Systèmes Embarqués
PFSEM 2007 - 2008

Real-Time & Embedded OS Principles

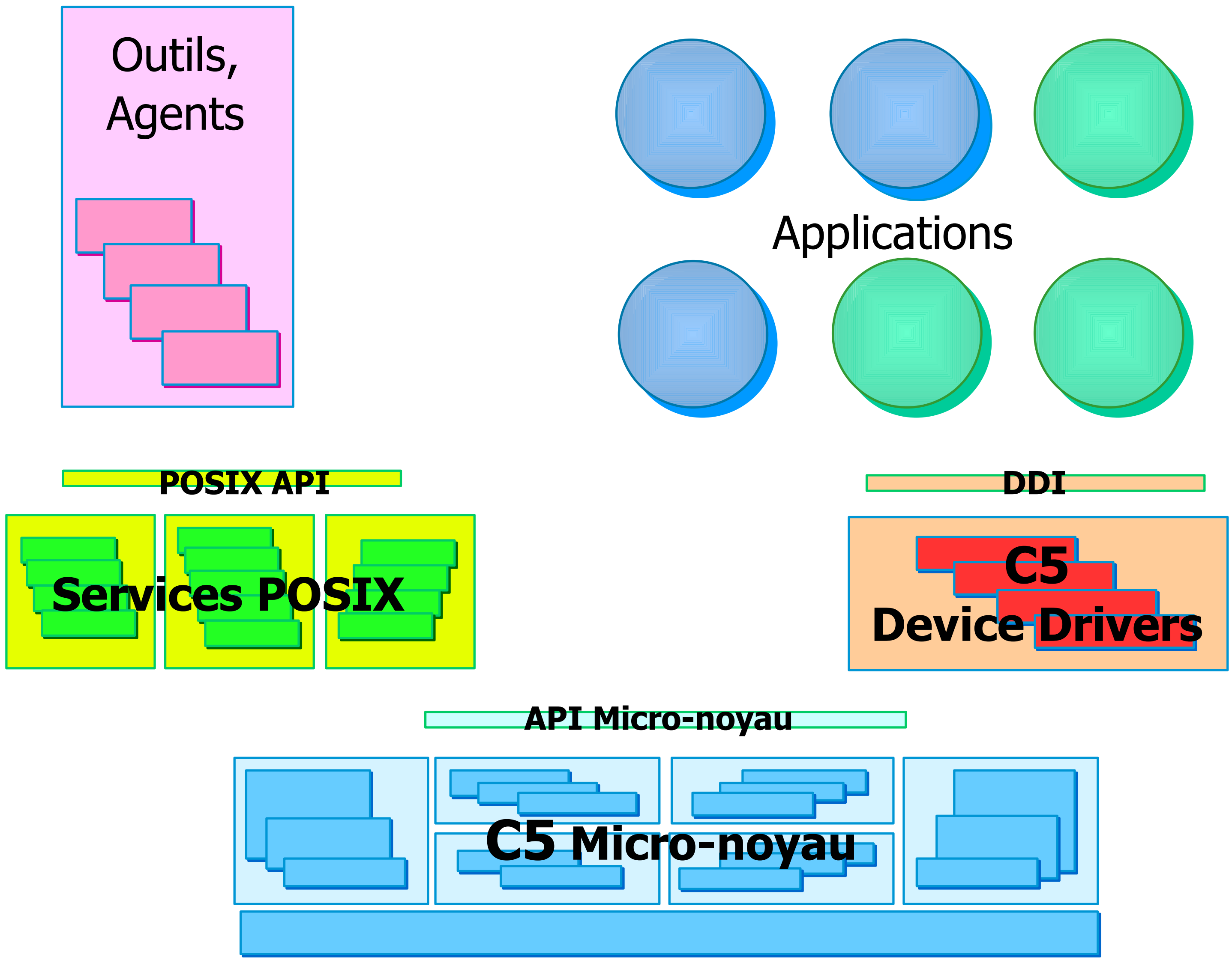
Use case: ChorusOS

Plan

- Environnement de Développement
- Micro-noyau C5 (ChorusOS®)
- Device Driver Framework
- Personnalité POSIX
- Voir "Programming Under Chorus", Jean-Marie Rifflet

