Multiprocessing

Advanced Operating Systems

Overview

- Process creation
  - fork
  - exec
- Scheduling
  - Time management
  - Scheduling strategies
- IPC
  - Shared memory
  - Semaphores
  - Message queues

Fork

`pid_t fork(void);`

- Creates a new child process by duplicating the calling process
- Implemented on top of the `clone()` system call on Linux

kernel/fork.c:

```
/*
 * Fork is rather simple, but the memory
 * management can be a bitch. See 'copy_page_range()'
 */
```

Fork: The Simple Part

- Duplicate task (`dup_task_struct`):
  - Copy most information, with exceptions, e.g. PID
- Allocate and initialize new kernel stack:
  - Setup `thread_info`
  - Copy trap frame (`pt_regs`) and update (e.g., %rax)
- And more:

```
retval = copy_files(clone_flags, p);
retval = copy_fs(clone_flags, p);
retval = copy_sighand(clone_flags, p);
retval = copy_signal(clone_flags, p);
retval = copy_mm(clone_flags, p);
```

Fork: The copy_mm Part

- Duplicate mm descriptor (`dup_mm`):
  - Copy over basic information
  - Initialize empty address space (new page directory)
- Duplicate address space (`dup_mmap`):
  - Copy over VMA information
  - Copy page tables (`copy_page_range`)
  - Fixup page table entries
    - MAP_SHARED VMAs: Share page frames
    - MAP_PRIVATE (R) VMAs: Share page frames
    - MAP_PRIVATE (R/W) VMAs: COW page frames

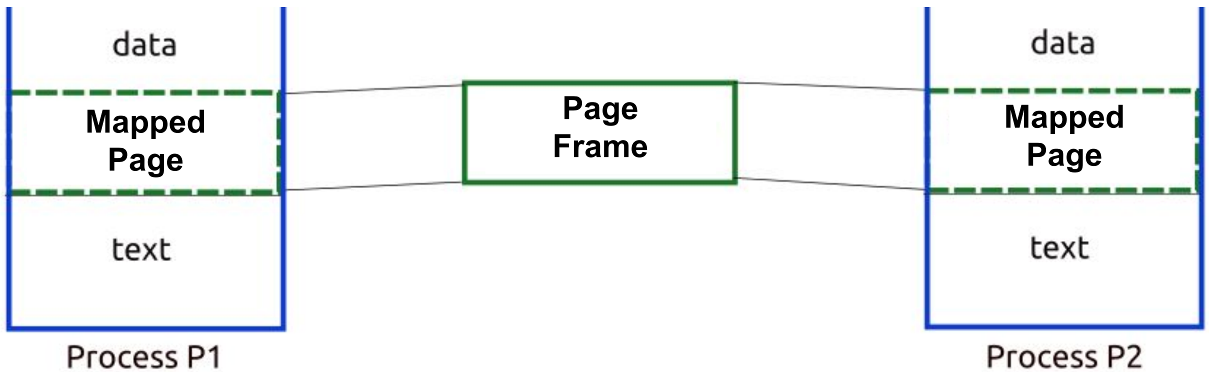
Fork: The copy_mm Part

```
/*
 * If it's a COW mapping, write protect it both
 * in the parent and the child
 */
if (is_cow_mapping(vm_flags)) {
        ptep_set_wrprotect(src_mm, addr, src_pte);
        pte = pte_wrprotect(pte);
}

/*
 * If it's a shared mapping, mark it clean in
 * the child
 */
if (vm_flags & VM_SHARED)
        pte = pte_mkclean(pte);

page = vm_normal_page(vma, addr, pte);
get_page(page);
set_pte_at(dst_mm, addr, dst_pte, pte);
```

Fork: The copy_mm Part



- Each common page frame: 2 VMAs, 2 PTEs
- Shmem: perm(VMA) = perm(PTE)
- COW: perm(VMA) = R/W, perm(PTE) = R
  - Page fault handler recognizes COW (`do_wp_page`)
  - Duplicates page frame and remaps a new private copy into the faulting PTE on demand

Exec

`int exec*(const char *filename, ...);`

- Executes the program pointed by filename
- Implementation:
  - Input and permission checking
  - Load binary headers in memory (`prepare_binprm`)
  - Find the binary format (`search_binary_handler`)
  - Flush old resources:
    - Reinitialize `task_struct` and (empty) `mm`
    - Flush VMAs, page tables, page frames
  - Load the binary (`load_binary`)

Exec

- Load binary (binary-format specific):
  - Parse headers and sections
  - Create corresponding VMAs (Data, Text, Stack, etc.)
  - Update %rsp and %rip in trap frame
    - %rsp = top of the user stack
    - %rip = program entry point for statically linked binaries
    - %rip = dynamic linker’s entry point otherwise
- Page tables initially empty
  - Even binary file pages (e.g., text) demand paged
  - Page fault handler maps them from page cache using COW-based strategy for safe sharing

Copy-On-Write: Applications

- MAP_PRIVATE file pages
  - Deduplicates binary pages for unrelated processes
  - Many (e.g., text pages) never COWed
- MAP_PRIVATE anon forked pages
  - Deduplicates pages within process hierarchy
  - Many (e.g., fork+exec) never COWed
- MAP_PRIVATE anon zero pages
  - Deduplicates zero pages for unrelated processes
  - At first read PF, map single read-only zero page
  - COW at first write PF

Scheduling

- We can now create multiple processes, how do we schedule them on a given CPU?
- The easy version (OpenLSD):
  - Tell the hardware to raise periodic timer interrupts
  - (Timer) interrupt handler invokes simple scheduler
- Simple scheduler:
  - Maintains a scheduling queue (FIFO)
  - Enqueues interrupted process at tail
  - Dequeues process at head and runs it on CPU
- This is a very simple preemptive round-robin scheduler, with no priorities, no fairness, etc.

Multiprocessing in OpenLSD

Lab 5: All together now (multiprocessing)

Core:

- Support multiprocessing with simple round-robin scheduling
- Support `fork` with Copy-On-Write (COW)

Bonuses:

- exec, more COW, IPC, timers

Time Management

- Hardware offers clock and timer circuits
- Clock circuits:
  - Expose counters incremented at given frequency
  - Can be used to keep track of current time of day
  - Can be used for precise time measurements
- Timer circuits:
  - Issue periodic interrupts at given frequency
  - Can be used for scheduling
  - Can be used to implement kernel and user timers

Time Management: Linux Clock Sources

- Abstraction of a clock circuit (`clocksource`)

  ```
  $ cat
  /sys/devices/system/clocksource/clocksource0/
  available_clocksource
  tsc hpet acpi_pm
  ```

- Clock sources (with different precision):
  - tsc: Timestamp counter (default)
  - hpet: high-precision event timer
  - acpi_pm: ACPI power management timer

Time Management: Real Time Clock (RTC)

- Special “source” (not a `clocksource`)
  - Persistent (battery-powered)
  - Low-precision (seconds)
- Used to get the current date and time:

```
$ cat /sys/class/rtc/rtc0/date
2016-09-26
$ cat /sys/class/rtc/rtc0/time
15:03:00
```

Time Management: Linux Clock Event Devices

- Abstraction of a timer circuit (`clock_event_device`)

```
$ cat /proc/timer_list | grep ^Clock | sort -u
Clock Event Device: hpet
Clock Event Device: lapic #local APIC
```

- Clock event device programmed to issue interrupts at `CONFIG_HZ` frequency, e.g.:
  - `CONFIG_HZ=250`, timer interrupt (or *tick*) every 4ms
- Both user and kernel preemption are possible
  - OpenLSD: non-preemptive kernel

