Systemy operacyjne oparte na mikrojądrze na przykładzie Minix3.

Maciej Łaszcz, Wojciech Łowiec, Patryk Spanily

2 XII 2008

Systemy oparte na mikrojądrze

- Jądro systemu jest bardzo małe
- Architektura mocno modualrna
- Systemy takie charakteryzują się:
  - Bezpieczeństwem
  - Stabilnością
  - Odpornością na błędy
  - Wolniejszym działaniem
  - Samoleczeniem

Microkernel

- Jądra takich systemów nie udostępniają usług systemowych a jedynie mechanizmy potrzebne do ich implementacji:
  - Niskopoziomowe zarządzenie przestrzeniami adresowymi
  - Niskopoziomowe zarządzanie zasobami procesora – scheduling, zarządzanie watkami/procesami
  - IPC – komunikacja między procesami
  - Autoryzacja serwerów
  - Obsługa przerwań

Microkernel

Zasada minimalności Lindtke'go:

„A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.”

Microkernel

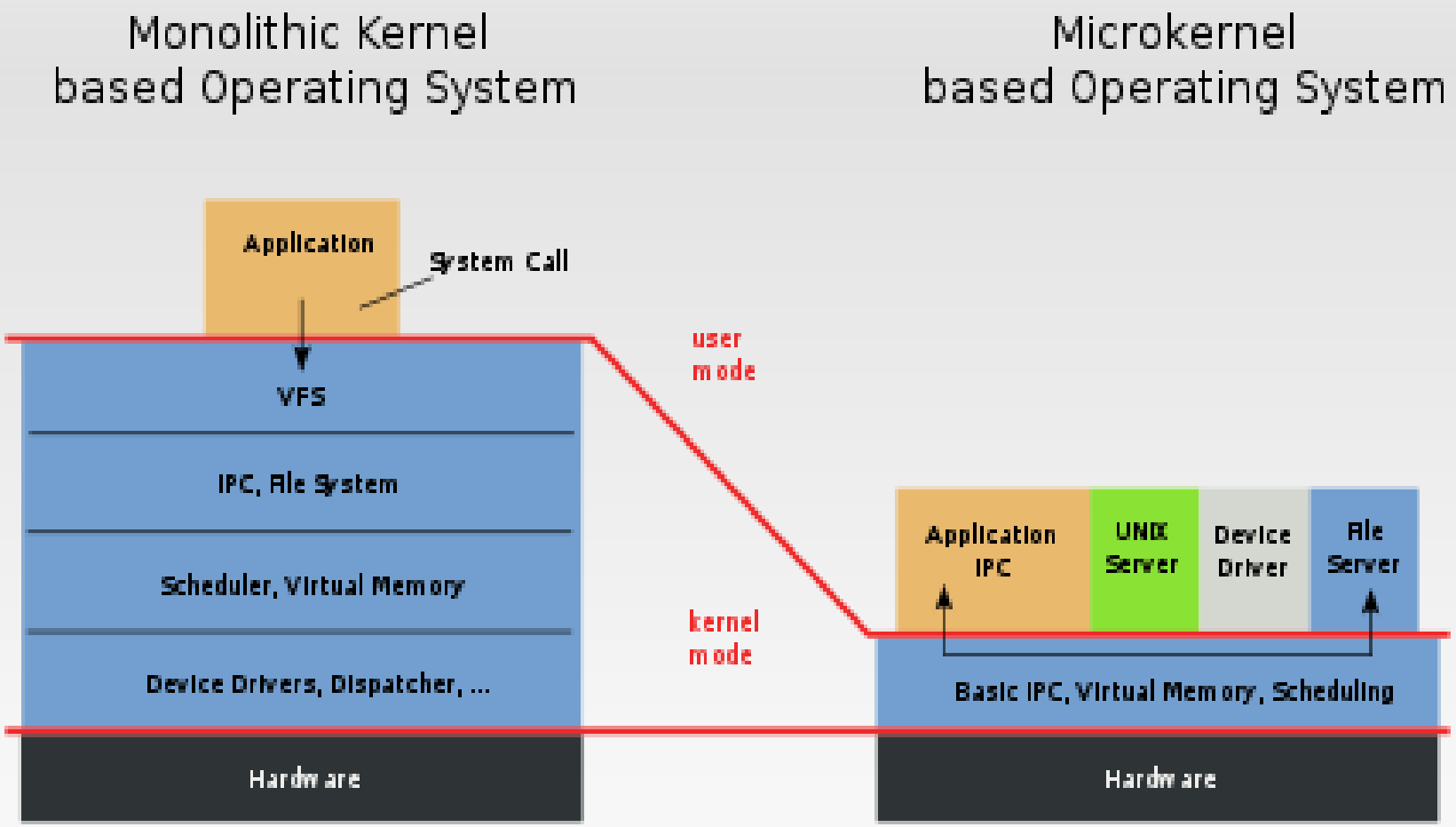
„Separation of mechanism and policy”