  http://www.pps.jussieu.fr/~rifflet/PUBLICATIONS/book4.html

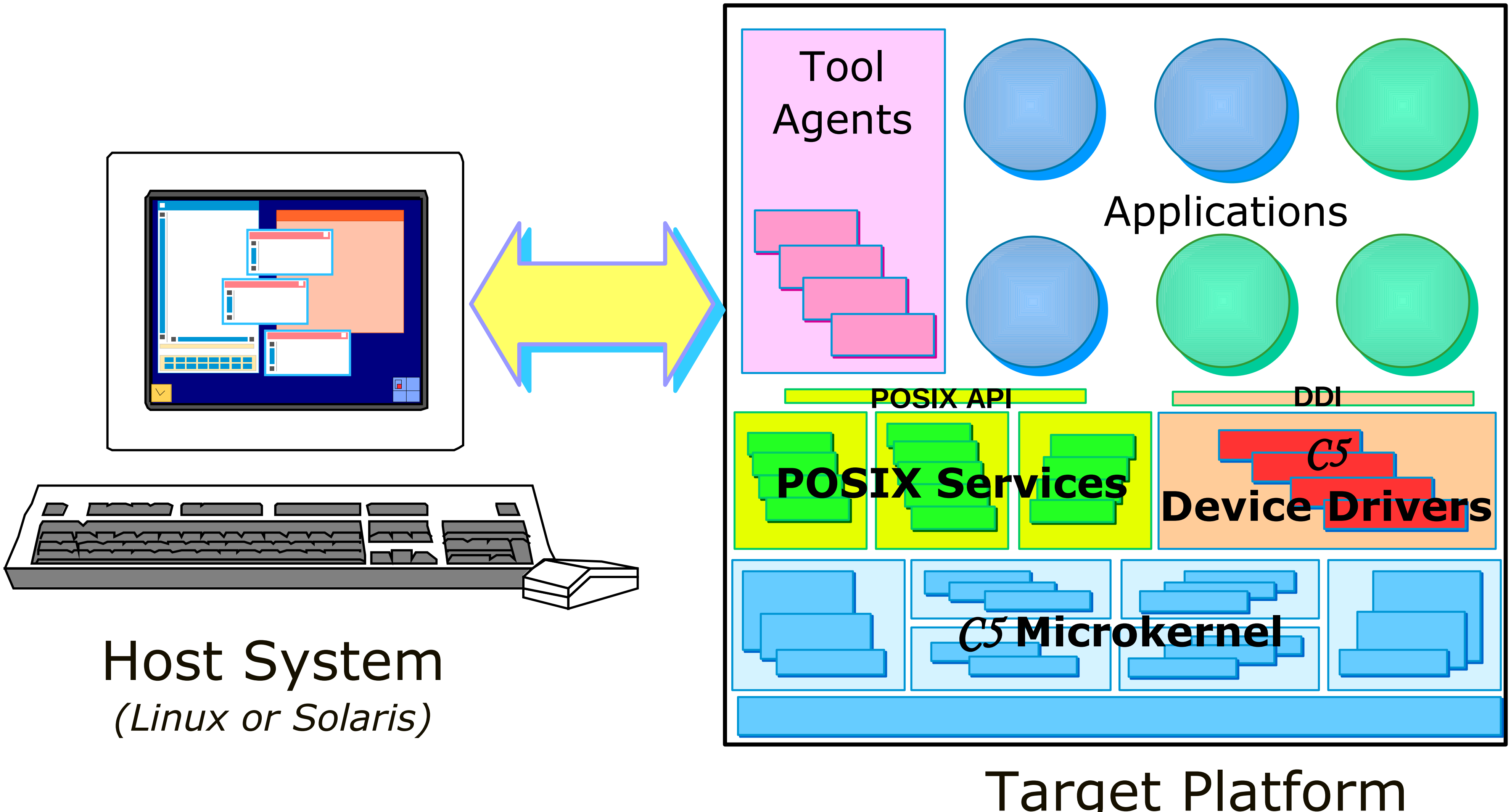
OS Architecture



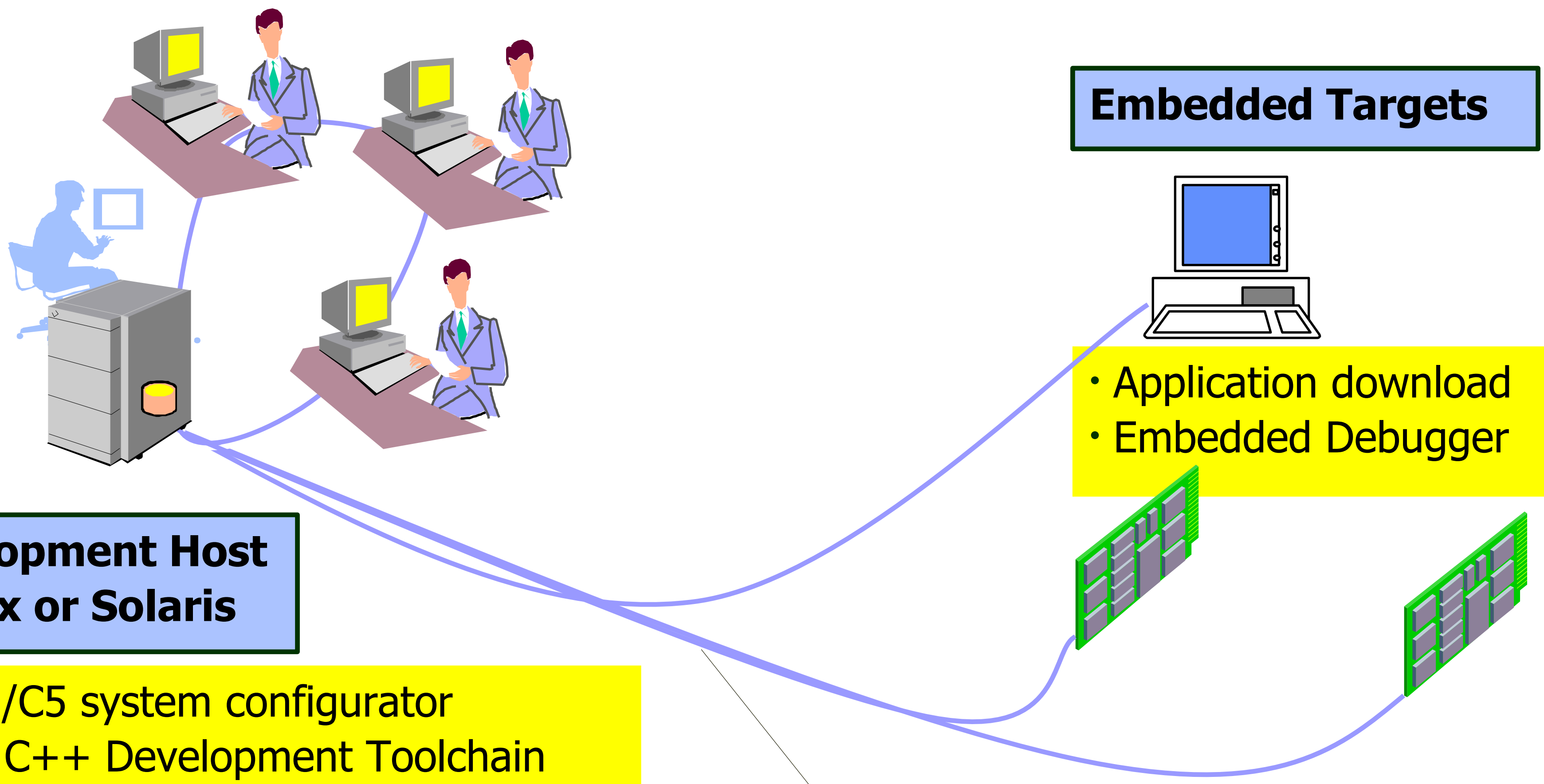
ChorusOS - Système Temps-Réel

- Environnement de développement croisé (*Host/Target*)
- Commandes d'administration embarquées
- C5 micro-noyau (*Chorus* 5ème génération)
  - Comportement temps-réel garanti
  - Verrouillage à grain fin (*fine-grain locking*)
  - Gestion(s) mémoire flexible
  - Device Drivers Framework
  - Debugger système (kdb) intégré

Host/Target System Environment



Environnement de Développement



Embedded Targets

- Application download
- Embedded Debugger

Development Host
Linux or Solaris

- Jaluna/C5 system configurator
- C and C++ Development Toolchain
- C and C++ Symbolic Debugger
- Application management utilities
- Set of libraries

Liens:
- Ligne série
- Ethernet
- JTAG

Développement Host/Target

- Compilateur croisé GNU ANSI-C et C++
  - Génère du code pour la machine cible sur la machine hôte
  - Ex: code PowerPC sur un PC/Intel
- Compilation croisée canadienne
  - (*Canadian Cross-Compilation*)
  - Host/Target appliqué au compilateur

Développement Host/Target

- Compilateur croisé
  - Utilise les “header files” du système cible
  - Répertoire par défaut header files n'est pas `/usr/include`
- Éditeur de liens croisé
  - Utilise bibliothèques du système cible
  - Répertoire de base n'est pas `/usr/lib`

Développement Host/Target Chorus

- Utilisation de Imake et règles de productions prédéfinies
  - Règles spécifiques pour produire binaires de drivers, d'applications superviseur et user
  - Utilitaire pour transformer un fichier imake en un fichier Makefile
- Mêmes outils pour produire le système, les drivers et les applications

Image système de boot

- Un seul "fichier" contenant le système (global) qui sera exécuté sur la machine cible
- bootstrap
- OS : micro-noyau et drivers
- Applications à démarrer automatiquement
- Variables de configuration, commandes de lancement

Configuration du Système

- Modules système (*optional features*)
- Drivers
- Applications
- Variables de configuration du système
  - Nombre de threads, etc.
  - Priorités de threads du système, etc.
- Variables d'environment (ala Unix)
  - Exemple : IP_ADDR=129.157.173.10

Lancement d'une application

- Lancement interactif
  - Services de communication standard
    - network links
    - NFS protocol
  - Outils standard sur machine hôte
    - <**rsh target binary-name**>
  - => phase de debug/outils d'administration
- Lancement automatique au boot du système
  - applications binaires inclues dans l'image de boot
  - Automatiquement lancées par le système
  - => dans les systèmes enfouis déployés

Debug Applications

- gdb (Gnu Debugger)
  - Pas embarqué sur la cible !!!
- Croisé sur la machine hôte
  - Ex: gdb pour PowerPC sur Intel
  - Protocole avec un agent sur la cible
  - Table des symboles sur la machine hôte
  - Adapté au debug d'applications multi-threads

Debug Système et Drivers

- « KDB » Chorus
  - Debugger integré dans micro-noyau
  - Interagit avec utilisateur par le biais de la console système
  - Traces via système de journalisation (buffer circulaire)
- Outils spécifiques : JTAG

Microkernel Debugger (KDB)

- Automatiquement appelé en cas de :
  - Erreurs détectées par le CPU (adresse invalide,..)
  - Tests de cohérences faux (compilation en mode debug)
  - Exécution sur un point d'arrêt
  - Invocation explicite - **`callDebug()`**
- Arrête toutes les activités du système, y compris les interruptions
- Utilise table de symboles stockés en mémoire

Services d'affichage de KDB

- Etat des objets du microkernel
  - Threads, régions mémoire, etc.
- Pile superviseur/utilisateur des threads
- Mémoire en unités de byte/short/long
- Registres du CPU
- Désassembleur
- Enregistrements du “journal” (avec options de filtrage)