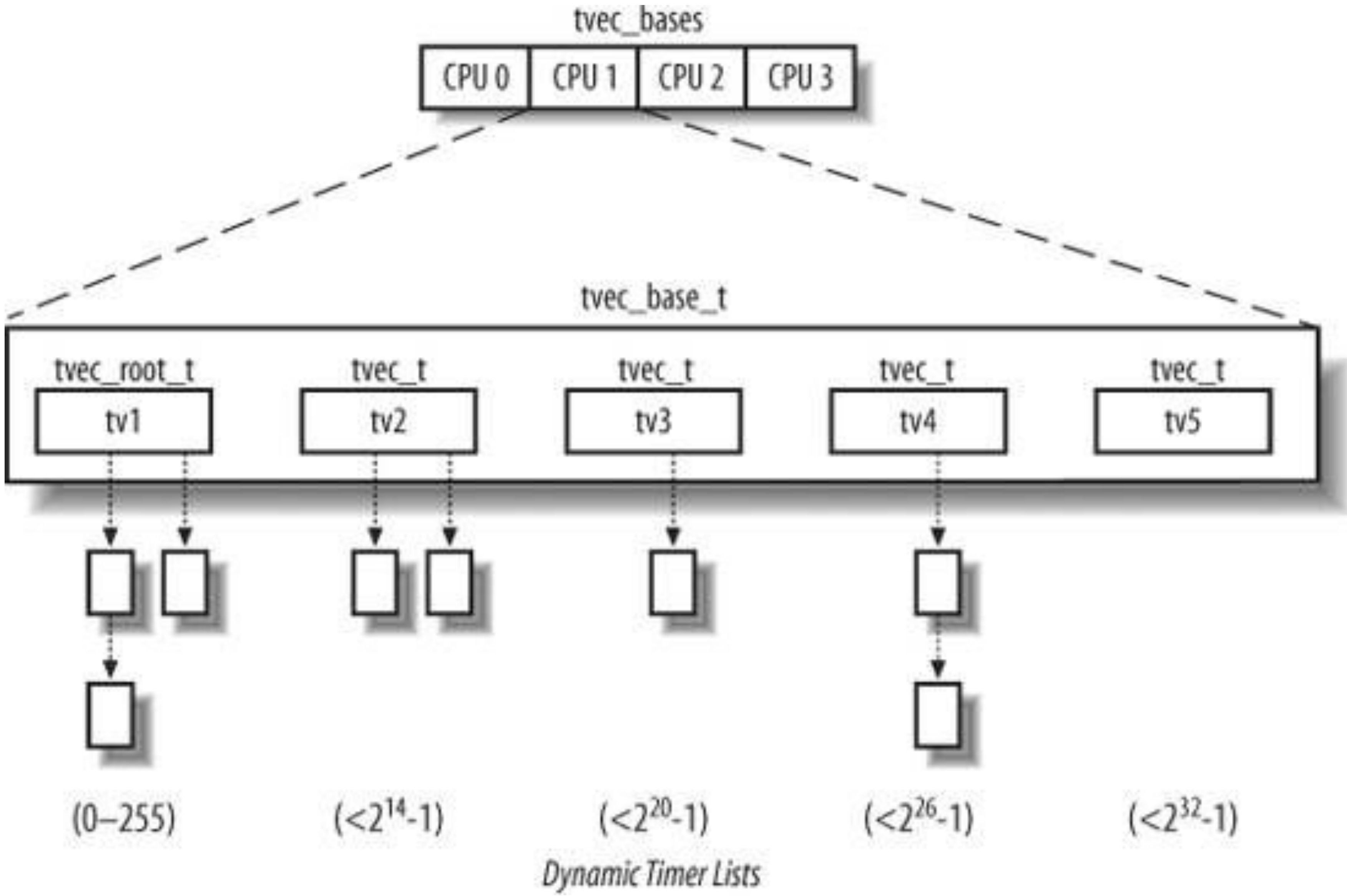
Time Management: Linux Timer Interrupts

- At each tick (roughly):

```
jiffies_64++; // update ticks since startup
update_wall_time(); // update current date/time
update_process_times(); // accounting
profile_tick(); // profiling

while (time_after_eq(jiffies, base->clk))
  expire_timers(base, ...); // expired timers

schedule(); // invoke the scheduler
```



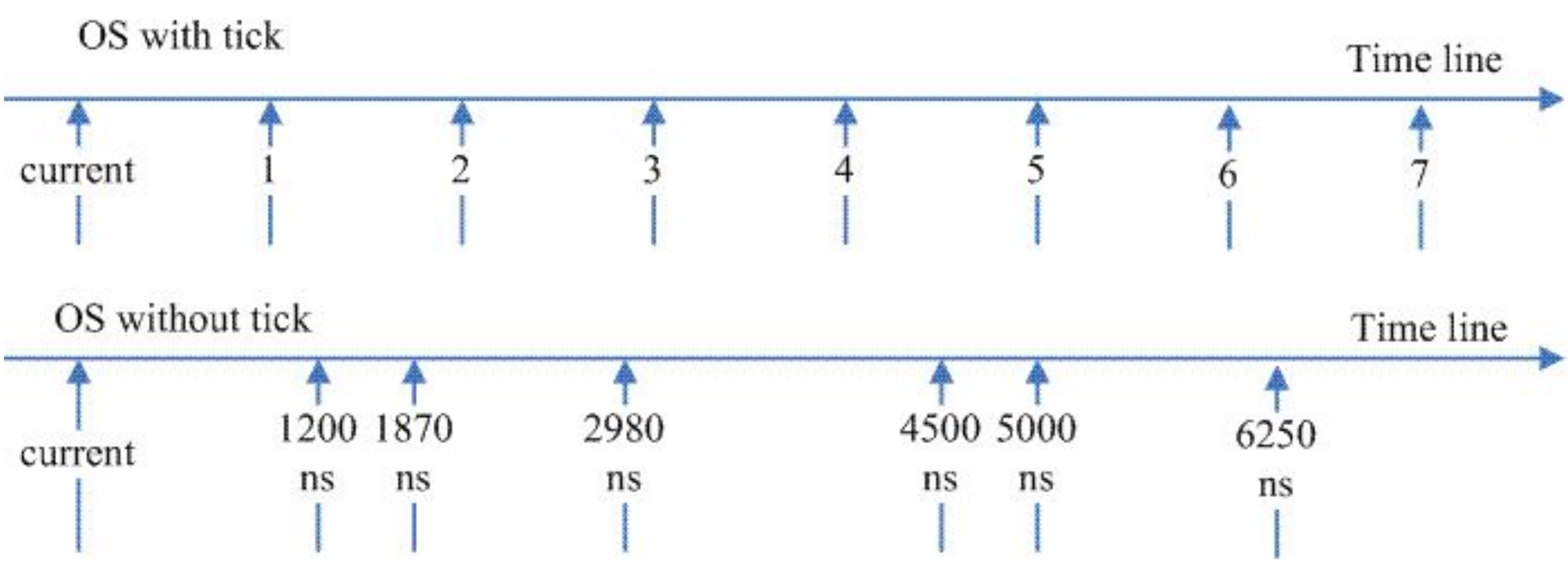
Linux Software Timers

Scheduling: Per-task Building Blocks

- State: Running, Runnable, Sleeping
- Quantum or time slice
  - Max number of jiffies a task can run on CPU
  - Initialized with task-specific formula (e.g., 100ms)
  - Decremented at every tick, task is done at 0
  - Sufficient to ensure fairness for CPU-intensive tasks
- Priority
  - Initialized with static (predetermined) priority
  - Possibly adjusted periodically
- Scheduling policy
  - NORMAL, BATCH, IDLE → Completely fair scheduler
  - FIFO, RR → Real-time scheduler

Scheduling: Tickless Kernel

- Can we go tickless (+responsive, -overhead)?
  - Create a new sw timer for “end of quantum” event
  - Reprogram hardware to tick at next timer to expire
- NO_HZ_IDLE|FULL: Tickless when idle|1 task