logika zaszyta w jądrze nie może być nadpisana w procesach trybu użytkownika, więc nie powinno jej się tam umieszczać

Microkernel

- Usługi systemowe dostarczane są przez serwery uruchomione w trybie użytkownika.
- Serwery uruchamiane są w trybie użytkownika
- Przykładowe serwery:
  - Serwer systemu plików
  - Serwer sterowników
  - Serwer stosu protokołów
  - Serwer procesów

Porównanie architektur



Mikrojądro – wywołania systemowe

Wywołania systemowe, uruchamiane przez procesy, zamieniane są na wiadomości wysyłane przez IPC do odpowiednich serwerów.

Przykłady systemów operacyjnych

- Amiga Exec / SG
- GNU Hurd
- Mach
- L4
- **Minix3**
- MkLinux
- Coyotos
- Integrity
- QNX
- Mac OS*

Zastosowania systemów

- Minix 3
- Mac OS
- Integrity
- QNX

Ogólnie systemy oparte na mikrojądrach używane są w systemach wbudowanych.

MINIX3 (Mini-unIX)

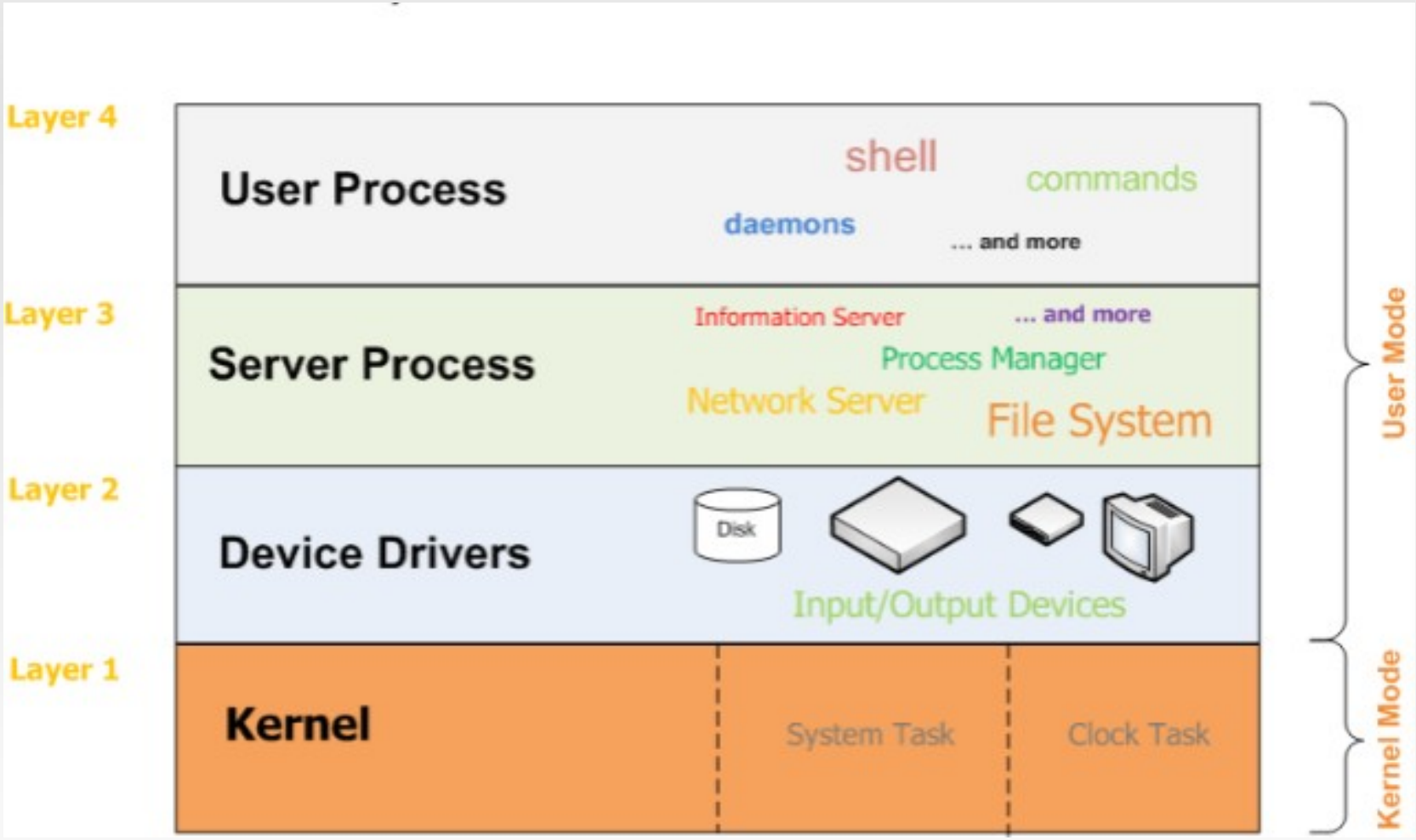
- Stworzony przez Andrew S. Tanenbaum'a
- Krótka historia:
  - **v.1.0** - dodany do książki „Operating Systems Design and Implementation” w 1987 r.
  - **v.1.5** - rok 1991
  - **v.2.0** - rok 1997, dodany do do drugiego wydania książki Tanenbauma, zgodny z POSIX
  - **v.3.1.2** - rok 2006, wydanie z narzędziami UNIXowymi (gcc, perl, vi ...), dodane X11

MINIX3 – opis systemu

- Komunikacja IPC jest synchroniczna
- W trybie jądra działają wątki:
  - Sys Task
  - Clock Task