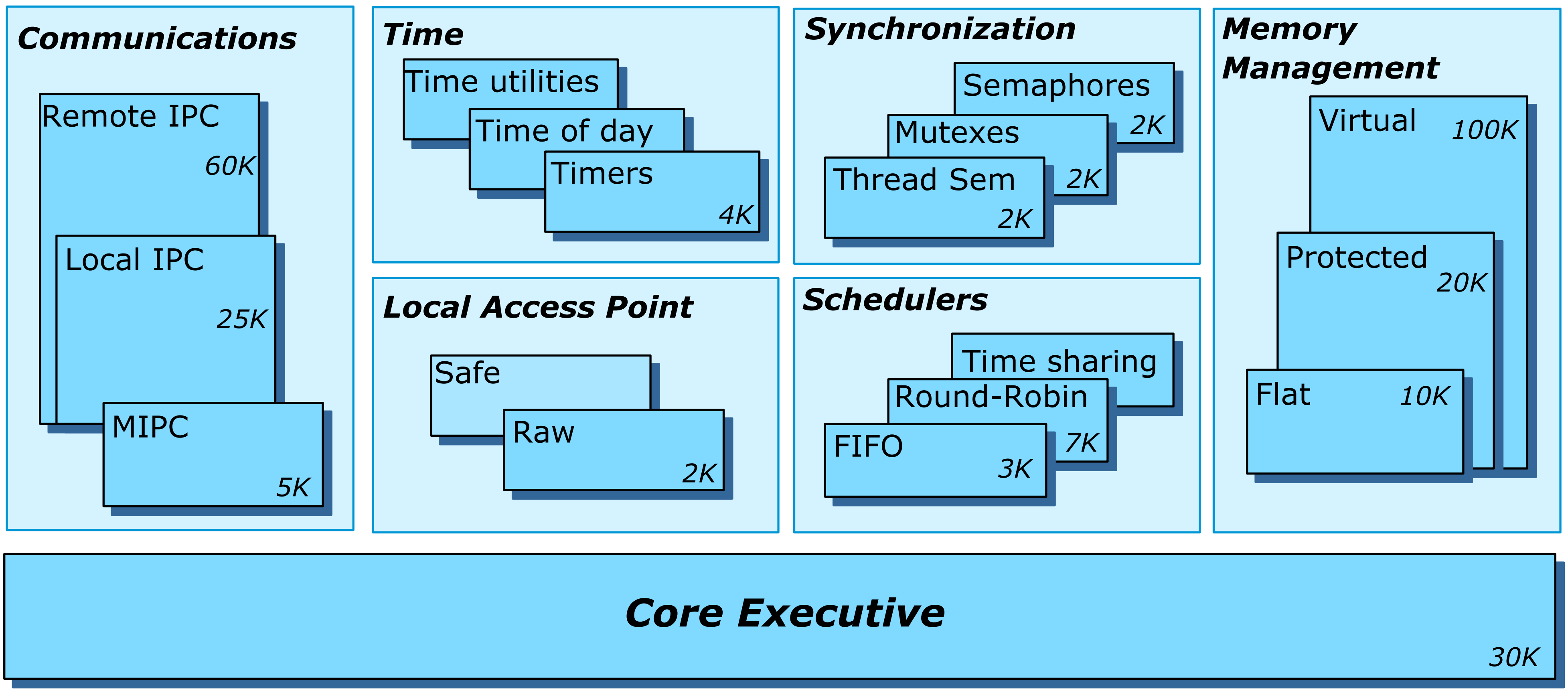
Commandes de KDB

- Set/reset breakpoints
  - Points d'arrêt logiciel
  - Points d'arrêt sur donnée (si disponible sur CPU)
- Exécution en mode “pas-à-pas”
  - Instruction (1 par défaut)
  - Stop au prochain appel de fonction
- Modifier un mot mémoire / un registre
- Appeler une fonction avec des arguments fournis par l'utilisateur

Plan

- Environnement de Développement
- Micro-noyau C5 (ChorusOS®)
- Device Driver Framework
- Personnalité POSIX

C5 Micro-kernel Features



Core Executive

- Basic execution services
  - actors
  - threads
  - synchronization services
- Kernel modules support
  - initialization
  - per thread/actor data
- Boot actors launching

Actors

- Equivalent à la notion de processus
- Contexte d'attribution de la plupart des ressources allouées dynamiquement
  - threads
  - mémoire (memory regions),
  - MIPC message spaces,
  - etc...
- Resources libérées lors de la destruction de l'acteur

Actor Types

- User actor
  - private [protected] memory address space
  - provided with optional PRM or VM memory modules
- Supervisor actor
  - share single system address space
- System actor = trusted actor
  - specific rights
  - all supervisor actors are trusted

Actors API

- Designated by a capability : `KnCap`
  - Unique Identifier (64 bits) + Key (64 bits)
  - K_MYACTOR designates current actor
- My capability
  - actorSelf(&myCap);
- Create an actor
  - res = actorCreate(&parentCap, &newCap,
    privilege, status);
- Delete an actor
  - res = actorDelete(&cap);

Threads

- Unit of program execution
- Sequential execution context
- Owned by an actor (home actor)
- Local Identifier (scope = home actor)
- States
  - not yet activated
  - active (running or ready)
  - waiting
  - stopped

Thread Execution Modes

- Two CPU Execution Modes
  - user => no privilege
  - supervisor => all privileges
- Execution actor
  - usually home actor
  - other supervisor actor after
    - trap instruction
    - cross-actor invocation (LAP)

Thread API

- Designated by a "Local Identifier"
  - actorCap + threadLi
- thLi = threadSelf();
- K_MYSELF designates current thread
- res = threadCreate(&cap, &thLi, status,
  &schedParam, &startInfo);

  status: **K_ACTIVE** or **K_INACTIVE**
  schedParam: thread priority, etc.
  startInfo: PC, stack bottom
- res = threadDelete(&cap, thLi);

Pile(s) d'exécution des threads

- Pile système = pile allouée par le noyau pour exécution des appels système
- Pile applicative = pile utilisée pour exécution du code applicatif
  - supervisor thread : pile système par défaut
  - user thread : pile utilisateur allouée et libérée
    - par noyau si `K_START_INFO_USER_POSIX` flag
    - par application sinon

Pile(s) d'exécution ...

- User thread always created by another thread
  - Allocate memory for application stack of new thread
  - `threadCreate()`
- User threads usually delete themselve, but:
  - `threadDelete(K_MYACTOR, K_MYSELF)` does not return
  - Deleting thread cannot free memory of its application stack before deleting itself
  - => both operations must be done by another thread

Synchronization Mechanisms

- Based on synchronization objects
  - represented by data structures in application space
  - no system limitation
- Mutexes
- Real-time mutexes
- Semaphores
- Thread semaphores

Synchro / Mutex

- Ensure exclusive access to shared objects
- Between [concurrent] **threads** only
- No recursivity
- No deadlock detection
- FIFO ordering of waiting threads
- Programming model

```
mutexGet(&mutex);
  /* access shared object */
mutexRel(&mutex);
```

Synchro / RT-Mutex

- Address priority inversion issue
- Prio(T1) < Prio(T2) < Prio(T3)
    - T1 acquires mutex M1
    - T1 preempted by T3
    - T3 blocked to acquire mutex M1
    - T1 preempted by T2

Inverts execution order of T2 and T3

Synchro / RT-Mutex (2)

- Specific data structure & system calls
- Records owning thread of RT-mutex
- Temporarily assigns to owning thread the [higher] priority of concurrent thread
- Transmits mutex ownership to thread waiting with the highest priority
- Manages sequential acquisition of multiple RT-mutexes

Synchro / Semaphore

- Producer/Consumer synchronization
- Includes a counter initialized with any value >= 0

```
semInit(&sema, initValue);
KnSem sema = K_KNSEM_INITIALIZER(initValue);
```

- Producers: threads and interrupt handlers

```
semV(&sema);
```

- Consumers: threads only

```
semP(&sema, WaitDelay);
```

- FIFO ordering of waiting threads

Synchro / Thread Semaphore

- Per-thread synchronization mechanism
- Binary state
  - posted
  - cleared
- More efficient than standard semaphore
  - no counter
  - no list of threads
- Usage
  - controlled selection of awoken thread
  - single "consumer" thread

Scheduling

- Microkernel highly preemptive
  - fine-grain locking policy in all real-time services
  - specific [global] lock for non real-time services (memory mngt, DDI, etc.)
- Multi-class scheduling
  - priority-based preemptive FIFO (default real-time)
  - priority-based round-robin
  - UNIX time-sharing

Coarse-grain Locking

- Use a global lock to protect a set of various ressources
- Functions which access ressources
  - Keep lock for the duration of their task
  - May invoke other functions which need also to acquire [global] locks
- **+** simple, efficient, easier to avoid deadlocks
- **-** not real-time compliant

Fine-grain Locking

- Use a lock for each specific resource
- Functions which access resources
  - Acquire lock each time they [need to] access a given resource
  - Never keep lock(s) when invoking other functions
- **+** real-time compliant
- **-** complex, deadlocks more likely to happen

Exemple : Message Ressource

- Minimize memory fragmentation
  - Messages composed of multiple fixed size memory “chunks”
- chunks allocated in a “peace-meal” fashion
- Control memory usage
  - Limited pool of memory chunks

Message API

```
#define CHUNK_SIZE 100
/* wait for available chunks if needed */
extern void* chunk_alloc();
/* awake waiting thread, if any */
extern void  chunk_free(void* chk);

typedef unsigned int msg_size_t;
typedef struct {
  msg_size_t size;
  void*      proto_header;
  void*      first_chunk;
  void*      last_chunk;
} msg_t;

extern msg_t* msg_alloc(msg_size_t size);
extern void   msg_free(msg_t* msg);
```

Msg Allocation – Coarse Grain

```
  msg_t*
msg_alloc(msg_size_t size)
{
  msg_coarse_lock_get();
  msg  = (msg_t*)chunk_alloc();
  msg->size = 0;
  while (msg->size < size) {
    chk = chunk_alloc();
    add_chunk_to_msg(msg, chk);
    msg->size += CHUNK_SIZE;
  }
  msg_coarse_lock_rel();
  msg->size = size;
  msg->proto_header = (char*)msg +
                      sizeof(msg_t);
  return msg;
}
```