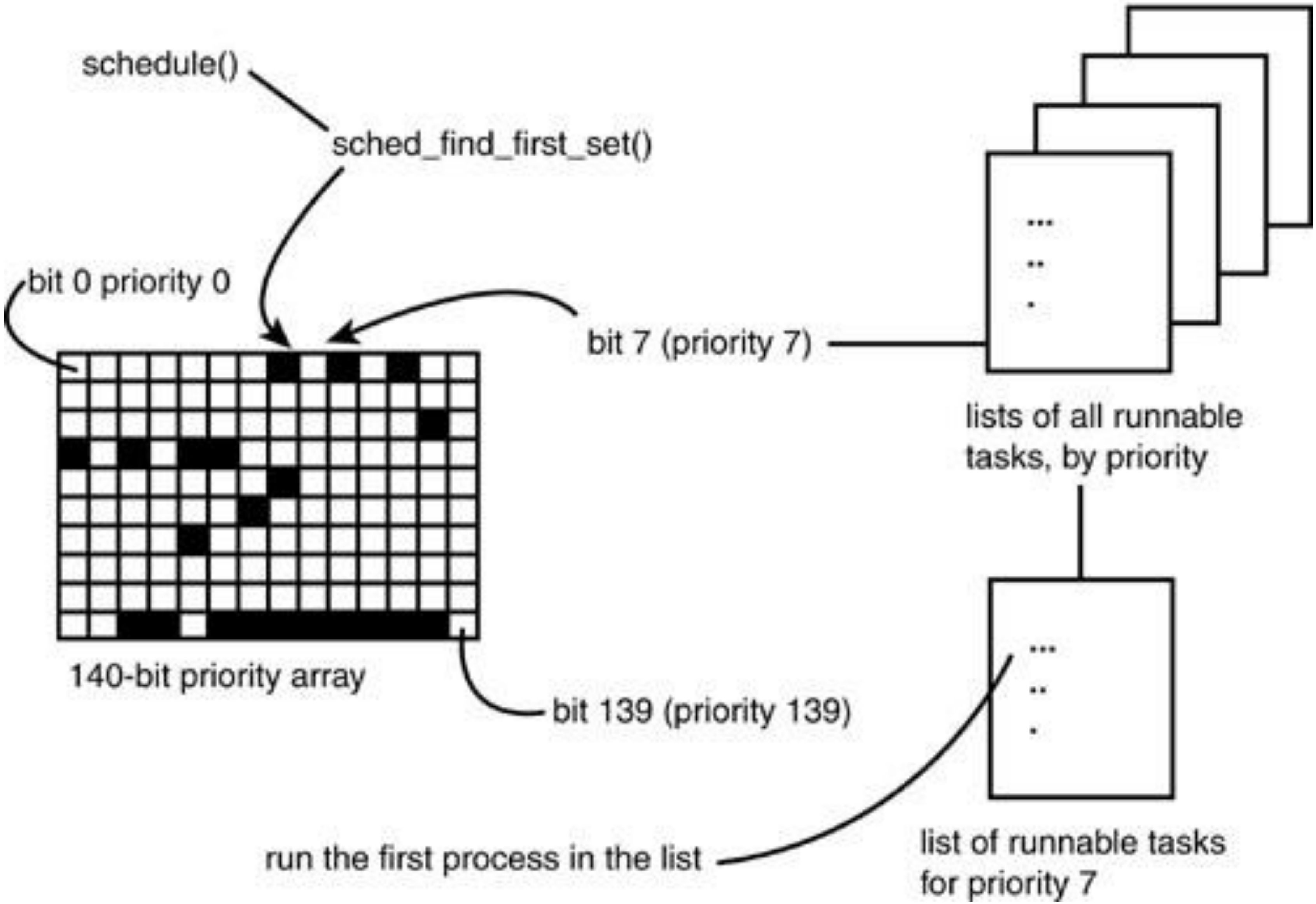


Scheduling: Linux O(1) Scheduler

- Preemptive round-robin priority scheduler
- Once default, now close to RT scheduler
- Maintains N run queues (1 per priority level)
- Scheduling strategy:
  - Find the highest-priority queue with runnable task
  - Find the first task on that queue and dequeue it
  - Calculate its time slice size based on priority and run it
  - When its time's up, enqueue it, and repeat
- Improving fairness:
  - Priorities are adjusted based on *sleep time*
  - Bonuses for I/O- vs. CPU-bound processes

Scheduling:
Linux O(1) Scheduler

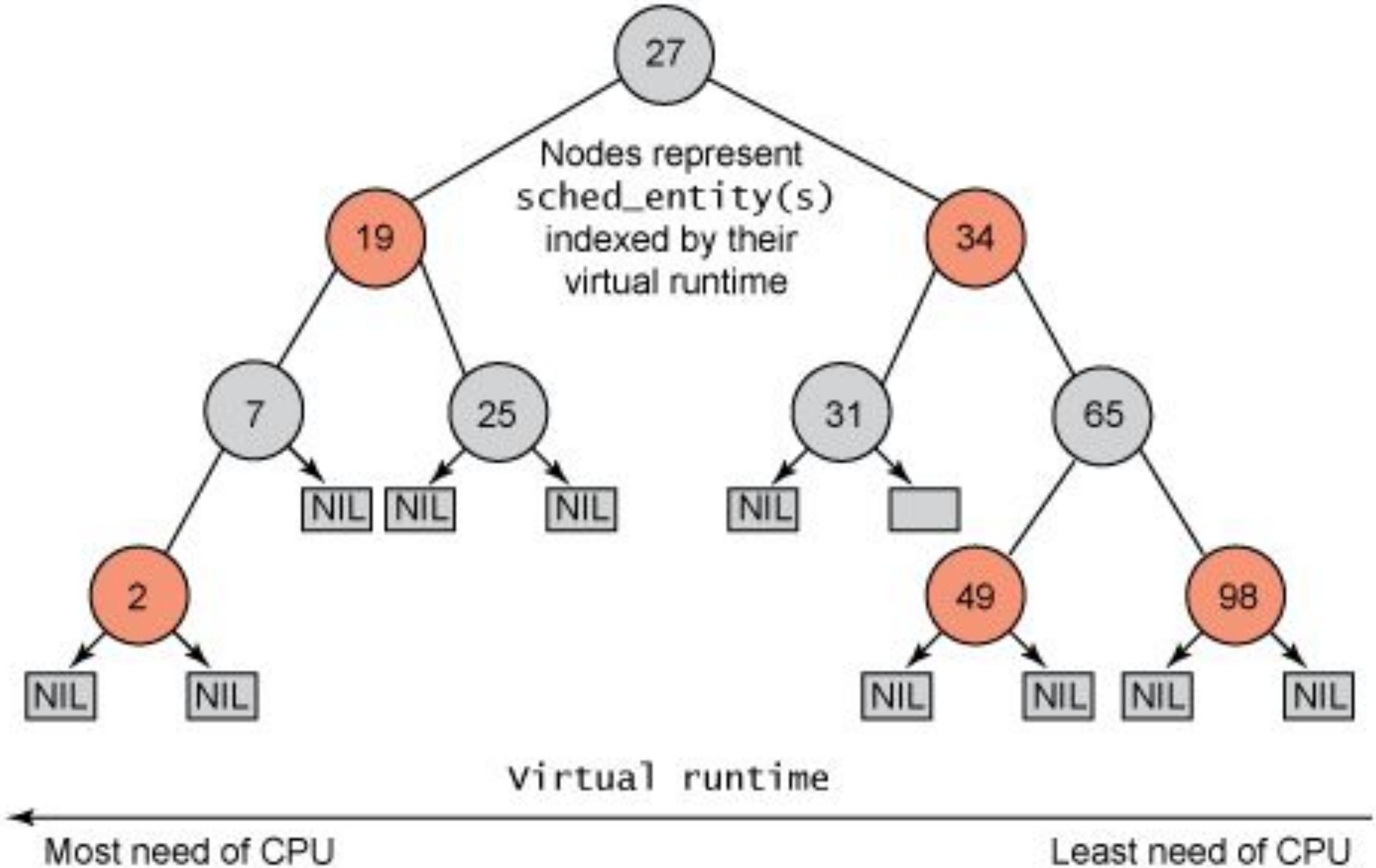
- Why O(1) Scheduler?
  - All the operations are O(1)
  - No loop across tasks like older O(n) scheduler
- Tasks only touched at dequeue/enqueue time
  - Recalculate priorities, quanta, bonuses
- Finding the right run queue is also O(1):
  - Bitmap indicating which queues have runnable tasks
  - Find the first bit that is set (first 1-bit instruction)
  - Time depends on the number of priority levels, not on the number of tasks (*scalability*)



