A Brief Introduction to OS/2 Multithreading

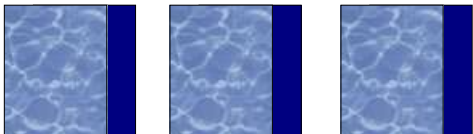
Jaroslav Kaƒer

jkacer@kiv.zcu.cz

University of West Bohemia

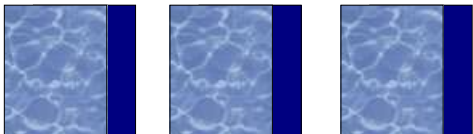
Faculty of Applied Sciences

Department of Computer Science and Engineering



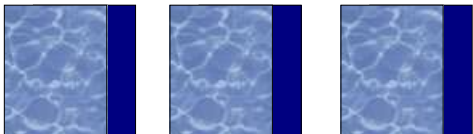
Why Use Parallelism?

- Performance increase
- Distributed computing
- OS/2: Usually 1 processor
- But useful too: Pretended performance increase has a big influence on user's comfort



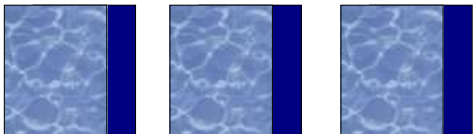
Job vs. Program

- Job:
  - Dynamic structure: processes, threads, ...
  - Activity
  - Has a state
- Program:
  - Static structure: procedures + shared data declaration
  - Just a description - "template" for a job



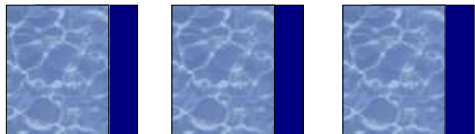
Working with Data

- Local data
  - Accessible by one process/thread only
  - No problems
- Global (shared) data
  - Accessed by more processes
  - Need to protect them $\Rightarrow$ Process synchronization



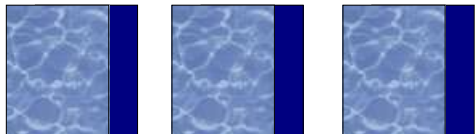
Global (Shared) Data

- Independent - Access without synchronization
- Dependent - Acces with synchronization
  - Locked variables
  - Reduction variables
  - Ordered variables



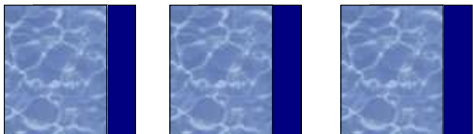
Process Synchronization

- Semaphores and locks
  - Semaphores: Passive waiting
  - Locks: Active waiting
  - OS/2 API, POSIX
- Monitors
  - Monitor procedures
  - Java, ADA



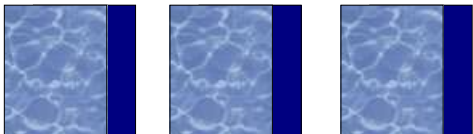
Architectures

- Shared memory
  - Symmetric multiprocessors (OS/2)
  - Assymetric multiprocessors
- Distributed memory
  - Communication network
  - PVM, MPI, etc.



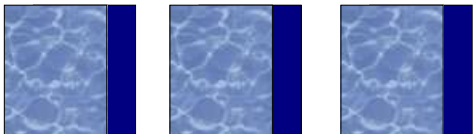
Realization of Parallel Processes

- Independent jobs of an OS (Coarse grain)
  - Distributed memory systems
  - Message passing
- Threads (Fine grain)
  - Shared memory systems (OS/2)
- Co-routines
  - Pseudo-parallelism



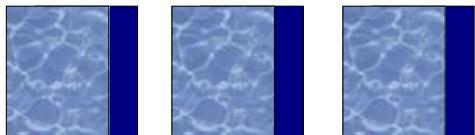
A Process Has:

- Program
- Local data
- State
  - Local data values
  - Values of processor registers
  - Stack
  - A point in its program



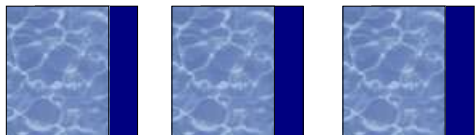
Process Control

- Time slices
- Preemtive multitasking
  - Priorities
- Control of Co-routines



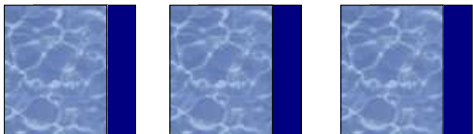
Parallelism Entities in OS/2

- Sessions
- Processes
  - Containers for threads and resources, PID
  - At least one thread
- Threads
  - Get processor time
  - Priority class + level, TID
  - System limit: 4096, THREADS