Msg Free – Coarse Grain

```
  void
msg_free(msg_t* msg)
{
  msg_coarse_lock_get();
  while (msg->size > CHUNK_SIZE) {
    chk = rm_chunk_from_msg(msg);
    chunk_free(chk);
    msg->size -= CHUNK_SIZE;
  }
  if (msg->size > 0) {
    chk = rm_chunk_from_msg(msg);
    chunk_free(chk);
  }
  chunk_free(msg);
  msg_coarse_lock_rel();
}
```

Chunk Mngt – Coarse Grain

```
 void*
chunk_alloc()
{
  while(chunk_pool_empty){
    wait_list_add(w_ctx);
    msg_coarse_lock_rel();
    wait(w_ctx);
    msg_coarse_lock_get();
  }
  chk = chunk_pool_rm();
  return chk;
}
```

```
 void
chunk_free(void* chk)
{
  chunk_pool_add(chk);
  if (wait_list_empty){
    return;
  }
  ctx = wait_list_rm();
  awake(ctx)
}
```

Msg Allocation – Fine Grain

```
  msg_t*
msg_alloc(msg_size_t size)
{
  msg  = (msg_t*)chunk_alloc();
  msg->size = 0;
  while (msg->size < size) {
    chk = chunk_alloc();
    add_chunk_to_msg(msg, chk);
    msg->size += CHUNK_SIZE;
  }
  msg->size = size;
  msg->proto_header = (char*)msg +
                         sizeof(msg_t);
  return msg;
}
```

Msg Free – Fine Grain

```
  void
msg_free(msg_t* msg)
{
  while (msg->size > CHUNK_SIZE) {
    chk = rm_chunk_from_msg(msg);
    chunk_free(chk);
    msg->size -= CHUNK_SIZE;
  }
  if (msg->size > 0) {
    chk = rm_chunk_from_msg(msg);
    chunk_free(chk);
  }
  chunk_free(msg);
}
```

Chunk Mngt – Fine Grain

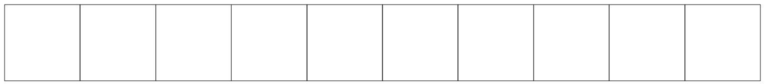
```
 void*
chunk_alloc()
{
  chk_fine_lock_get();
  while(chunk_pool_empty){
    wait_list_add(w_ctx);
    chk_fine_lock_rel();
    wait(w_ctx);
    chk_fine_lock_get();
  }
  chk = chunk_pool_rm();
  chk_fine_lock_rel();
  return chk;
}
```

```
 void
chunk_free(void* chk)
{
  chk_fine_lock_get();
  chunk_pool_add(chk);
  if (wait_list_empty){
    chk_fine_lock_rel();
    return;
  }
  ctx = wait_list_rm();
  chk_fine_lock_rel();
  awake(ctx)
}
```

Memory Deadlock (1)

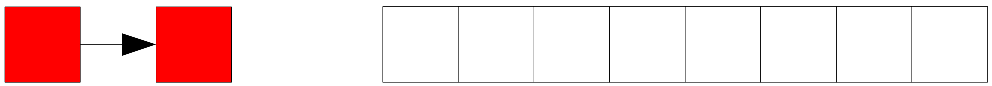
- 10 free chunks of size 100 in chunk_pool
- Low priority thread
  message of size 200 => need 3 chunks
- Mid priority thread
  message of size 500 => need 6 chunks
- High priority thread
  message of size 600 => need 7 chunks

Memory Deadlock (2)

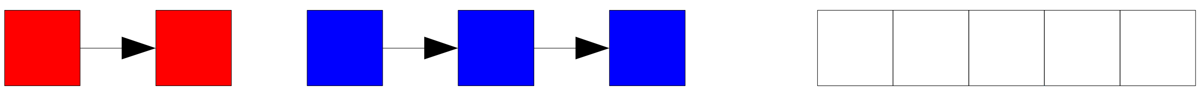


10 free chunks in chunk_pool

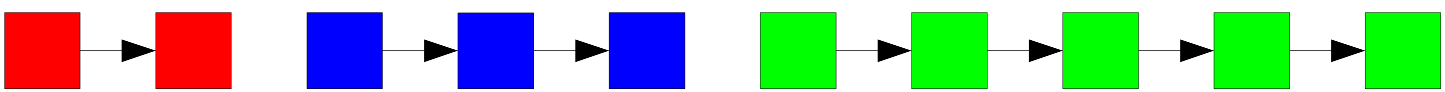
1. low priority thread invokes msg_alloc()



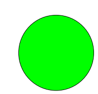
2. mid priority thread prempts low priority thread



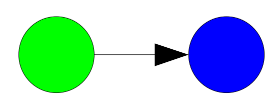
3. high priority thread prempts mid priority thread



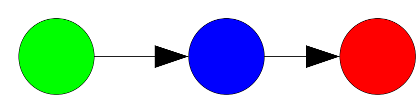
4. high priority thread waits in chunk_alloc()



5. mid priority thread waits in chunk_alloc()



6. low priority thread waits in chunk_alloc()



Memory Deadlock – IPC solution

- IPC protocols design enables transmission of “uncomplete” messages
  - postpone reclaiming of missing portion until message retrieval by destination application
- Messages allocated in a 1+N chunks method
  - First chunk allocated in blocking mode
  - Remaining chunks allocated in non-blocking mode

Memory Deadlock Solution (1)

```
extern void* chunk_alloc(bool wait_ok);
  msg_t*
msg_alloc(msg_size_t size)
{
  msg  = (msg_t*)chunk_alloc(TRUE);
  msg->size = 0;
  while (msg->size < size) {
    chk = chunk_alloc(FALSE);
    if (chk == 0) return msg;
    add_chunk_to_msg(msg,chk);
    size += CHUNK_SIZE;
  }
  msg->size = size;
  msg->proto_header = (char*)msg +
                      sizeof(msg_t);
  return msg;
}
```

Memory Deadlock Solution (2)

```
  void*
chunk_alloc(bool wait_ok)
{
  chk_fine_lock_get();
  while(chunk_pool_empty){
    if (! wait_ok) {
      chk_fine_lock_rel();
      return 0;
    }
    wait_list_add(w_ctx);
    chk_fine_lock_rel();
    wait(w_ctx);
    chk_fine_lock_get();
  }
  chk = chunk_pool_rm();
  chk_fine_lock_rel();
  return chk;
}
```

```
  void
chunk_free(void* chk)
{
  chk_fine_lock_get();
  chunk_pool_add(chk);
  if (wait_list_empty){
    chk_fine_lock_rel();
    return;
  }
  ctx = wait_list_rm();
  chk_fine_lock_rel();
  awake(ctx)
}
```