Linux O(1) Scheduler

Scheduling: Linux CFS Scheduler

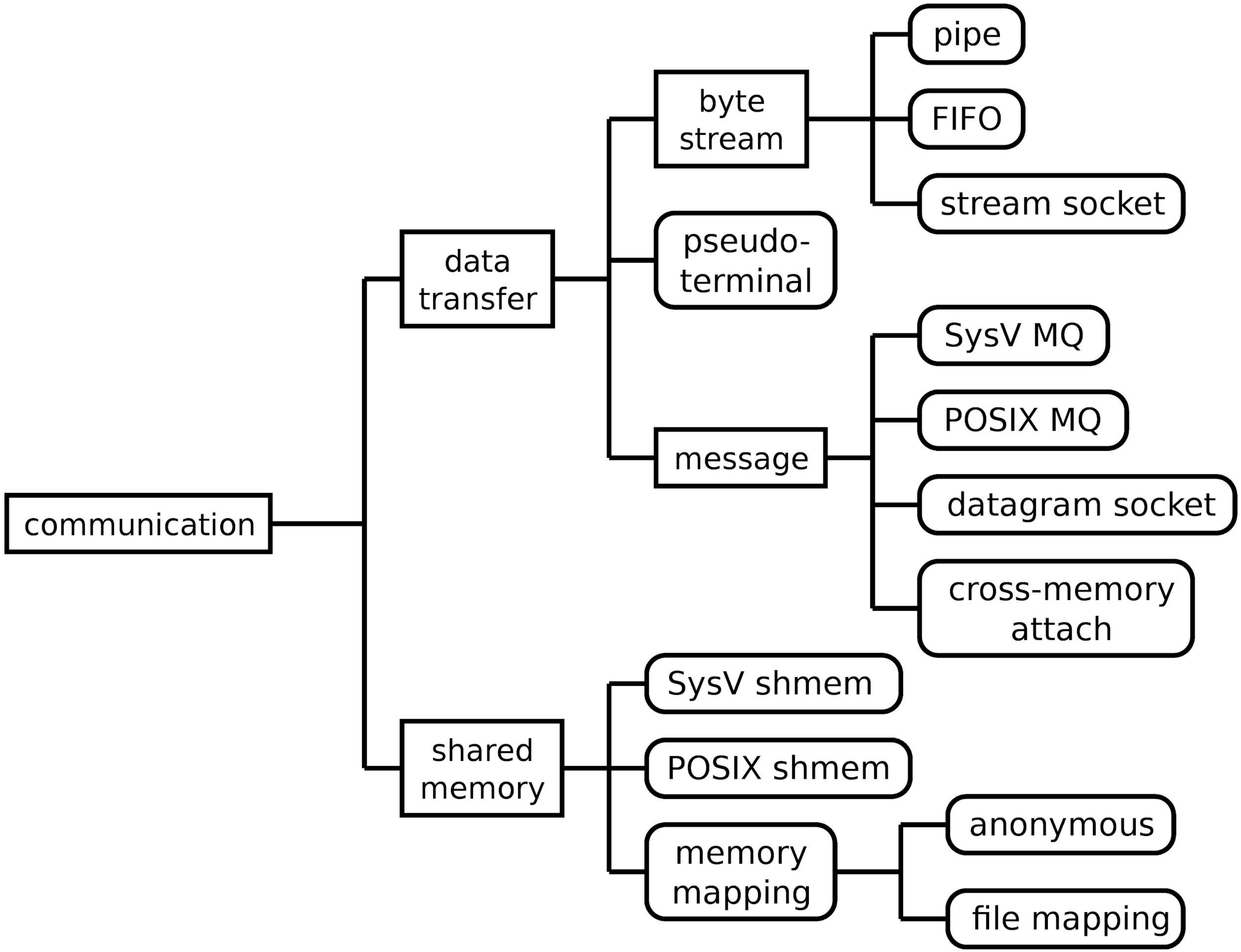
- O(1) used hard-to-maintain hacks for fairness
- CFS: Tasks get a “completely fair” CPU share
- Models an “ideal, precise, multi-tasking” CPU
- Basic idea with N tasks:
  - Record how much CPU time each task has been given
  - Schedule task with biggest delta to `tot_CPU_time/N`
  - Virtualize time (*virtual runtime*) to deal with priorities
  - Increase virtual runtime faster for lower-priority tasks
- No heuristics to distinguish tasks
- No run queues, uses a red-black tree



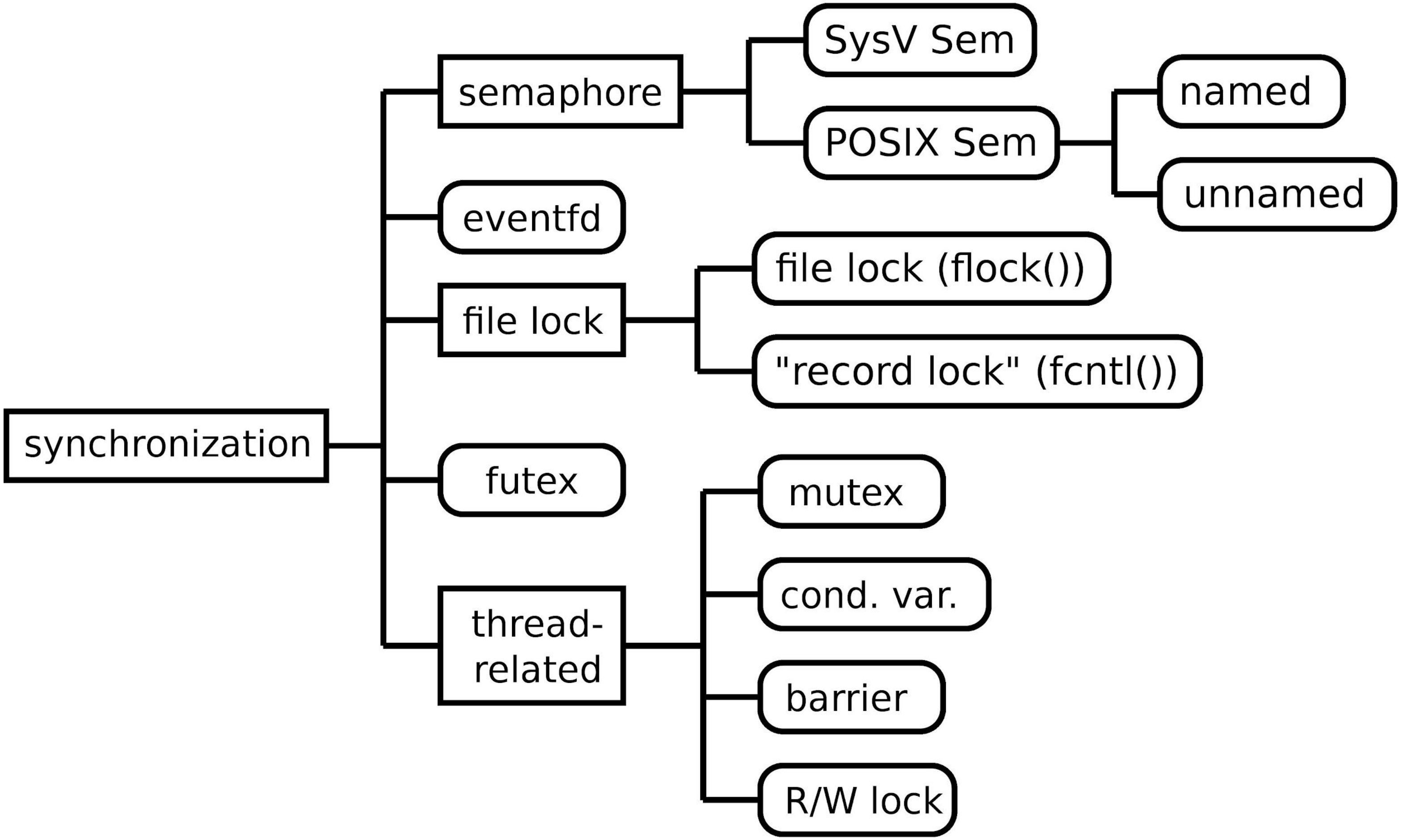
Linux CFS Scheduler

IPC

- Communication
  - Allow processes to exchange data
- Synchronization
  - Allow processes to synchronize execution
- Signals
  - Asynchronous notifications
- Multiple standards and interfaces...



