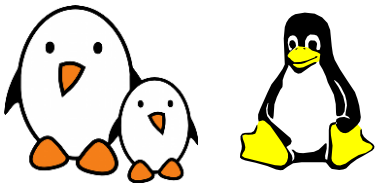
Open Source Programování

http://rtime.felk.cvut.cz/osp/

Pavel Píša
<pisa@fel.cvut.cz>
http://cmp.felk.cvut.cz/~pisa
Michal Sojka
František Vacek
DCE FEL ČVUT



Zdroje limitující aplikace

- Výkon a počet CPU (standard MIPS, FLOPS)
- Velikost fyzické paměti (počet stránek)
- Šířka pásma pro přístup k paměti
- Šířka pásma pro přístup k datům na disku
- Šířka pásma síťového připojení/infrastruktury

Zpracování událostí

- Na různých úrovních
  - Na úrovni hardware
    sběrnice, CPU, programový kanál, programový kanál s přerušením (IRQ), s přímým přístupem k paměti (DMA), scatter-gather, s I/O koprocesorem/inteligentním řadičem
  - V jádře operačního systému
  - **Na úrovni knihoven a systémových volání**
  - V grafickém subsystému a aplikačním frameworku
  - V síti uzlů (komunikace TCP/IP, Ethernet, HTTP, cluster OpenMP, PCI Express, ...)
  - Jako reakce na uživatele (pohyb myší, klávesnice atd.)
  - V řízeném systému se zpětnou vazbou
    robot, řídicí aplikace, letadlo, vlak, drive-by-wire, haptic device, force feedback atd.

Základní přístupy

- **Periodické testování** změny, nastavené události (polling)
  - Nejprimitivnější, **plýtvá** časem procesoru v případě, že k události ještě nedošlo
  - Když k události dojde, tak je pro náhodný interval vzniku průměrná prodleva T/2, **velká latence**
  - Častěji⇒plýtvání, delší perioda⇒latence
  - **Trvalá zátěž** na každou sledovanou událost O(n_registered+n_active)
- Automatické doručení informace o změně a čekání na události
  - IRQ→CPU nebo raději DMA a IRQ po větším bloku dat
  - Nutnost dopravit událost z jádra OS přes nějaké **jednotné API**
  - Hlavní činnosti zatěžující systém a uživatele jsou **registrace** událostí a po jejich příchodu **distribuce** těm, kterým jsou **určeny**

Události a SMP

- Typicky se jedná o vstupně výstupní (I/O) události, časové limity (timeout) a události v absolutním čase
- Využití SMP při klient-server architektuře
  - Vytvářet na zpracování každé události nové vlákno
    většinou příliš velká režie a pro synchronní protokoly není potřeba, častěji **jedno vlákno na jednoho klienta** + vlákno pro příjem spojení a zakládání vláken, i to je však při tisícovkách klientů režie nad možnosti HW a propustnost klesá
  - Zpracovávat vše v jednom procesu/vlákně – pokud neběží další aplikace nebo systémové úlohy tak nevyužije více CPU. Neuplatňuje se režie na další vlákna (např. Linux každé vlákno přes 4 nebo 8kB nestránkovatelné paměti v jádře + několik stránek na zásobník a další data v userspace)
  - Většinou se používá kombinace přístupů, určité množství připravených vláken nebo procesů, které požadavky přebírají – **thread-pool**, je potřeba řídit přidělování úloh, buď z userspace nebo přímo v jádře OS, je výhodné řídit max. počet aktivních threadů, aby nedocházelo k přílišnému počtu přepínání (**interaktivní** versus **dávkové** úlohy). Obvykle pravidlo **2**× až **4**× více procesů/vláken než je CPU