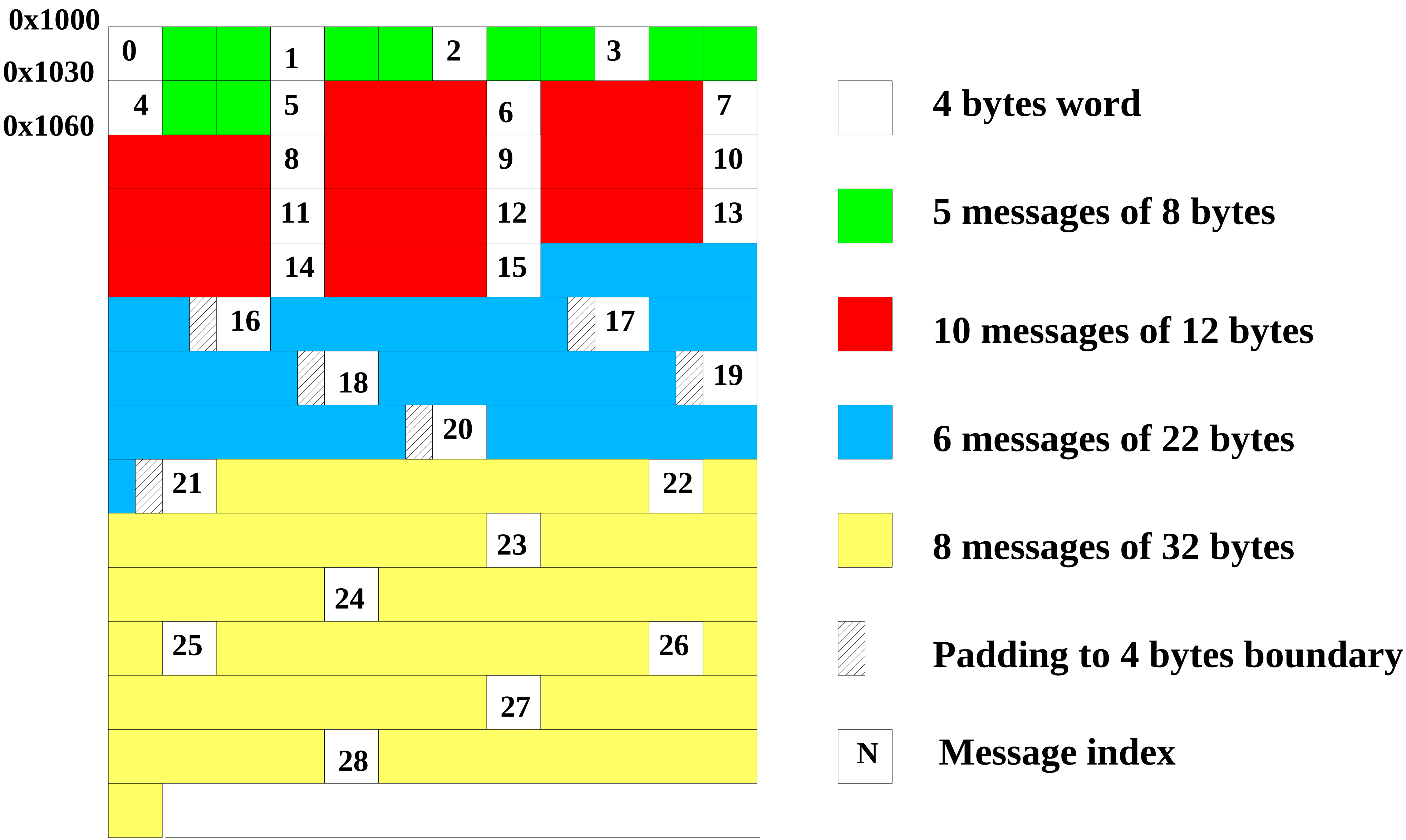
MIPC Communications

- Designed for Real-Time communications
  - zero-copy
  - guaranteed deterministic behavior
  - allows communications from interrupt handlers to user threads
- Local only
- Also used by customers as a memory allocator with real-time properties...

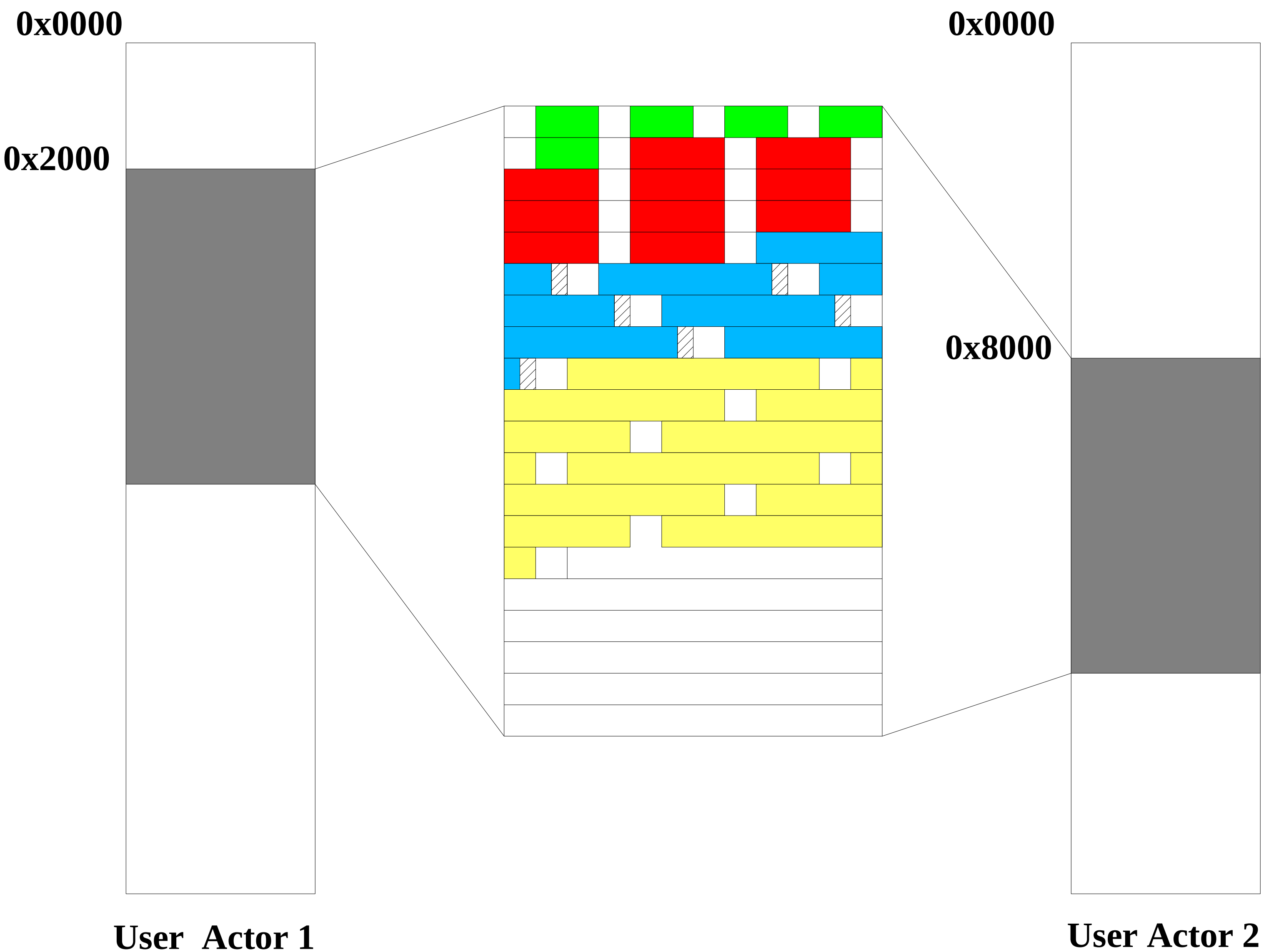
MIPC Communications (2)

- Design based on “Message Space”
  - Set of pools of messages
    - message size
    - number of messages
  - Set of “mailboxes”
- All resources pre-allocated at creation time
  - private to creating actor
  - shared among multiple actors
- Message memory mapped in space of actors
  => avoid “pointers” within messages...

Message Memory Layout



Message Memory Mapping



MIPC Communications

- Sender

```
msgAllocate(spId, poolId, size, delay, &msg);
   /* directly set message content's */
msgPut(spId, mbox1, msg, prio1);
```

- Receiver

```
msgGet(spId, mbox1, &msg, delay);
   /* directly access to message content's */
msgFree(spId, msg);
```

or

```
msgPut(spId, mbox2, msg, prio2);
```

MIPC Communications - Exemple

```
#define KBD_EVT     0
#define MOUSE_EVT 1
typedef struct {unsigned char evt_src;} in_evt_t;
typedef struct {
    in_evt_t evt;
    char        key_code;
} kbd_evt_t;
typedef struct {
    in_evt_t   evt;
    unsigned short x_pos; unsigned short y_pos;
    unsigned char buttons;
} ms_evt_t;
```

MIPC Exemple

```
#define INPUT_MBOX 0
#define MOUSE_PRIO 0
#define KBD_PRIO   1

#define KBD_EVT_POOL   0
#define MOUSE_EVT_POOL 1
KnMsgPool in_evt_pools[2] = {
  {sizeof(kbd_evt_t),   10},
  {sizeof(mouse_evt_t), 50},
}
spid = msgSpaceCreate(K_PRIVATEID, 1, 2, in_evt_pools);
```

MIPC Exemple

```
Mouse_Interrupt_Handler() {
  ms_evt_t* mse;


  diag = msgAllocate(spid, MOUSE_EVT_POOL,
                     sizeof(ms_evt_t), 0, &mse);
  mse->evt.evt_src = MOUSE_EVT;
  mse->x_pos    = hard_mouse_x_reg;
  mse->y_pos    = hard_mouse_y_reg;
  mse->buttons = hard_mouse_buttons;
  diag = msgPut(spid, INPUT_MBOX, mse, MOUSE_PRIO);
}
```

MIPC Exemple

```
Keyboard_Interrupt_Handler() {
  kbd_evt_t* kbde;


  diag = msgAllocate(spid, KBD_EVT_POOL,
                     sizeof(kbd_evt_t), 0, &kbde);
  kbde->evt. evt_src = KBD_EVT;
  kbde->key_code     = hard_kbd_key_reg;
  diag = msgPut(spid, INPUT_MBOX, kbde, KBD_PRIO);
}
```