IPC: Communication

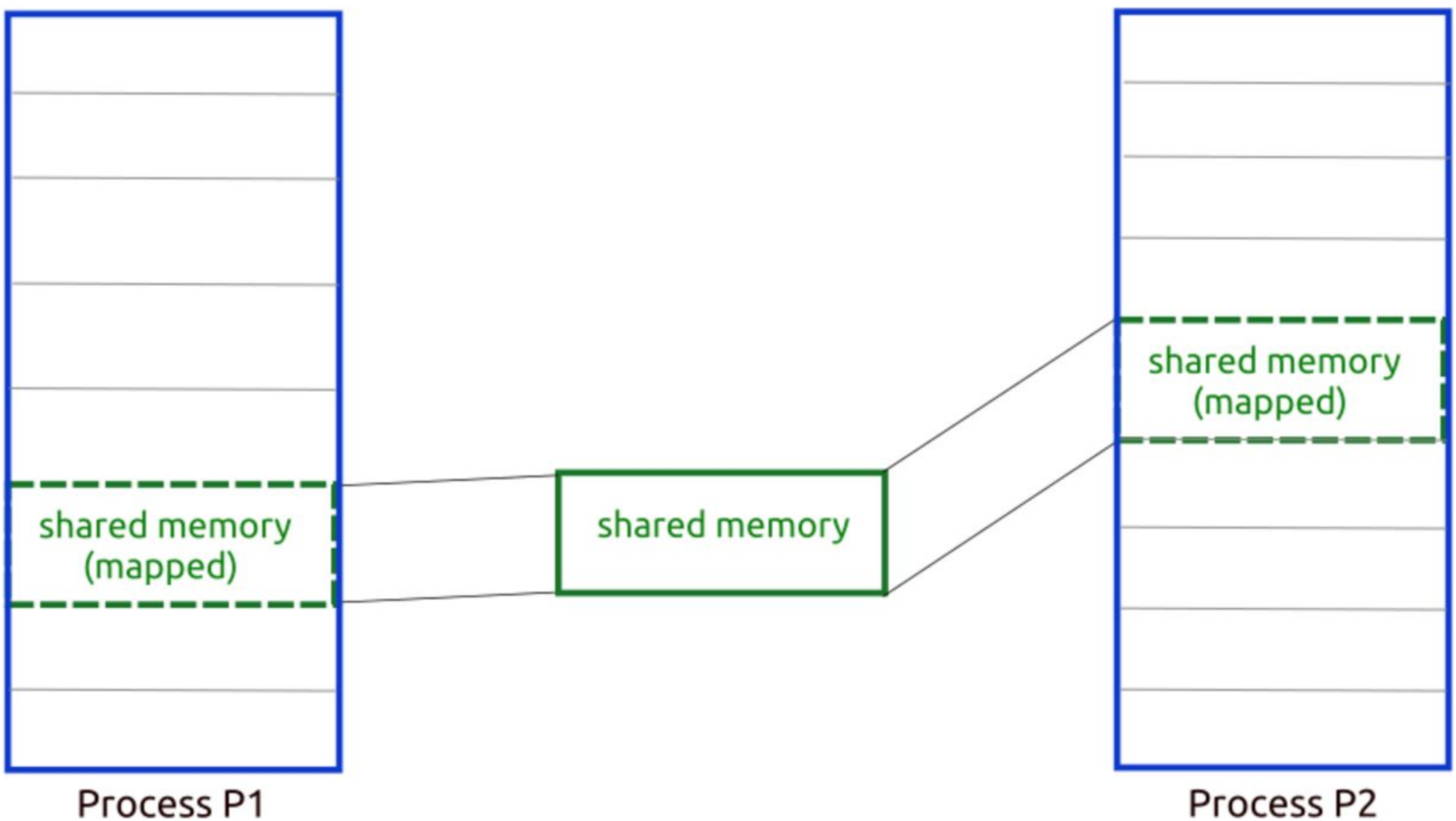


IPC: Synchronization

IPC: System V vs. POSIX

- Similar key IPC mechanisms
  - Shared memory
    - Share a region of memory
  - Semaphores
    - Synchronize with post/wait primitives
  - Message queues
    - Exchange messages between processes
- System V
  - Original UNIX implementation, +compatibility
- POSIX
  - Standardized later, +user-friendly, +features

System V: Shared Memory



System V: Shared Memory

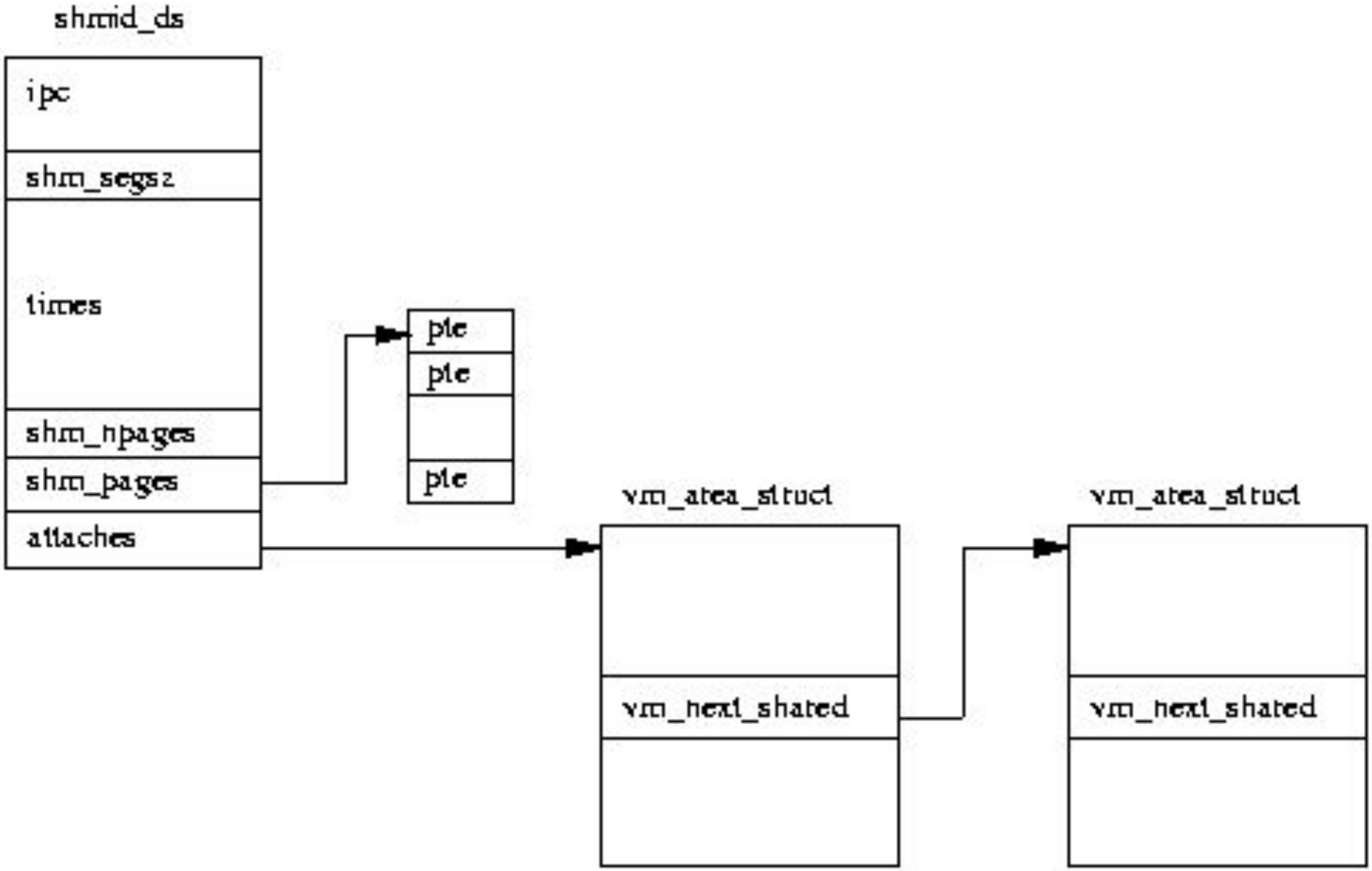
- Get/create shmem segment by key, perm:

```
int shmget(key_t key, size_t size, int shmflg);
```