  - Kod sterujący przepływem komunikatów
- Samoleczenie – reincarnation server
- 4 warstwy systemu (następny slajd)

(Dozwolone kanały i odbiorcy wiadomości są na stałe wpisane do tablicy procesów i zależne od klasy procesu (usr/sys/srv/drv) (src/kernel/table.c))

MINIX3 - architektura



MINIX3 - kernel

Sys task – w nieskończonej pętli odbiera od procesów żądania (w postaci komunikatów) sys_call, sprawdza ich poprawność i wykonuje odpowiednie czynności

MINIX3 - kernel

Clock task – za każdym odebraniem komunikatu od zegara sprzętowego sprawdza, czy aktualnie uruchomiony proces nie przekroczył swojego quantum, obsługuje też rejestrowalne timer'y

MINIX3 - kernel

Przerwania sprzętowe są mapowane na komunikaty wysyłane do odpowiednio zarejestrowanych sterowników

Dodatkowo na kod kernel'a przypada obsługa komunikatów

MINIX3 - scheduling

- Podział procesów na trzy klasy:
  - TSK
  - SRV
  - USR
- Procesy TSK są nie wywłaszczalne (wątki jądra)
- Procesy TSK i SRV są traktowane jako procesy systemowe
- Dodatkowy proces IDLE ustawiony na najniższy priorytet

MINIX3 – sterowniki

- Są procesami w przestrzeni użytkownika
- Komunikują się ze swoimi urządzeniami za pomocą wywołań systemowych
- Odbierają zapytania od serwerów i wykonują je w nieskończonej pętli
- Odbierają opakowane w wiadomości przerwania sprzętowe

MINIX3

Szczegóły użytych w MINIX'ie rozwiązań.

