OS Révolutionnaire — Concept & Pseudocode (Minimaliste)

> Document de concept et pseudocode destiné à guider la conception d’un système d'exploitation minimaliste, sécurisé et performant.

---

1. Résumé exécutif

Ce document présente un concept d'OS minimaliste, **sécurisé par capacités**, basé sur une **micro-kernel** ultra-léger, écrit majoritairement en **Rust** pour sécurité mémoire, et conçu pour être extensible via des services utilisateurs (drivers et services en espace utilisateur). L'objectif : obtenir un noyau petit, formellement vérifiable si nécessaire, avec une surface d'attaque réduite, et des performances compétitives grâce à des optimisations ciblées (Fast IPC, batching, évitement des copies mémoire).

2. Principes de conception

1. **Minimalité** : le noyau ne fournit que les mécanismes fondamentaux (gestion d'adresses, gestion de threads, IPC, ordonnancement basique, gestion des capacités). Toute politique vit en espace utilisateur.
2. **Moindre privilège** : chaque service reçoit uniquement les capacités nécessaires.
3. **Sécurité par conception** : capacités non-forgeables, séparation stricte, code système auditable.
4. **Composabilité** : services légers et remplaçables en espace utilisateur.
5. **Pragmatique performance** : mesures ciblées (zero-copy quand possible, batch IPC, lock-free structures) sans sacrifier la sécurité.
6. **Lisibilité & maintenabilité** : APIs claires, pseudocode humainement lisible.

3. Cas d'usage ciblés

- Systèmes embarqués et appareils IoT sûrs
- Cloud / unikernels pour services conteneurisés
- Stations de travail minimalistes et isolation forte
- Recherche et plateforme de prototypage pour kernels vérifiables

4. Architecture globale (vue haute)

- **Bootloader** → initialise la MMU, GDT/IDT (si x86), charge le microkernel.
- **Microkernel (Ring 0)** : gestion mémoire minimale, scheduler, IPC rapide, gestion de capacités, exceptions.
- **Managers utilisateur** (en espace utilisateur): Process Manager, Device Manager (drivers sandboxés), Filesystem service, Network service.
- **Runtime & libOS** : bibliothèques pour apps et services (par ex. tiny libc/rust core) qui utilisent les capacités.

5. Modèle de sécurité — capacités

- Les *capabilities* sont des handles immuables émis par le kernel. Elles encodent : l'objet ciblé, les droits (read, write, exec, map, ipc send/recv) et une version/nonce pour révoquer.
- Opérations sur capacités : `invoke`, `copy-limited`, `seal`, `revoke`.
- Les syscalls prennent un capability-id et une operation; kernel vérifie la validité et les droits.

6. Langage & outils recommandés

- **Rust** pour la majeure partie du code (sécurité mémoire + `unsafe` limité au minimum).
- Composants formellement vérifiables (ex. seL4-style) pour la couche la plus critique si besoin.

7. API minimale (extraits)

- `cap_create(obj_type, rights) -> cap_id`
- `cap_invoke(cap_id, op, params...) -> result`
- `thread_create(entry_point, stack, caps[]) -> tid`
- `ipc_send(dest_cap, msg_ptr, len)` / `ipc_recv(src_cap, buf, len)`
- `map_memory(cap_mem, vaddr, flags)`
- `yield()` / `sleep(ms)`

8. Policy vs Mechanism

Le noyau implémente **uniquement** des mécanismes. Par exemple, le scheduler fournit `yield` / `set_priority`, mais la stratégie de haute performance (affinity, work-stealing) se fait en espace utilisateur via un *scheduler service* si nécessaire.

9. Format de drivers

- Drivers compilés en modules utilisateur avec sandbox (chroot/logical isolation + capability set).
- Communication via fast IPC avec pages partagées mappées uniquement en lecture/écriture selon capabilities.

Pseudocode détaillé

> Style : semi-formel, expliqué ligne par ligne. Les fonctions critiques sont écrites de façon à pouvoir être traduites en Rust/C.

A. Structures de base

```
// Identifiants simples (atomiques)
struct Cap { id: u64, type: CapType, rights: Rights, version: u32 }
struct Thread { tid: u64, state: ThreadState, regs: Regs, caps: List<Cap> }
```

```
struct PageRef { phys_addr: u64, refcount: atomic<int> }

// Kernel global (minimal)
kernel {
  cap_table: map<CapID, KernelCapEntry>
  thread_table: map<Tid, Thread>
  ready_queue: priority-queue of Tid
  ipc_queues: map<CapID, Queue<Message>>
  phys_mem_table: map<PhysAddr, PageRef>
}
```

B. Boot sequence (pseudocode)

```
boot() {
  init_cpu_early()           // MMU off -> set identity mappings for kernel
  setup_exception_vectors()
  init_phys_mem_manager()    // build phys_mem_table
  init_capability_root()     // root capability for boot services
  start_scheduler()          // spawn idle thread and scheduler loop
  start_init_process()       // user-space init that starts managers
}
```

C. Capability manager (core semantics)

```
cap_create(obj_type, rights) -> cap_id {
  cap = new Cap(next_id++, obj_type, rights, version=0)
  kernel.cap_table[cap.id] = KernelCapEntry(obj=obj_type.default_constructor(),
cap)
  return cap.id
}

cap_invoke(caller_tid, cap_id, op, args...) -> result {
  entry = kernel.cap_table[cap_id]
  if entry == null: return ERR_BAD_CAP
  if not check_rights(entry.cap, op): return ERR_DENIED
  if entry.locked: return ERR_BUSY
  // for certain ops, perform kernel mediated action
  switch(entry.cap.type) {
    case THREAD_SERVICE: return thread_service_handle(op, args...)
    case MEMORY_OBJECT: return memory_object_handle(op, args...)
    case IPC_ENDPOINT: return ipc_handle(op, args...)
    default: return ERR_UNSUPPORTED
  }
}
```

```
revoke(cap_id) {
  entry = kernel.cap_table[cap_id]
  entry.cap.version += 1
  entry.invalidated = true
}
```

Explication : la `version` permet d'empêcher l'usage après révocation (nonce vérifier lors de cap lookups).