- Attach / detach segment `shmid`:

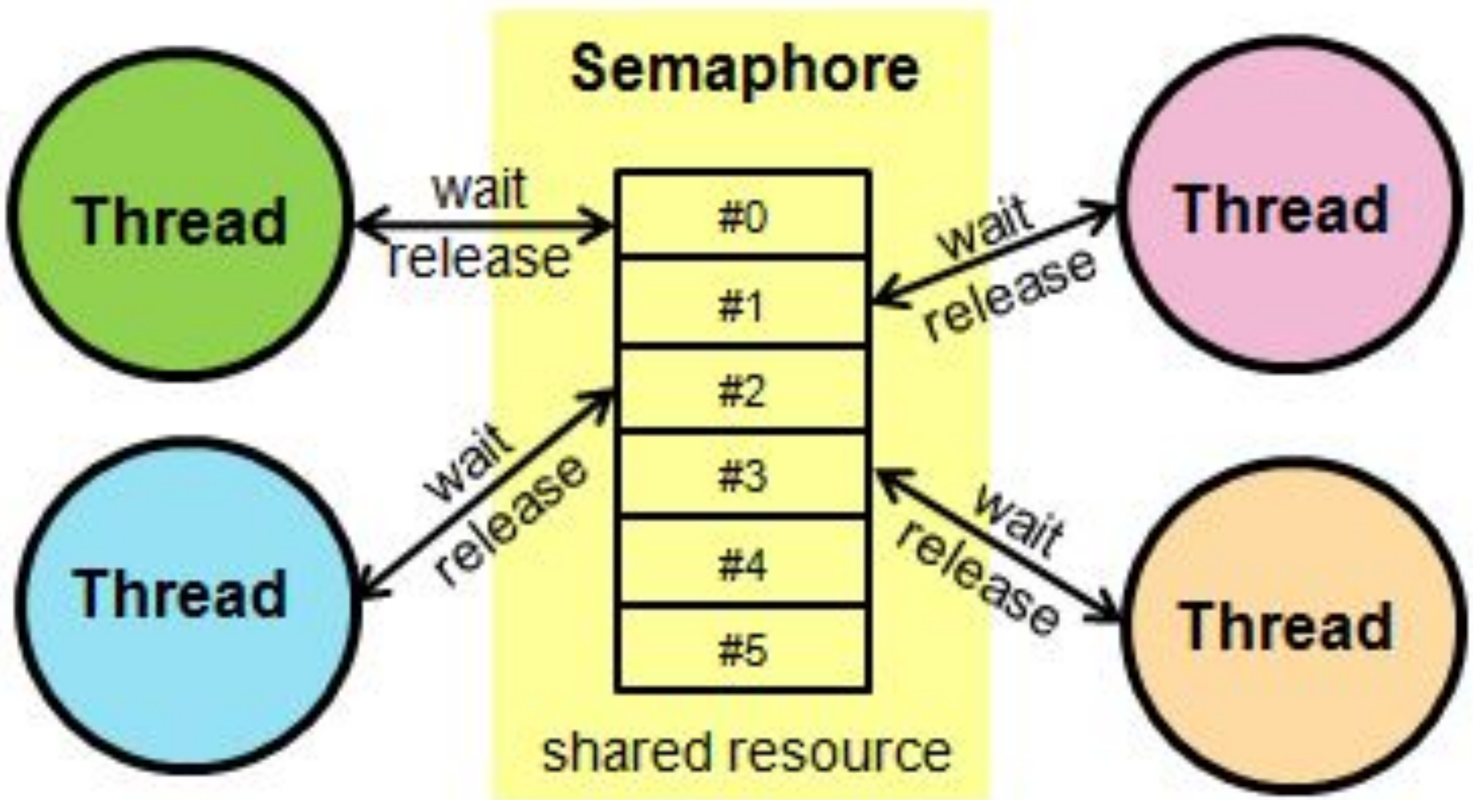
```
void *shmat(int shmid, void *addr, int shmflg);
int shmdt(void *addr);
```

- Any `addr` (if any) can be specified for `shmat` in different processes
- All the processes attaching the same segment share the same page frames



System V: Shared Memory Implementation

System V: Semaphores



System V: Semaphores

- Get/create semaphore set by key, perm:

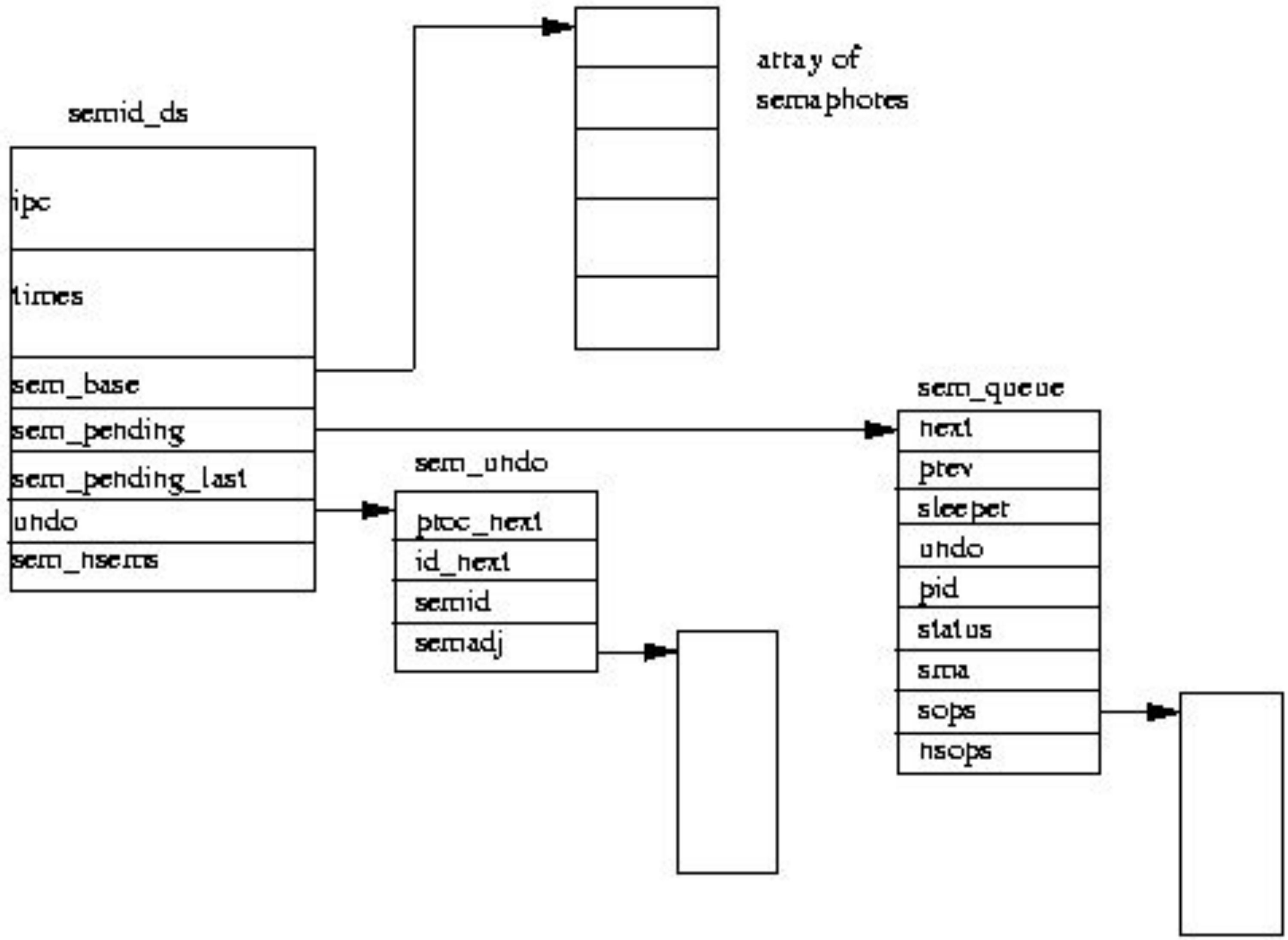
```
int semget(key_t key, int nsems, int semflg);
```