MINIX – SYS_TASK (kernel/system.c)

```
 1: PUBLIC void sys_task()
 2: {
 3: /* Main entry point of sys_task.  Get the message and dispatch on type. */
 4:   /* Initialize the system task. */
 5:   initialize();
 6:
 7:   while(TRUE) {
 8:     if((r=receive(ANY, &m)) != OK)
 9:       minix_panic("receive() failed", r);
10:
11:     if(m.m_source == SYSTEM)
12:       continue;
13:
14:     sys_call_code = (unsigned) m.m_type;
15:     call_nr = sys_call_code - KERNEL_CALL;
16:     who_e = m.m_source;
17:     okendpt(who_e, &who_p);
18:     caller_ptr = proc_addr(who_p);
19:
20:     [...]
21:
22:     if (!GET_BIT(priv(caller_ptr)->s_k_call_mask, call_nr))
23:       result = ECALLDENIED;
```

MINIX – SYS_TASK c.d.

```
20:       [...]
21:
22:       if (!GET_BIT(priv(caller_ptr)->s_k_call_mask, call_nr))
23:         result = ECALLDENIED;
24:       else {
25:         result = (*call_vec[call_nr])(&m); /* handle the system call */
26:       }
27:
28:       [...]
29:
30:       if (result != EDONTREPLY) {
31:         /* Send a reply, unless inhibited by a handler function.
32:          * Use the kernel function lock_send() to prevent a system
33:          * call trap.
34:          */
35:         m.m_type = result;            /* report status of call */
36:         if(WILLRECEIVE(caller_ptr, SYSTEM)) {
37:           lock_send(m.m_source, &m);
38:         }
39:       }
40:     }
41:   }
```

MINIX – CLOCK_TASK (kernel/clock.c)

- Dzieli struktury danych z resztą jądra
- Obsługuje przerwanie zegara (HARD_INT):
  - Uruchamia w swoim kontekście zarejestrowane funkcje budzików
  - Uruchamia schedulera, gdy aktywny proces wykorzystał cały swój czas
  - Nalicza czas aktywnego procesu

MINIX – CLOCK_TASK c.d.

```
 1: PUBLIC void clock_task()
 2: {
 3: /* Main program of clock task. If the call is not HARD_INT it is an error.
 4:  */
 5:   message m;       /* message buffer for both input and output */
 6:   int result;      /* result returned by the handler */
 7:
 8:   init_clock();    /* initialize clock task */
 9:
10:   /* Main loop of the clock task.  Get work, process it. Never reply. */
11:   while(TRUE) {
12:     /* Go get a message. */
13:     result = receive(ANY, &m);
14:
15:     if(result != OK)
16:         minix_panic("receive() failed", result);
17:
18:     /* Handle the request. Only clock ticks are expected. */
19:     switch (m.m_type) {
20:     case HARD_INT:
21:         do_clocktick(&m); /* handle clock tick */
22:         break;
23:     default: /* illegal request type */
24:         kprintf("CLOCK: illegal request %d from %d.\n",
25:             m.m_type, m.m_source);
26:     }
27:   }
28: }
```