Více událostí na jedno vlákno

- Veškeré operace by měly být **neblokující** (pozor na DNS a jiné blok.)
  POSIX open/fcntl O_NONBLOCK, Win32 CreateFile FILE_FLAG_OVERLAPPED attribute
- Multiplex/dopravení události a vyhledání cíle
  - Na vyšší úrovni například
    - Zpráva/**broadcast** do stromu/podstromu grafických objektů (podle topologie, aktivace/focus, pozice)
    - **Signal-slot** mechanizmus
    - U **klient-server** aplikací a protokolů často **1:1**
  - Potřeba navázat vyšší vrstvu na vrstvy nižší (např. Glib na epoll)
    - Většinou přes **deskriptory** (zobecněných) **souborů** – file-handle
    - Možné i jiné přístupy (někdy dokonce nutné)
      - Synchronizační události **mutexy**, **signály**, **IPC**
      - Doprava událostí jako **zprávy** (např. Win32 grafika)
      - **Rendezvous** – setkání server/klient především u mikrojader

Vlastní zkušenosti na základě

- uLEvPoll Library - uLan/Universal Light Event Poll Library
  http://ulan.git.sourceforge.net/git/gitweb.cgi?p=ulan/ulevpoll
  http://ulan.sourceforge.net/pdf/ulevpoll.pdf
- Původní záměr: minimální obálka nad **libevent** a **glib** (Gtk, Qt)
- Jako první implementovaný pro jednoduché aplikace **poll**
- Po analýze omezení libevent-1.x i vlastní implementace **epoll**
- Postupně vyřešeno kombinování s **glib** a **libevent**-1.x a 2.x
- Pro minimální kompatibilitu s Windows přidán starý dobrý Unixový **select**
- V rozpracovaném stavu Windows **WaitForMultipleObjectsEx**

Kostra implementace serveru

```
int sockfd, client_fd;
struct sockaddr_in my_addr;
struct sockaddr_in client_addr;
int sin_size;
int yes = 1;
int retval;

sockfd = socket(PF_INET, SOCK_STREAM, 0);
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes))

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(MYPORT);
my_addr.sin_addr.s_addr = INADDR_ANY;
memset(&(my_addr.sin_zero), 0, 8);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
listen(sockfd, 10 /*BACKLOG*/);

while (1) { /* Wait for client to show up*/
  /* wait for activity on sockfd or other fds */
  /* accept new client */
  client_fd = accept(server_sock,(struct sockaddr *) &client_addr,
                     &size);
}
```

Volání select()

```
#include <sys/select.h>
fd_set rdfds, readable, wrfds, writable;

FD_SET(fd, &readable);
do{
   rdfds = readable; wrfds = writable;
   n = select(maxfd, rdfds, wrfds, exfds, &tout);
   if (FD_ISSET(fd, rdfds)) { read }
   FD_SET(fd, &writable);
   FD_CLR(fd, &readable);
} while(!exit_loop);
```

- Nevýhodné, trvalé procházení celého bitového pole deskriptorů jak na straně uživatele, tak na straně jádra a to do maximálního použitého FD
- Na straně jádra O(fd_max+n_active) na každé volání, kratší pokud je již aktivní
- Na straně uživatele stejné nebo lze optimalizovat na O(n_registered+n_active)
- Limit pro všechna vlákna dohromady FD_SETSIZE (1024), na Linuxu lze i překročit

Volání poll()

```
#include <poll.h>

struct pollfd[MAX4THREAD];
array[i].fd = fd;
array[i].events = POLLIN;
do {
   n = poll(array, max, timeout);
   if (array[i].revents & POLLIN) { read }
   array[i].events = POLLOUT;
} while(!exit_loop);
```

- Na straně jádra O(n_registered+n_active) na každé volání
- Na straně uživatele stejné, nemá limit a overhead na FD použité v jiných threadech
- Výhoda je, že není třeba kopírovat požadavky mezi poli jako u *fd_set*, jsou od sebe oddělené *events* a *revents*

Volání epoll()