- Perform operations on semaphore set `semid`:

```
int semop(int semid, struct sembuf *sops, int n);
struct sembuf {
   unsigned short sem_num;  /* number */
   short          sem_op;   /* operation */
   short          sem_flg;  /* flags */
}
```

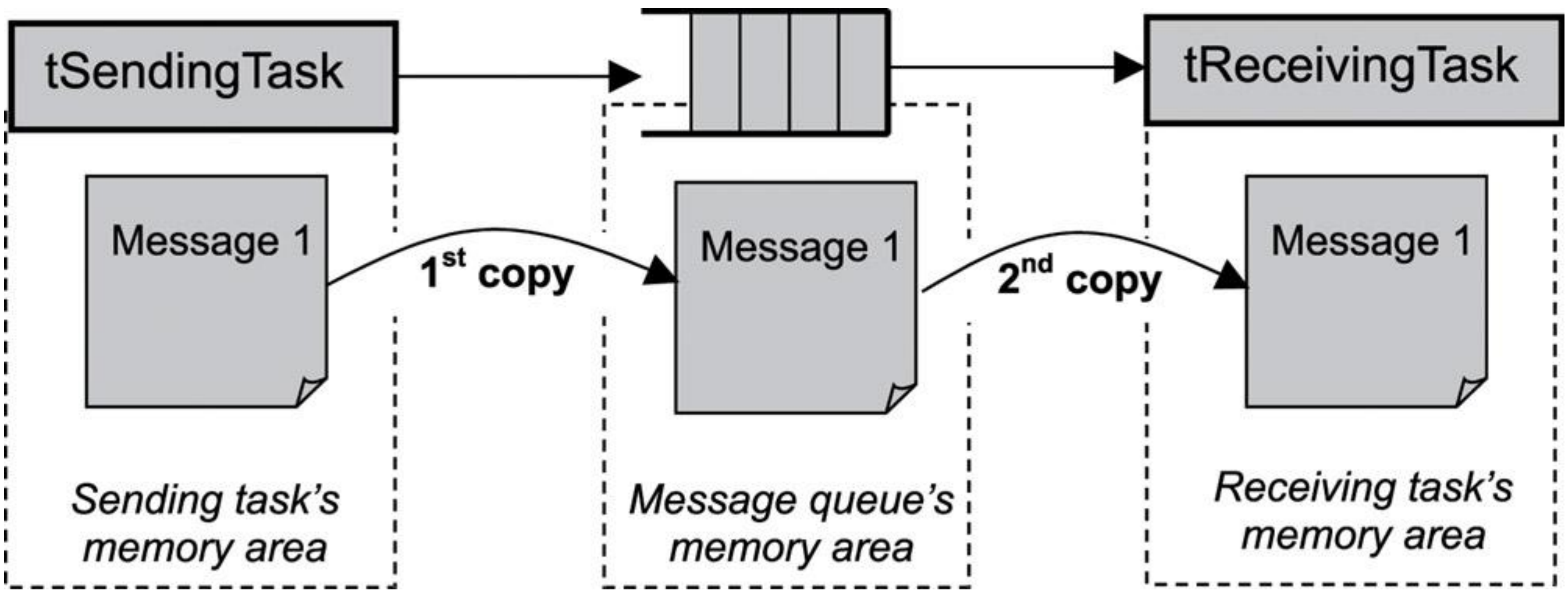
System V: Semaphores

- Initially semaphore value (`val`) is `0`
- Semaphore post (`sem_op > 0`):
  - Atomically increment semaphore by `sem_op`
- Semaphore wait (`sem_op < 0`):
  - If `val >= -sem_op`:
    - Atomically decrement semaphore by `-sem_op`
  - Else:
    - Block and wait for val >= `-sem_op`
- Semaphore waitzero (`sem_op == 0`):
  - Wait for `val == 0`



System V: Semaphores Implementation

System V: Message Queues



System V: Message Queues

- Get/create message queue by key, perm:

```
int msgget(key_t key, int msgflg);
```

- Send/receive message on message queue `msqid` with an optional type:

```
int msgsnd(int msqid, const void *msgp, size_t
msgsz, int msgflg);
```

```
ssize_t msgrcv(int msqid, void *msgp, size_t
msgsz, long msgtyp, int msgflg);
```