MINIX – CLOCK_TASK c.d.

```
 1:   /* A process used up a full quantum. The interrupt handler stored this
 2:    * process in 'prev_ptr'.  First make sure that the process is not on the
 3:    * scheduling queues.  Then announce the process ready again. Since it has
 4:    * no more time left, it gets a new quantum and is inserted at the right
 5:    * place in the queues.  As a side-effect a new process will be scheduled.
 6:    */
 7:   if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
 8:       if(prev_ptr->p_rts_flags == 0) {  /* if it was runnable .. */
 9:     lock_dequeue(prev_ptr);     /* take it off the queues */
10:         lock_enqueue(prev_ptr);     /* and reinsert it again */
11:       } else {
12:     kprintf("CLOCK: %d not runnable; flags: %x\n",
13:         prev_ptr->p_endpoint, prev_ptr->p_rts_flags);
14:       }
15:   }
16:
17:   /* Check if a clock timer expired and run its watchdog function. */
18:   if (next_timeout <= realtime) {
19:     tmrs_exptimers(&clock_timers, realtime, NULL);
20:     next_timeout = (clock_timers == NULL) ?
21:         TMR_NEVER : clock_timers->tmr_exp_time;
22:   }
```

MINIX – komunikaty w systemie

- Komunikaty są stałej wielkości
- Do przesyłu komunikatów używa się zawsze takiej samej struktury (niezależnie od typu)
- Komunikaty mogą być wysyłane i odbierane poprzez:
  - SENDREC
  - SEND
  - RECEIVE
  - NOTIFY (niskopoziomowe)

MINIX - scheduler

- Dokładnie tyle kolejek ile jest priorytetów
- Proces po skolejkowaniu trafia:
  - Na początek kolejki, jeśli nie wykorzystał jeszcze swojego czasu (time slice)
  - Na koniec, w p.p.
- W funkcji pick_proc() wybierany jest następny proces do uruchomienia. Jest to proces o najniższym priorytecie ze wszystkich gotowych

MINIX – scheduler (c.d.)

```
 1: PUBLIC void enqueue(rp)
 2: register struct proc *rp;   /* this process is now runnable */
 3: {
 4: /* Add 'rp' to one of the queues of runnable processes.  This function is
 5:  * responsible for inserting a process into one of the scheduling queues.
 6:  * The mechanism is implemented here.   The actual scheduling policy is
 7:  * defined in sched() and pick_proc().
 8:  */
 9:   int q;                    /* scheduling queue to use */
10:   int front;                    /* add to front or back */
11:
12: #if DEBUG_SCHED_CHECK
13:   if(!intr_disabled()) { minix_panic("enqueue with interrupts enabled", NO_NUM); }
14:   CHECK_RUNQUEUES;
15:   if (rp->p_ready) minix_panic("enqueue already ready process", NO_NUM);
16: #endif
17:
18:   /* Determine where to insert to process. */
19:   sched(rp, &q, &front);
20:
21:   /* Now add the process to the queue. */
22:   if (rdy_head[q] == NIL_PROC) {        /* add to empty queue */
23:       rdy_head[q] = rdy_tail[q] = rp;       /* create a new queue */
24:       rp->p_nextready = NIL_PROC;       /* mark new end */
25:   }
```

MINIX – scheduler (c.d.)

```
18:   /* Determine where to insert to process. */
19:   sched(rp, &q, &front);
20:
21:   /* Now add the process to the queue. */
22:   if (rdy_head[q] == NIL_PROC) {          /* add to empty queue */
23:       rdy_head[q] = rdy_tail[q] = rp;       /* create a new queue */
24:       rp->p_nextready = NIL_PROC;         /* mark new end */
25:   }
26:   else if (front) {                 /* add to head of queue */
27:       rp->p_nextready = rdy_head[q];        /* chain head of queue */
28:       rdy_head[q] = rp;                 /* set new queue head */
29:   }
30:   else {                        /* add to tail of queue */
31:       rdy_tail[q]->p_nextready = rp;        /* chain tail of queue */
32:       rdy_tail[q] = rp;                 /* set new queue tail */
33:       rp->p_nextready = NIL_PROC;         /* mark new end */
34:   }
35:
36:   /* Now select the next process to run, if there isn't a current
37:    * process yet or current process isn't ready any more, or
38:    * it's PREEMPTIBLE.
39:    */
40:   if(!proc_ptr || proc_ptr->p_rts_flags ||
41:     (priv(proc_ptr)->s_flags & PREEMPTIBLE)) {
42:      pick_proc();
43:   }
44: }
```