MIPC Exemple

```
Input_Devices_Event_Process() {
  in_evt_t* evt;
  for (;;) {
    diag = msgGet(spid, INPUT_MBOX, K_NOTIMEOUT,
                  &evt, NULL);
    switch (evt->evt_src) {
      case KBD_EVT: kbd_evt_process(evt); break;
      case MOUSE_EVT: mouse_evt_process(evt); break;
      default: invalid_evt_process(evt); break;
    }
    msgFree(spid, evt);
  }
}
```

Memory Management

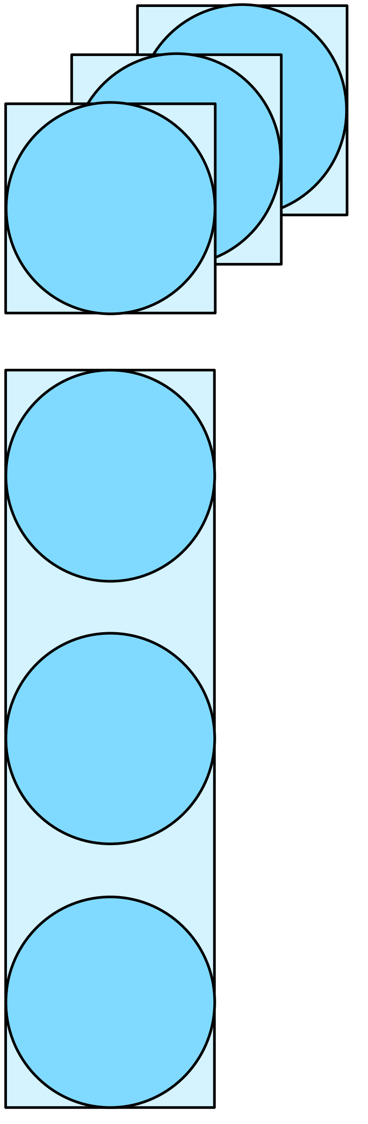
- Based on notion of **region**
  - address, size, access modes (R,W,X)
  - inheritance properties
- Configurable memory management support
  - **FL**at **M**emory (FLM)
  - **PR**otected **M**emory (PRM)
  - **V**irtual **M**emory (VM)
- Depends upon hardware support (MMU)
- Tradeoff performances/protection

Flat Memory (FLM)

- Single Supervisor Memory Space
- Unprotected
- Shared by microkernel and all actors
- Basic MMU support
  - to enable memory cache(s)
  - to setup non-cached memory regions (DMA)
  - 1-to-1 mapping
  - invalid address
    => unrecoverable system error

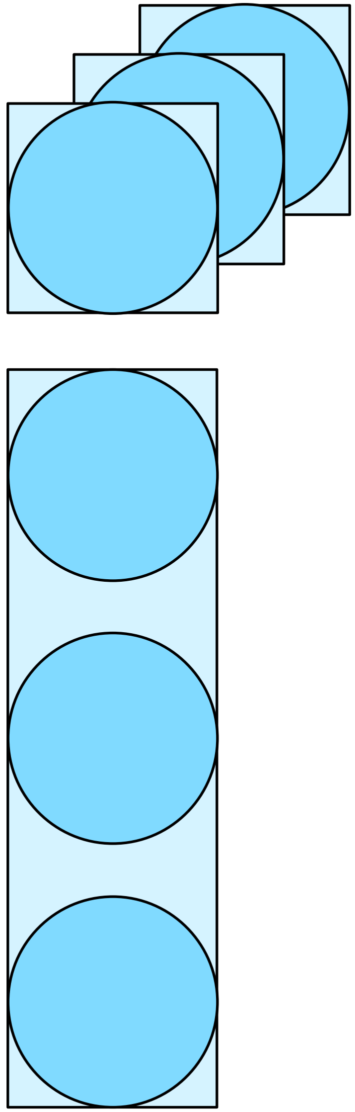
Protected Memory (PRM)

- Multiple user address spaces
- Mutually protected
- No lazy on-demand page allocation
- Invalid user-level address error
  - impacted to faulting thread
  - can be recovered by faulting application
- Slightly impacts performances
  - system calls
  - context switches



Virtual Memory (VM)

- Includes PRM features
- Dynamic lazy physical page allocation
  - fill-zero option ("bss" region)
- Copy-on-write optimization
  - page inheritance ("init data" region)
- Optional page swapping
  - external swapper
  - swap space accounted in available memory



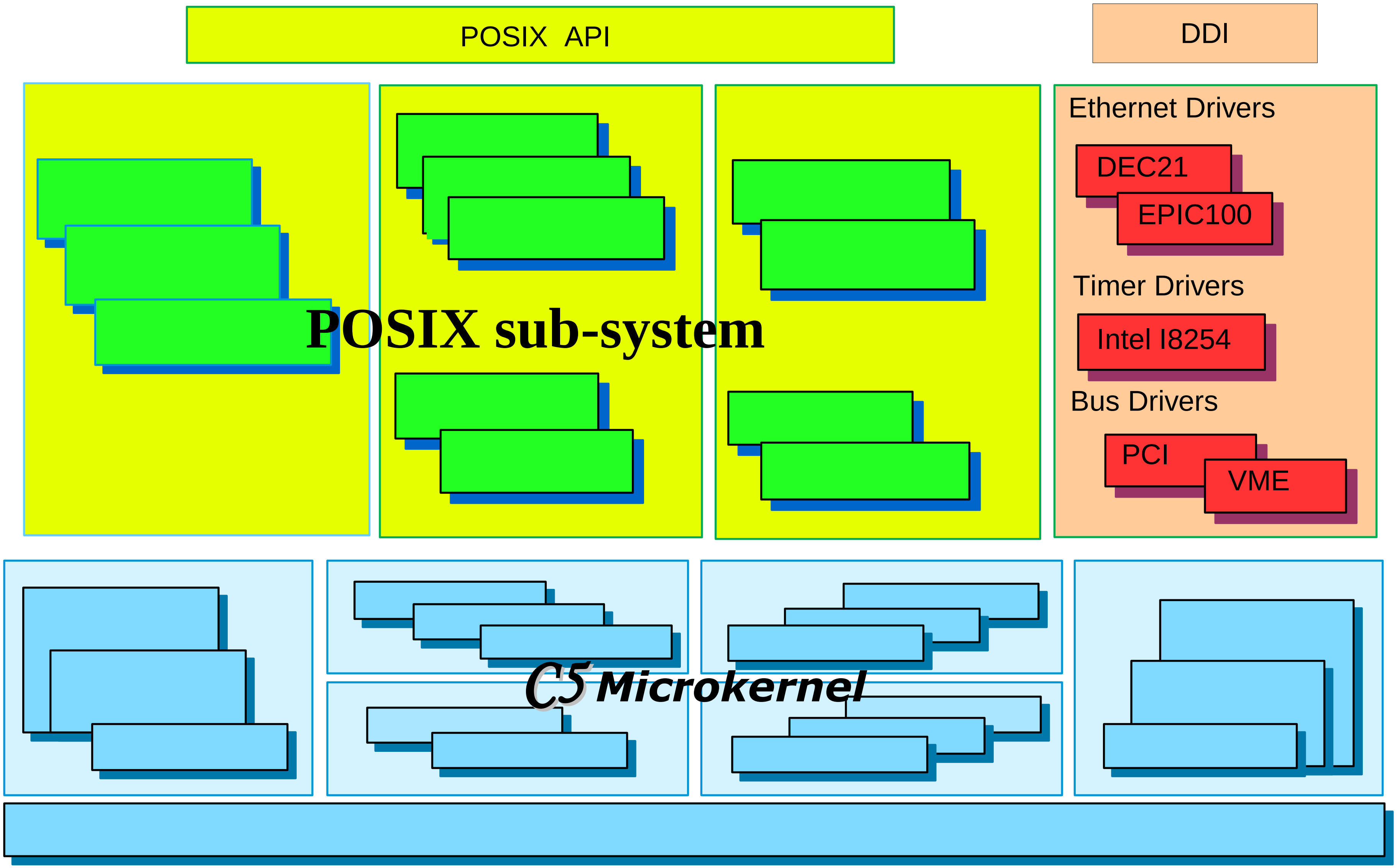
Memory mapped segment (VM)

- Map external objects in actor memory space
  - data files with R/W capabilities
  - executable binary files (read-only)
- Supports coherent file caching & mapping
- On-demand page loading
- Background "dirty" page flushing
- Interface with external mapper(s)
  - built over Chorus RPC

Overview

- C5 (ChorusOS®) Microkernel
- Device Driver Framework
- POSIX Personality
- Development Environment