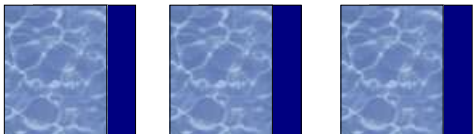
Processes

- DosExecPgm()
- DosKillProcess()
  - KILLPROCESS exception
  - DosSetExceptionHandler()
  - DosExit()
- DosWaitChild()



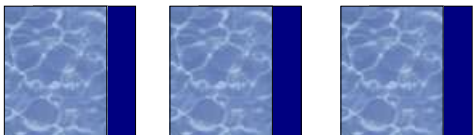
Sessions

- 5 types
- DosStartSession()
- DosStopSession()
- DosSelectSession()
- DosSetSession()
- STARTDATA
- DosQueryAppType()



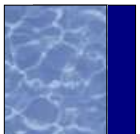
OS/2 Scheduler

- 4 priority classes
  - Time-critical, server, regular, idle-time
- 32 levels within a class
- Boosting
  - Foreground boost
  - I/O boost
  - Starvation boost



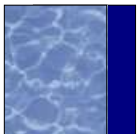
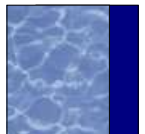
OS/2 API for Multithreading

- #define INCL_DOS
- #include <os2.h>
- gcc switch: -Zmt
- Open Watcom switch: -bm



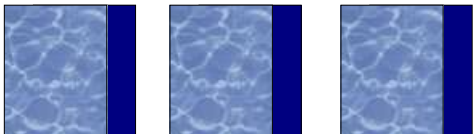
Thread Creation

- DosCreateThread()
  - VOID APIENTRY fnThread(ULONG ulArgs)
- _beginthread()
  - void *thread(void *args)
  - _endthread()


Other API

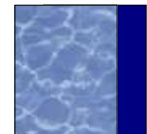
- DosExit()
- DosSetPriority()
- DosSuspendThread()
- DosResumeThread()
- DosKillThread()
- DosWaitThread()
- DosSleep()



Critical Section

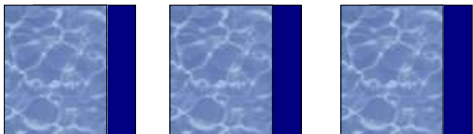
- Atomicity of operations inside a CS
- Protection of shared data
- No switching inside a CS
- DosEnterCritSec()
- DosExitCritSec()
- Dangerous! $\Rightarrow$ No semaphores inside a CS!





Mutex Semaphores

- Protection of shared data
- "Mutual Exclusion"
- Only 1 thread can acquire a mutex
- Others are blocked
- "Opening and closing brackets" of protected code

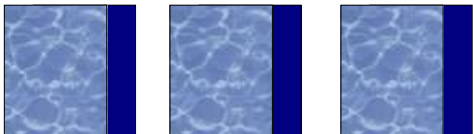


A Problem

- 2 (or more) threads, a shared variable, local variables i

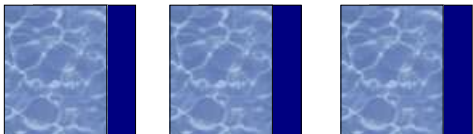
1. int i;
2. read(i);
3. i = i+1;
4. write(i);

- Result: unpredictable (1 or 2)



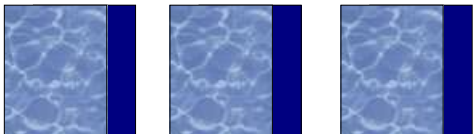
A Solution

1. int i;
2. mutex m;
3. mutex_lock(m);
4. read(i);
5. i = i+1;
6. write(i);
7. mutex_unlock(m);



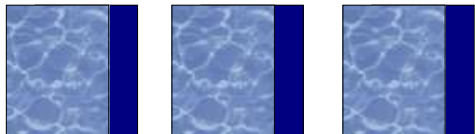
OS/2 API for Mutex Manipulation

- DosCreateMutexSem()
- DosOpenMutexSem()
- DosCloseMutexSem()
- DosQueryMutexSem()
- DosRequestMutexSem() / WinRequestMutexSem()
- DosReleaseMutexSem()



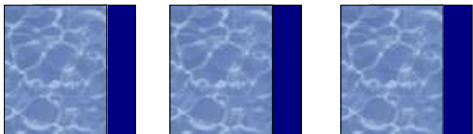
Event Semaphores

- 2 (or more) threads
- One waiting for an event to happen (blocked)
- The other signal the event to the first one
- The first one is restarted
- Analogy of traffic lights