MINIX – scheduler (c.d.)

```
 1: PRIVATE void sched(rp, queue, front)
 2: register struct proc *rp;           /* process to be scheduled */
 3: int *queue;                 /* return: queue to use */
 4: int *front;                 /* return: front or back */
 5: {
 6: /* This function determines the scheduling policy.  It is called whenever a
 7:  * process must be added to one of the scheduling queues to decide where to
 8:  * insert it.  As a side-effect the process' priority may be updated.
 9:  */
10:   int time_left = (rp->p_ticks_left > 0);   /* quantum fully consumed */
11:
12:   /* Check whether the process has time left. Otherwise give a new quantum
13:    * and lower the process' priority, unless the process already is in the
14:    * lowest queue.
15:    */
16:   if (! time_left) {                /* quantum consumed ? */
17:       rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
18:       if (rp->p_priority < (IDLE_Q-1)) {
19:           rp->p_priority += 1;          /* lower priority */
20:       }
21:   }
22:
23:   /* If there is time left, the process is added to the front of its queue,
24:    * so that it can immediately run. The queue to use simply is always the
25:    * process' current priority.
26:    */
27:   *queue = rp->p_priority;
28:   *front = time_left;
29: }
```

MINIX – scheduler (c.d.)

```
 1: PRIVATE void pick_proc()
 2: {
 3: /* Decide who to run now.  A new process is selected by setting 'next_ptr'.
 4:  * When a billable process is selected, record it in 'bill_ptr', so that the
 5:  * clock task can tell who to bill for system time.
 6:  */
 7:   register struct proc *rp;         /* process to run */
 8:   int q;                    /* iterate over queues */
 9:
10:   /* Check each of the scheduling queues for ready processes. The number of
11:    * queues is defined in proc.h, and priorities are set in the task table.
12:    * The lowest queue contains IDLE, which is always ready.
13:    */
14:   for (q=0; q < NR_SCHED_QUEUES; q++) {
15:       if ( (rp = rdy_head[q]) != NIL_PROC) {
16:           next_ptr = rp;            /* run process 'rp' next */
17: #if 0
18:       if(rp->p_endpoint != 4 && rp->p_endpoint != 5 && rp->p_endpoint != IDLE && rp->p_endpoint != SYSTEM)
19:         kprintf("[run %s]",  rp->p_name);
20: #endif
21:           if (priv(rp)->s_flags & BILLABLE)
22:               bill_ptr = rp;            /* bill for system time */
23:           return;
24:       }
25:   }
26:   minix_panic("no ready process", NO_NUM);
27: }
```

MINIX – przepływ komunikatów

Omówienie kodu z minix_src.pdf

A jak to jest w praktyce z MINIX'em?