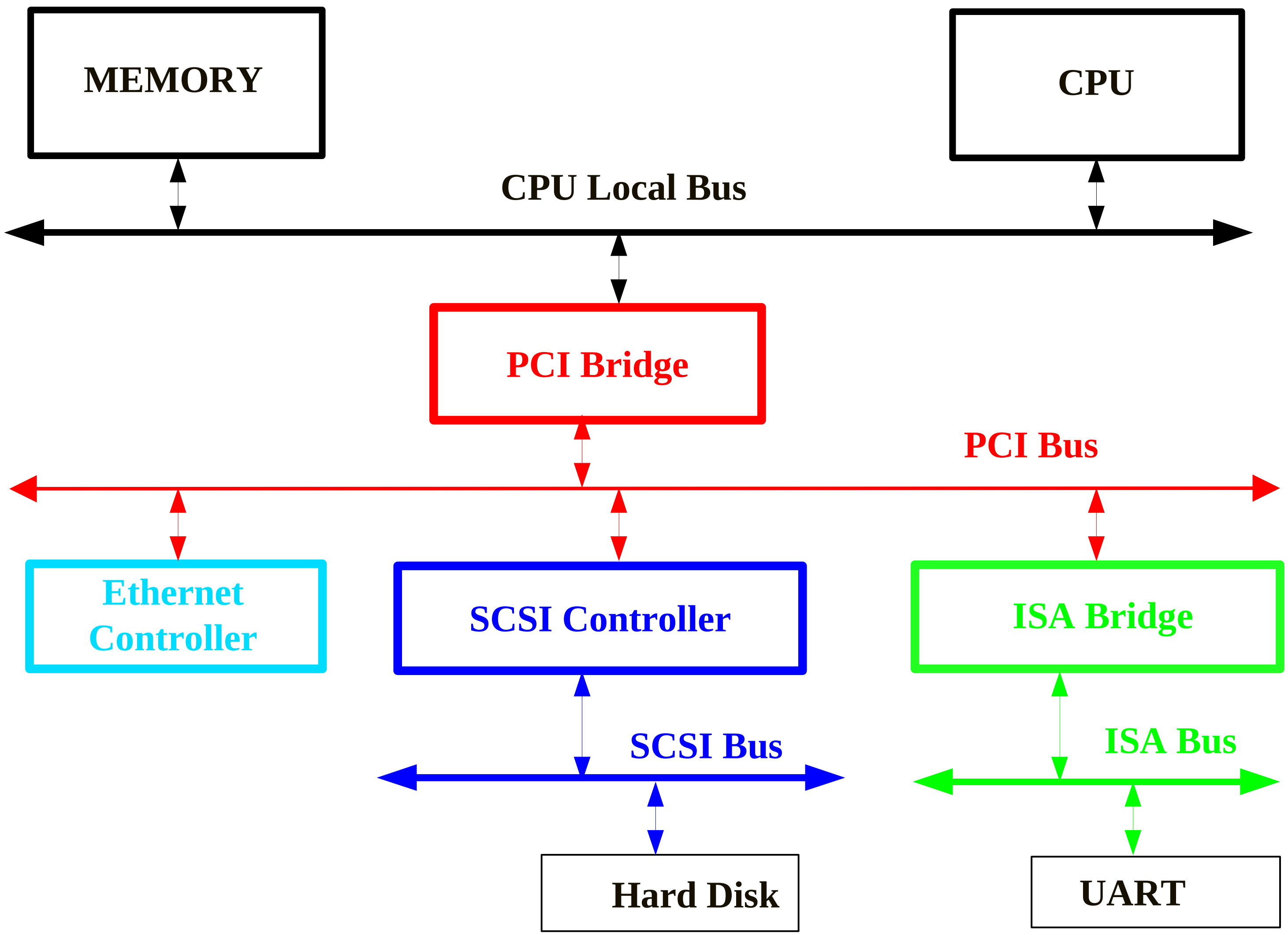
Device Drivers Architecture



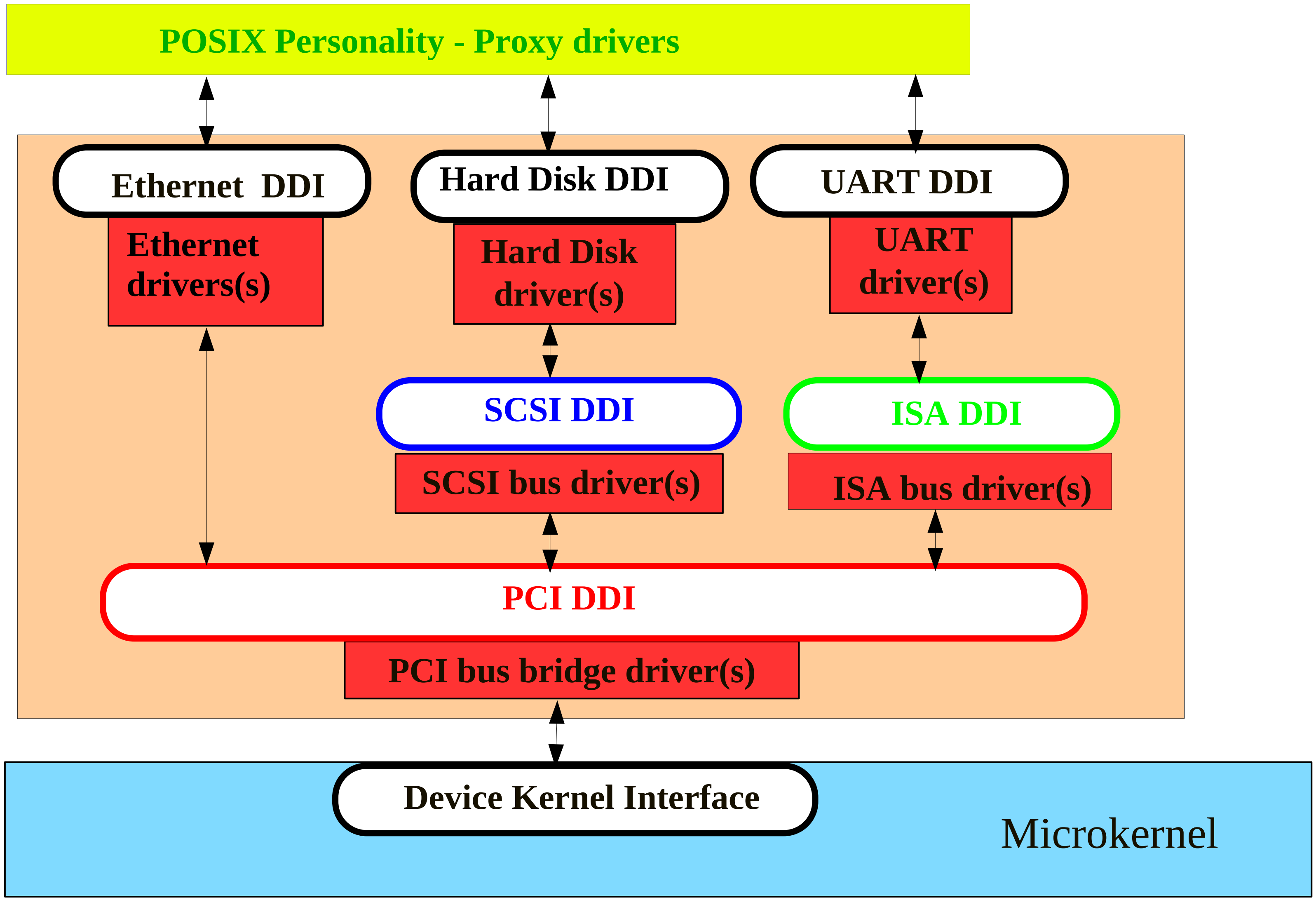
DKI/DDI Principles

- Develop CPU & platform independent drivers
  - I/O devices only I/O bus dependent
  - portable drivers
  - deliverable within binary packages
- Support of hot-plug and hot-swap devices
  - cPCI I/O boards
  - PCMCIA
- Extended driver framework
  - hardening support: driver resilient to hardware faults
  - fault injection: test driver behaviour

Physical Device Architecture



Device Drivers Layout



DDI/DKI Interfaces

- DKI (Driver Kernel Interface)
  - set of services provided by microkernel to drivers
  - generic DKI services
  - processor-specific DKI services
- DDI (Device Driver Interface)
  - interfaces exported by all layers of the device tree
  - typically one DDI per class of bus or device

Generic DKI Services

- Synchronization through the DKI thread
- Device Tree, Driver Registry, Device Registry
- General purpose memory allocation
- Timeout management
- Precise busy wait
- Special purpose physical memory allocation
- System event management
- Global interrupt masking

CPU-specific DKI Services

- Processor interrupts management
  - CPU interrupt vectors
  - CPU interrupt contexts
- Processor caches management
  - data cache flushing
- Processor specific I/O services
  - I/O instructions
  - load-and-swap specific instructions
- Processor specific fault handling
  - invalid physical address