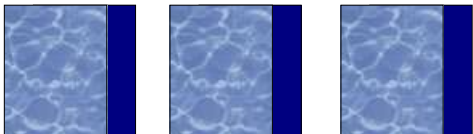
OS/2 API for Event-Semaphores Manipulation

- DosCreateEventSem()
- DosOpenEventSem()
- DosCloseEventSem()
- DosQueryEventSem()
- DosResetEventSem()
- DosPostEventSem()
- DosWaitEventSem() / WinWaitEventSem()



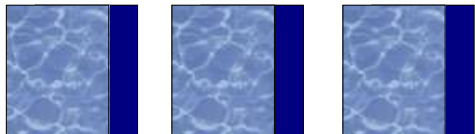
Mux-Wait Semaphores

- A bundle of mutex and event semaphores
- Adding / removing to / from a mux-wait
- Similar API +
  - DosAddMuxWaitSem()
  - DosDeleteMuxWaitSem()

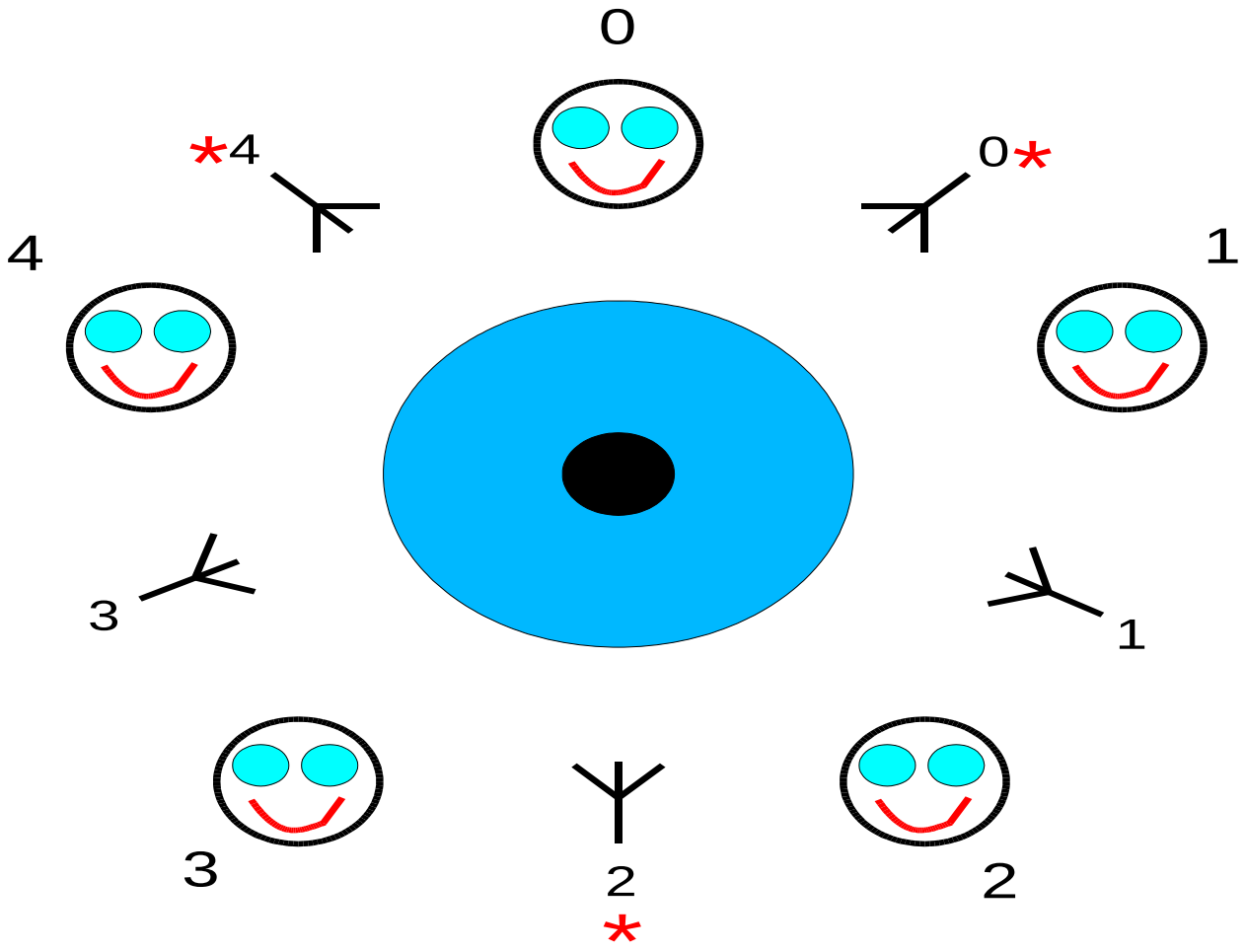


Dining Philosophers

- E. Dijkstra, 1970s
- Round table
- N philosophers eating spaghetti and thinking
- N forks
- A philosopher must get both forks before eating