Wg Tanenbaum'a – 5-10% straty

Rzeczywistość pokazuje inaczej

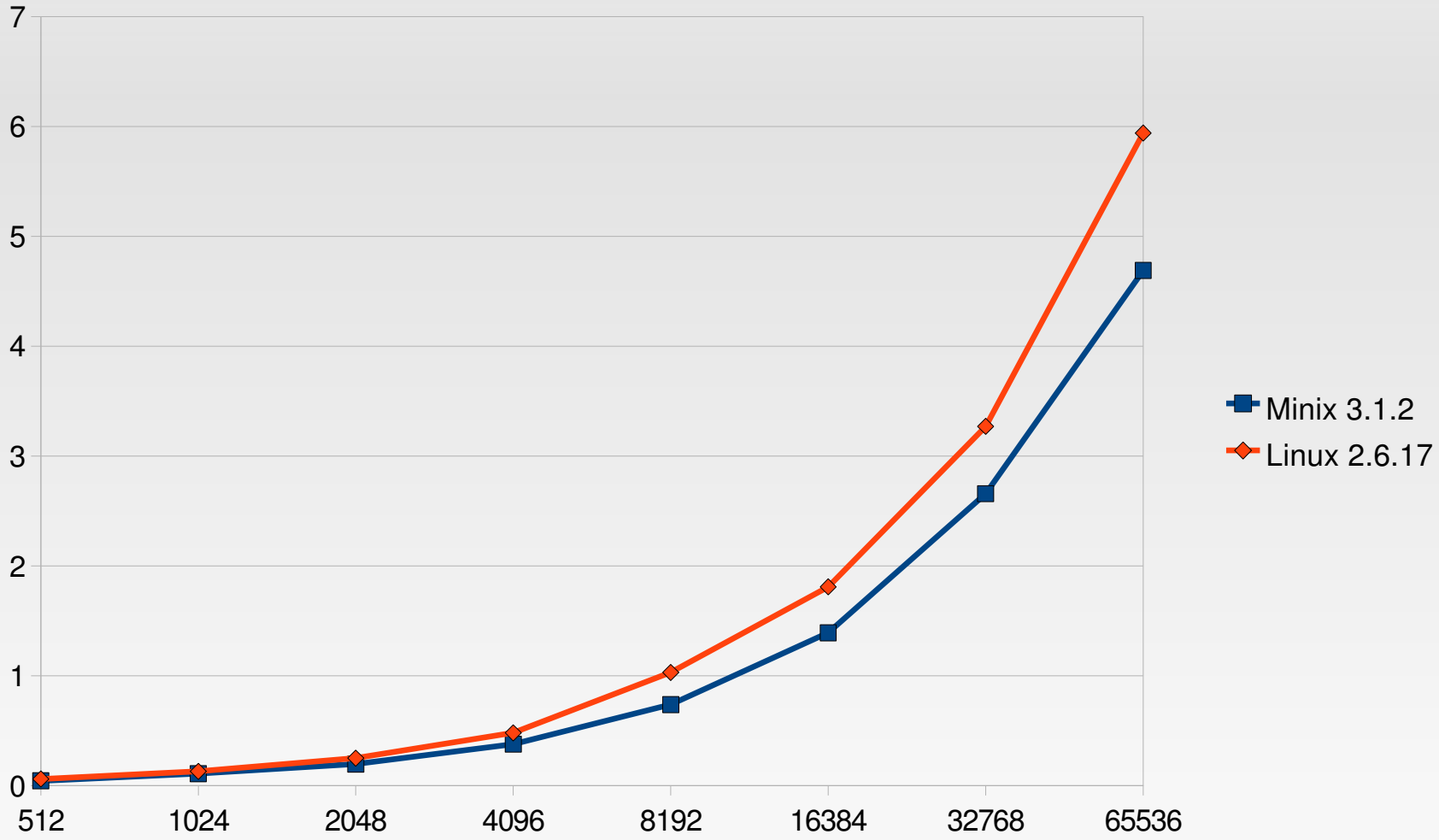
Trudności w testowaniu

Testowanie

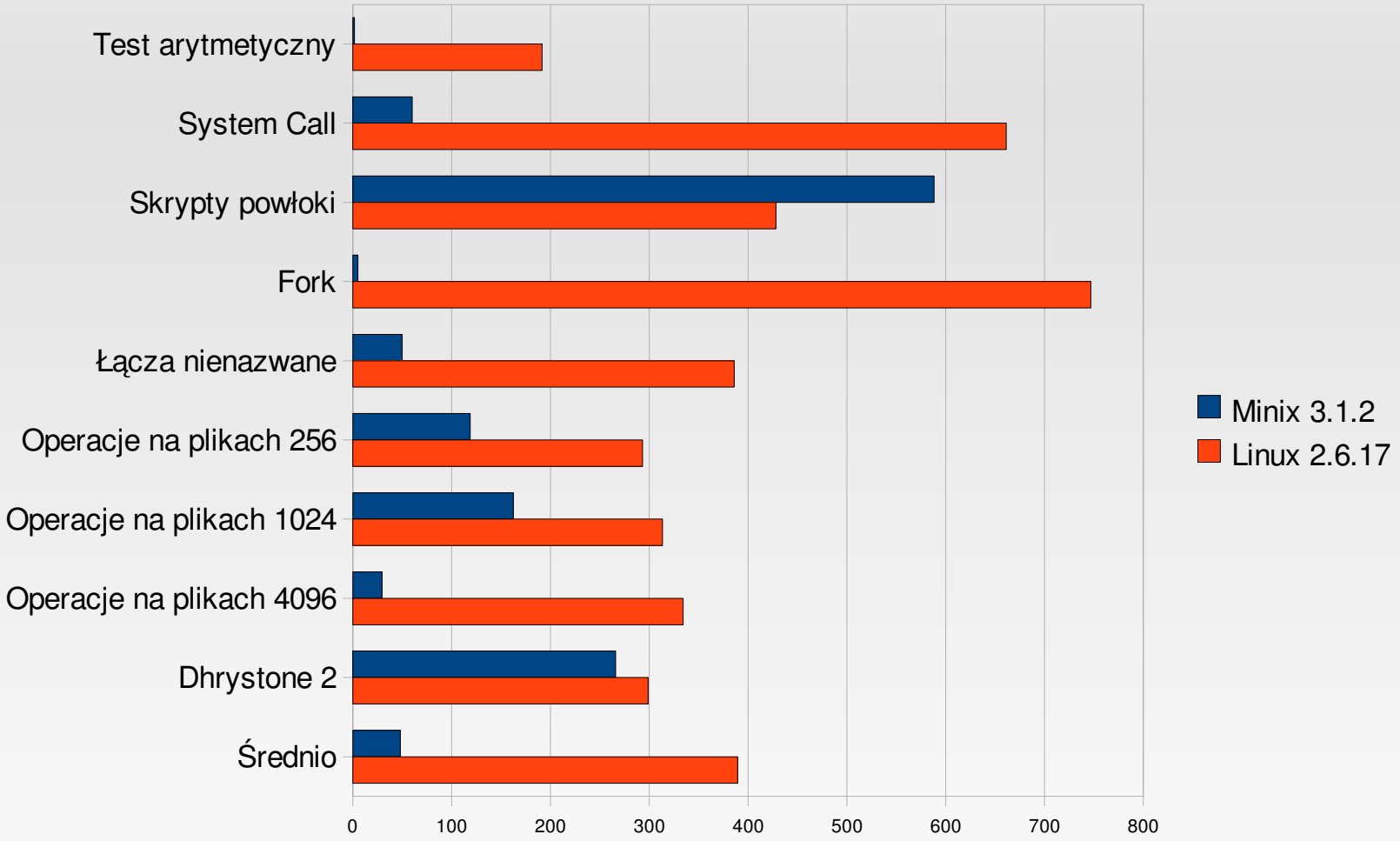
Benchmarki: IOTest, UnixBench

Maszyna testowa + ograniczenia

IOTest 2.31


7
6
5
4
3
2
1
0
512
1024
2048
4096
8192
16384
32768
65536
Minix 3.1.2
Linux 2.6.17

UnixBench 4.0.1



Torvalds – Tanenbaum

Minix jako ”matka” Linuksa

Komentarz A. Tanenbaum'a

Debata na comp.os.minix

Wady i zalety, Minix kontra Linux

Co zostało po czasie?

Mikrojądro – podsumowanie

Zalety:

- Odporność na błędy
- Stabilność
- Przyjemne testowanie

- Mniejsze jądro – łatwiej zoptymalizować...
- …i zrozumieć

Mikrojądro - podsumowanie

Wady:

- Mniejsza wydajność (o tym więcej za chwilę)
- Wielowątkowość trudniejsza w obsłudze
- Trudne wykrywanie błędów w komunikacji
- Trudność w projektowaniu

Pytania?