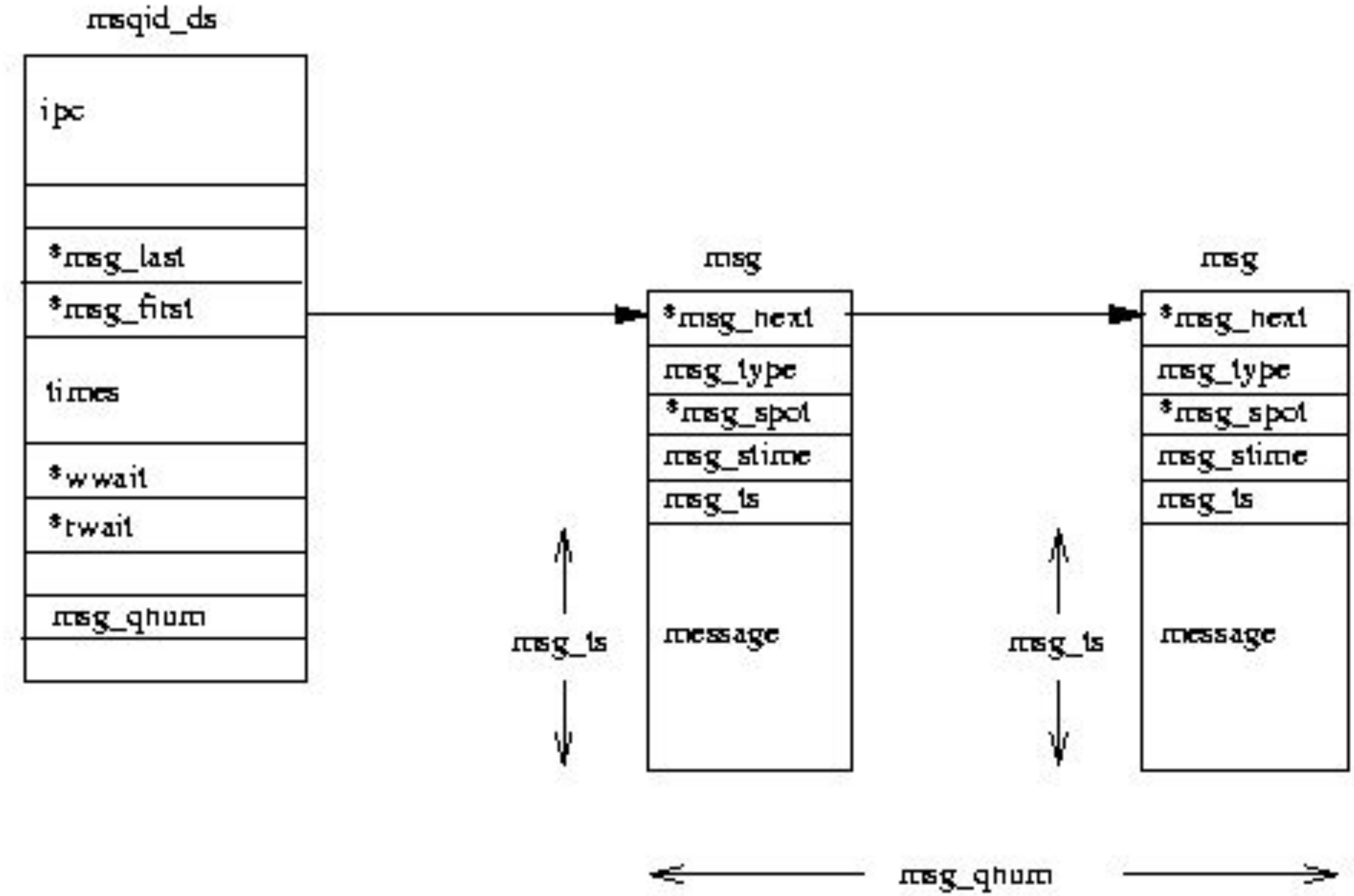
- Blocks if queue is full (send) or empty (receive). Queue is size-limited

System V: Message Queues

- Variable-size message of fixed format:

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
};
```

- Unlike pipes:
  - Not stream-oriented
  - Supports random queue access
  - Bidirectional
  - Always named



System V: Message Queue Implementation

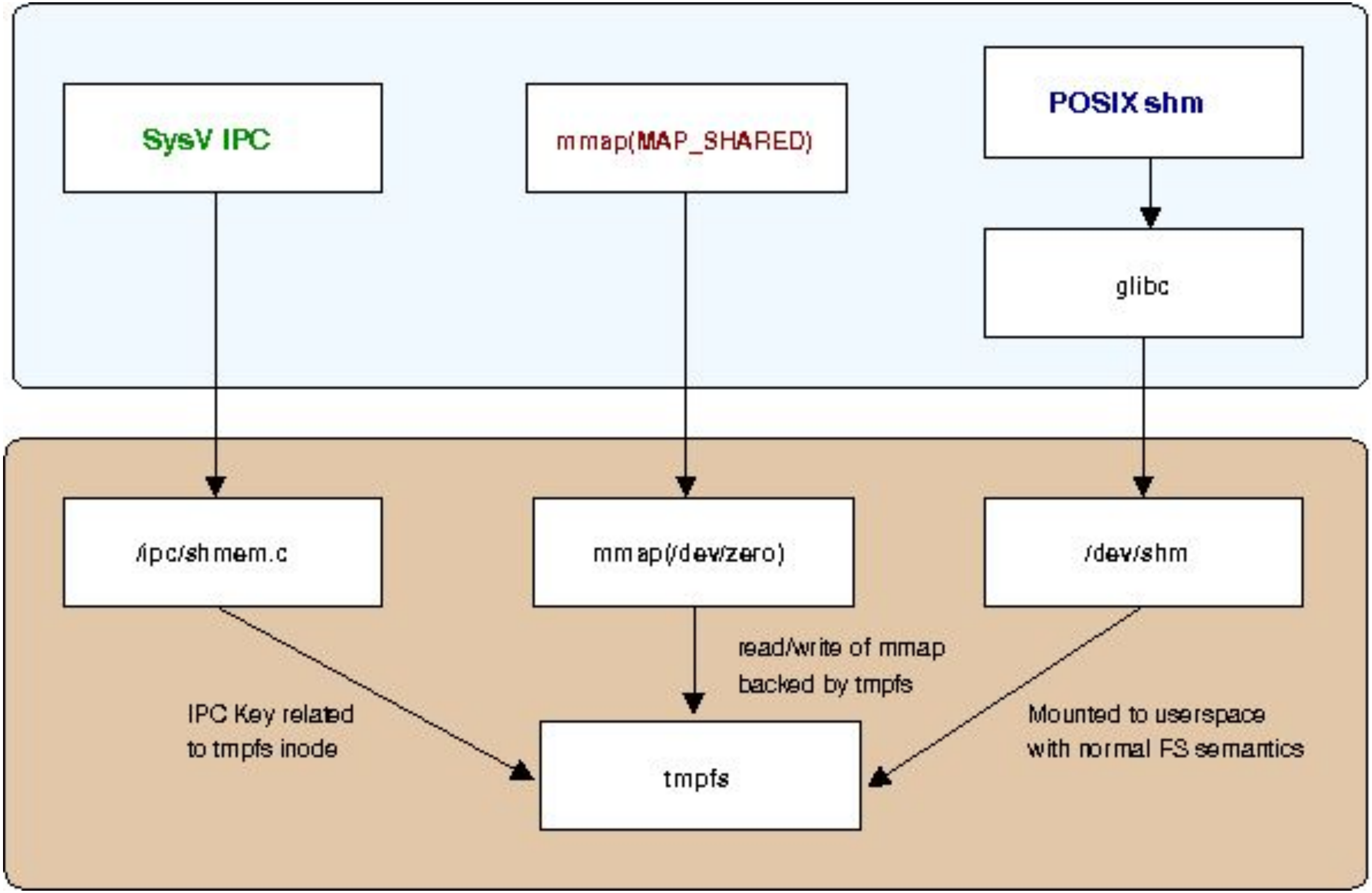
System V	Message queues	Semaphores	Shared memory
Object handle	msqid	semid	shmid
Create/open	msgget()	semget()	shmget() + shmat()
Control/Close	msgctl()	semctl()	shmctl / shmdt()
Perform IPC	msgsnd / msgrcv()	semop()	R/W

POSIX	Message queues	Semaphores	Shared memory
Object handle	mqd_t	sem_t *	fd
Create/open	mq_open()	sem_open()	shm_open() + mmap()
Close	mq_close()	sem_close()	munmap()
Perform IPC	mq_send /mq_receive()	sem_post / sem_wait()	R/W

System V vs. POSIX

POSIX IPC: Notable Differences

- Uses names, not keys (e.g., shm_open)
  - More in line with the UNIX model
- Uses reference counting (e.g., shm_unlink)
  - Easier to deallocate resources (and no `ipcs`/`ipcrm`)
- Provides thread safety
  - Easier to mix multiprocessing and multithreading
- Shared memory is file-oriented
  - Closer to other shared memory interfaces
  - Closer to native `tmpfs` implementation
  - `tmpfs` provides memory-backed filesystem with size limits + swap support



Modern Linux Shmem / `tmpfs`

References

[1] Bovet, Daniel P., and Marco Cesati. Understanding the Linux Kernel, 2005.

[2] Gorman, Mel. Understanding the Linux Virtual Memory Manager, 2004.

[3] “Linux Cross Reference,” n.d. https://elixir.bootlin.com/linux/latest/source.

[4] “Time Keeping,” n.d. https://www.kernel.org/doc/Documentation/timers/timekeeping.txt.

[5] “CFS Scheduler,” n.d. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[6] “Reducing Scheduling-Clock Ticks,” n.d. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.