Device Tree Structure

- Hierarchical representation of hardware arch.
  - root node = local CPU bus
  - nexus nodes = I/O bus bridges
  - leaf nodes = I/O devices
- Nodes designated by a pathname
  - ex: /raven/pci1011,9@e,0
- Set of properties associated to each node
  - property = (name, value)
  - size, layout & meaning of value are device specific
  - ex: ETHER_PROP_THROUGHPUT=100000000

Device Tree

- Device tree populated
  - at system init time
  - in a top-down fashion
- Static pre-defined set of nodes
  - local CPU bus
  - devices behind bus with no device probing
  - devices with initial properties
- Dynamic device discovery
  - bus with device probing capabilities (PCI)
  - hot-plug devices (PCMCIA, cPCI, USB)

Driver & Device Registry

- Driver Registry
  - set of “active” drivers
  - drivers register themselves when started
- Device Registry
  - set of “activated” devices (< device tree)
  - devices registered by drivers
  - device lookup, locking & releasing
  - “events” (shutdown) propagation

Bus Driver Interface

- Bus DDI = generic abstraction of I/O bus
- Common Bus Driver Interface
  - interrupt attachment, etc...
  - system and bus events propagation (shutdown, etc...)
- Specific DDI for every class of bus
  - PCI
  - ISA
- CPU local bus driver directly built upon DKI services
- Multiple clients (I/O devices, bus bridges)

Bus Driver Interface (2)

- Interrupt handling
  - handler attachment
  - source interrupt acknowledgment / demultiplexing
- I/O resource access
  - I/O registers mapping / load/store
- DMA support
  - bus address translation
  - memory alignment constraints
  - memory caching issues (bus snooping)

Device Driver Interface

- Device DDI = generic abstraction of I/O devices
- Specific DDI for every class of device
  - Ethernet
  - Watchdog
  - NVRAM
  - High Resolution Timer
- Client/Server model
  - usually imposes single client
  - driver client always a trusted system component
  - synchronization rules enforced by client

Ethernet DDI

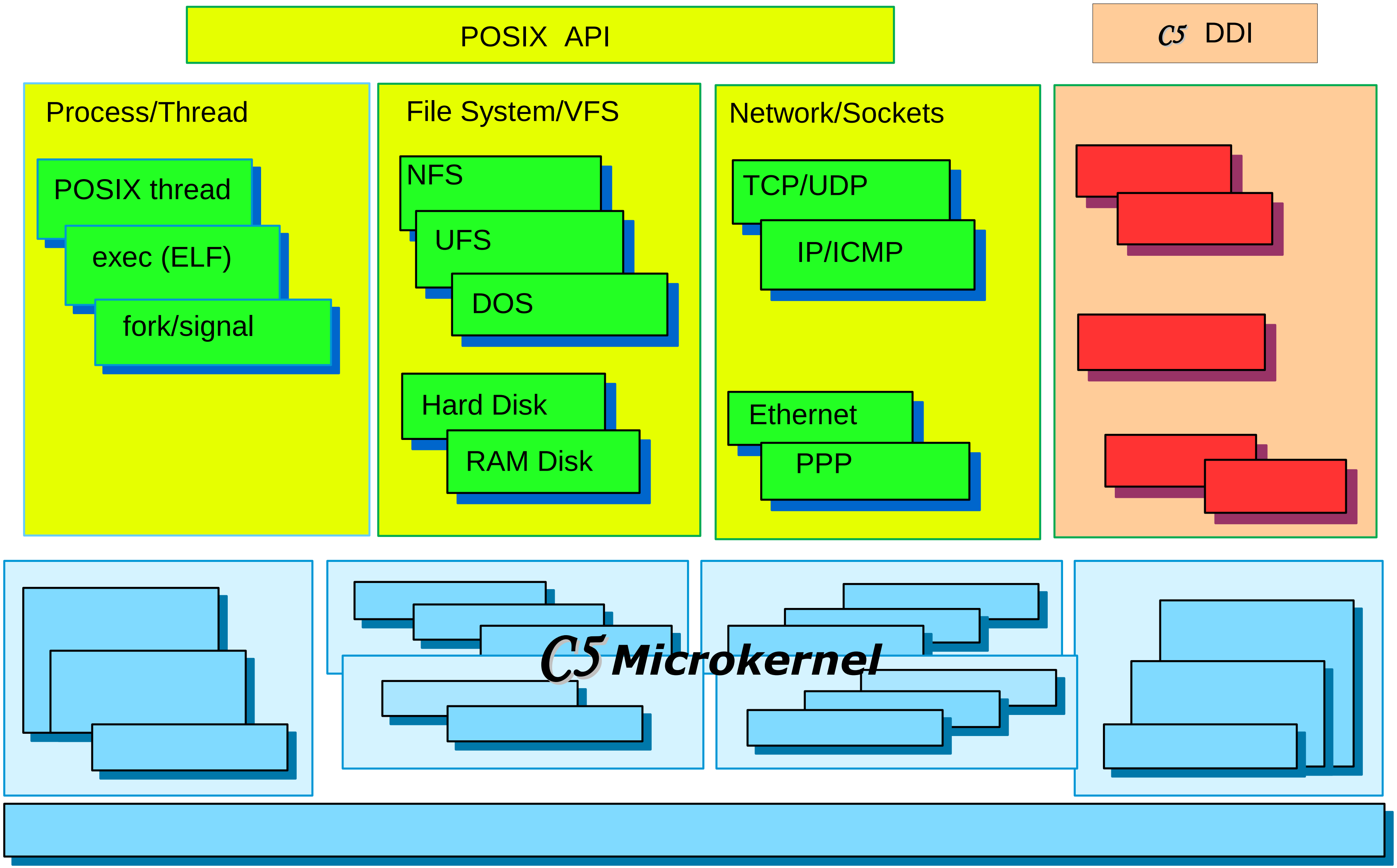
```
typedef struct {
    void (*transmitNotify)(void* cbId);
    void (*receiptNotify)(void* cbId);
    NetBuf* (*netBufAlloc)(void* cbId, uint32 size);
    void (*netBufFree)(void* cbId, NetBuf* bufList);
    void (*ioErrorNotify)(void* cbId, EtherIoError);
    void (*linkStateNotify)(void* cbId, EtherLinkState);
} EtherCallBack;
```

```
typedef struct {
    int  (*open)(void* dev, void* cbId, EtherCallBack*);
    void (*close)(void* dev);
    int  (*frameTransmit)(void* dev, NetFrame* outFr);
    int  (*frameReceive)(void* dev, NetFrame* inFr);
    void (*promiscuousEnable)(void* dev);
    void (*promiscuousDisable)(void* dev);
    void (*multicastSet)(void* dev, uint n, EtherMcast*);
} EtherDevOps;
```

Overview

- C5 (ChorusOS®) Microkernel
- Device Driver Framework
- POSIX Personality
- Development Environment

POSIX Personality



POSIX Personality

- Based on FreeBSD 4.1
- Easy migration/port of UNIX applications
- Single programming model & environment for C5 microkernel and POSIX sides
- Mixed usage of POSIX and C5 microkernel API's
- Simplify design of gateway applications
- Enable optimizations

POSIX Process

- Applications in user and supervisor space
- Few restrictions in supervisor space
  - fork()
  - POSIX signal handling
- ELF binary format
  - compressed binaries
  - Execution In Place (XIP)
  - dynamic libraries
  - shared libraries

POSIX Threads

- Full multi-threaded process
- FreeBSD kernel code adaptation
- POSIX thread = C5 thread + POSIX context
- Dynamic POSIX incarnation of C5 threads
- API & libraries adapted to multi-threading
  - errno
  - malloc/free
  - fprintf
  - etc.

POSIX Files

- File Systems
  - UFS & FFS (with 64bits disk and file size)
  - NFS v2, NFS v3
  - PROCFS
  - MSDOS FS (with long pathnames), FAT12, FAT16, FAT32
- Drivers
  - hard disk (IDE, SCSI)
  - Flash Memory
  - RAM

POSIX Networking

- TCP/UDP/IP v4 & v6
- IP multicast
- Dynamic creation of network interfaces
- Ethernet
  - dynamic insertion/removal of Ethernet boards
- PPP
- DHCP (client)
- DNS (client)

Embedded Administration

- Self-contained system administration
- Embedded command interpreter (**C_INIT**)
  - built-in commands
    - mkdev
    - rarp, ifconfig
    - route add/delete
    - mount, umount
  - "rshd" deamon by default
- Embedded initial configuration file
  - set of initialization commands
  - set of initialization parameters