation

QA:

Java does not use explicit condition variables for monitors because it’s an easier way for developers.

Java now use a signal to notify a thread’s killed, which ensures the consistent states for condition variables.

The methods of deadlock prevention: break the circular dependencies, banker’s algorithm; two-phase; or have order for resource requirers.