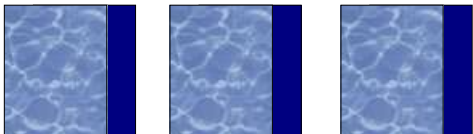


Dining Philosophers



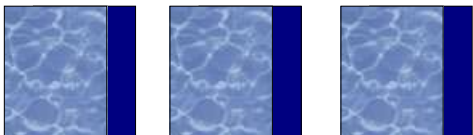
Deadlock Possible

- First left fork, then right fork
- Everybody has his left fork
- Nobody has his right fork
- Nobody can release his left fork
- Everybody blocked forever



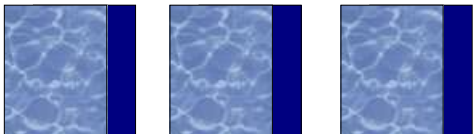
A Deadlock Occures When:

- Shared resouces
- One resource can be given to at most one process
- A process waits indefinitely for a resource
- Only the process having a resource can release it
- Cyclic dependency
  - P1 has R1 and wants R2, and
  - P2 has R2 and wants R3, and ...
  - Pn has Rn and wants R1.



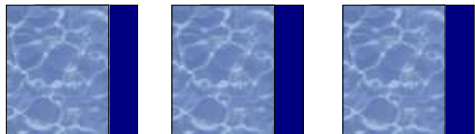
Solutions

- Detection methods
  - Allocation graph
  - Banker's algorithm
- Protection methods
  - E.g. Different priorities of shared resources
  - Processes must allocate resources in ascending order
  - Sufficient for DP: 2 priorities only



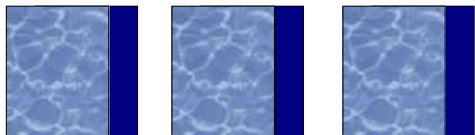
OS/2 API Solution

- Arrays of forks and philosophers
- A fork holds info about its owner, protected with a mutex semaphore
- Every philosopher knows his left, right, first and second fork
- A philosopher's life = a thread
- A philosopher must acquire both mutexes, possibly blocked



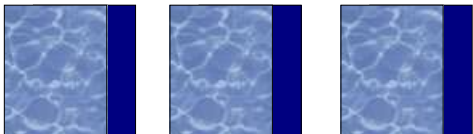
POSIX Threads Solution (1)

- Almost the same as the OS/2 solution
- Different types
  - pthread_t
  - pthread_mutex_t
  - pthread_cond_t



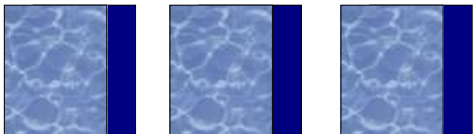
POSIX Threads Solution (2)

- Different functions
  - pthread_create(), pthread_mutex_init(), pthread_mutex_destroy(), pthread_cond_init(), pthread_cond_destroy()
  - pthread_mutex_lock(), pthread_mutex_unlock()
  - pthread_cond_wait(), pthread_cond_signal()



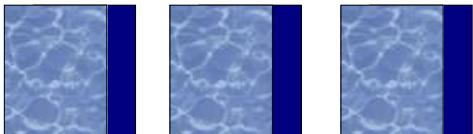
Java Solution (1)

- Object oriented: classes Fork and Philosopher
- Java threads
- Monitors - no semaphores, no manual waiting
- Object references, no arrays and indices
- Easy, short, elegant



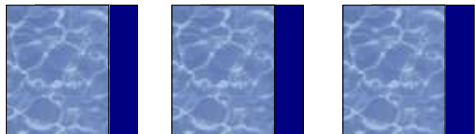
Java Solution (2)

- A fork:
  - Monitor methods acquireFork() and releaseFork()
  - About 5 lines of code each
  - synchronized, wait(), notify()



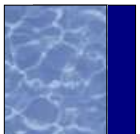
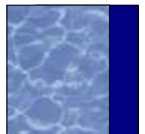
Java Solution (3)

- A philosopher:
  - A thread - class Philosopher extends Thread
  - Method run() overridden
  - No care about locking - done in the Fork class
  - Just calls acquireFork() and releaseFork() of both forks



Where to Get It

- http://home.zcu.cz/~jkacer/cz/os2
- http://www.os2.cz/warpstock (Paper only)
- Czech Warpstock 2002 CD


Thank You!