D. Fast IPC (zero-copy where possible)

```
// Basic message structure
struct Message { sender_tid, payload_ptr, payload_len, caps[] }

ipc_send(sender_tid, dest_cap, payload_ptr, payload_len, caps_to_transfer[]) {
  // 1) validate dest_cap is IPC_ENDPOINT and caller has send rights
  // 2) if payload small -> copy into kernel mailbox buffer
  // 3) if payload large and both sides agreed on shared page -> install shared
page mapping and send page-ref
  // 4) transfer any capabilities by moving entries in cap_table (or copy-
limited semantics)
  enqueue(kernel.ipc_queues[dest_cap], Message(...))
  wake_thread_if_blocked(dest_cap)
  return OK
}

ipc_recv(recv_tid, my_ipc_cap, buf, buf_len) -> (bytes_received, sender_tid) {
  msg = dequeue_or_block(kernel.ipc_queues[my_ipc_cap])
  if msg.payload_was_shared_page:
    map_shared_page_into_recvers_address_space(msg.page_ref)
    return (PAGE_SIZE, msg.sender_tid)
  else:
    copy(msg.payload_ptr, buf, min(buf_len, msg.payload_len))
    return (copied, msg.sender_tid)
}
```

Notes : privilégier mappage de pages pour très gros transferts pour éviter copies.

E. Scheduler (minimal, extensible)

```
// Kernel provides a minimal preemptive timeslice scheduler with priorities
scheduler_loop() {
  while(true) {
    next_tid = pick_from_ready_queue()
```

```
    if next_tid == null: run_idle()
    context_switch_to(next_tid)
  }
}

pick_from_ready_queue() {
  // simple: highest priority, round-robin within priority
  return ready_queue.pop()
}

context_switch_to(tid) {
  prev = current_thread
  save_context(prev)
  load_context(kernel.thread_table[tid])
  current_thread = tid
}
```

Extensibilité : l'`init` utilisateur peut run un *scheduler service* (via a dedicated capability) pour policies avancées ; kernel exposera hooks pour donation of CPU, preemption control, and accounting.

F. Device & driver model (user-space)

```
// Device manager creates device capabilities that wrap the hardware resource
device_open(driver_name) -> dev_cap {
  // driver runs as user process with only the device capabilities it needs
  // device manager maps hw regs into driver sandbox with caps
}

// Driver communicates with apps via capability endpoints and shared memory
pages
```

G. Memory manager (minimal semantics)

```
alloc_page() -> cap_mem {
  phys = phys_mem_alloc()
  page = PageRef(phys, refcount=1)
  kernel.phys_mem_table[phys] = page
  return cap_create(MEMORY_OBJECT, rights=MAP | READ | WRITE)
}

map_memory(cap_mem, vaddr, flags) {
  entry = lookup_cap(cap_mem)
  if not entry.rights.contains(MAP): return ERR_DENIED
  mmu_map(entry.obj.phys_addr, vaddr, flags)
```

```
  return OK
}
```

Reference counting + explicit release to limit kernel allocations.

H. Process lifecycle (create / exec / kill)

```
process_create(image_cap, args[], caps_for_process[]) -> pid {
  // 1) allocate address space object
  // 2) create main thread with stack
  // 3) install initial capabilities into the process' capability table
  // 4) schedule thread
}

process_exec(pid, new_image_cap) {
  // replace address space; revoke old memory caps; new mapping
}

process_kill(pid) {
  // revoke all caps owned by process (version bump), release pages
  // notify waiting parents via IPC
}
```

I. Simple example: user-space file read sequence (flow)

1. App holds `fs_client_cap` for filesystem service.
2. App builds `READ` request and invokes `cap_invoke(fs_client_cap, READ, path_ptr, len)`.
3. Filesystem service receives IPC, uses its storage device cap to fetch blocks, maps pages into its address space, returns capability to mapped pages to app (or copies small payloads).

---

10. Optimisations et bonnes pratiques

- **Batcher l'IPC** : grouper petits messages pour amortir overhead.
- **Avoid kernel allocations** : utiliser objets préalloués et pools.
- **Zero-copy page passing** : mappage de pages partagées avec droits restreints.
- **Lock-free queues** pour mailbox IPC quand possible.
- **Profiling hooks** en espace utilisateur (éviter instrumentation invasive dans noyau).

11. Roadmap d'implémentation (phases)

1. Prototype minimal : boot → simple scheduler → cap_create/map → IPC basic.
2. User-space init + simple fs service + simple net echo driver.

3. Performance tuning : fast IPC, page-passing.
4. Security hardening & testing (fuzzing, formal proofs sur le microkernel si possible).
5. Ecosystème : libOS, package toolchain, tooling de debug.

12. Annexes — Checklist pour design

- [ ] Définir représentation binaire des capabilities
- [ ] Définir ABI d'IPC (header, caps array) pour compatibilité.
- [ ] Tests de performance & misuse cases (revocation racing)
- [ ] Plan de déploiement (unikernel/cloud/embedded)

Fin du document. Merci — si tu veux, je peux : - convertir le pseudocode en Rust-like skeletons, - générer un diagramme d’architecture, - produire des exemples d’API plus détaillés pour drivers ou un scheduler avancé.