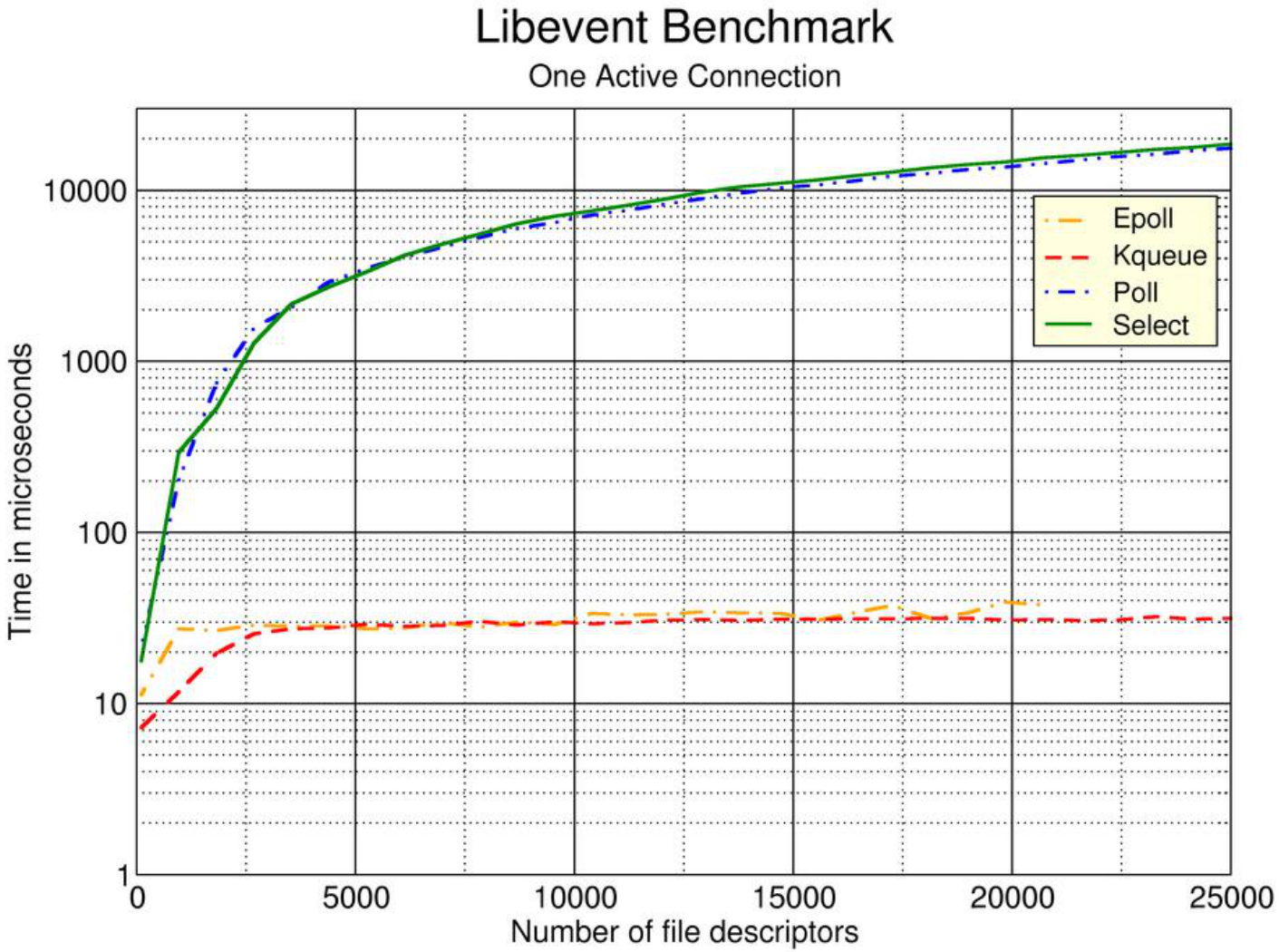
```
#include <sys/epoll.h>

epfd = epoll_create(0);
evt.data.fd = fd;
evt.events = EPOLLIN;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &evt);

do {
  epoll_wait(epfd, res_buff, res_size, tout);

  evt.data.fd = fd;
  evt.events = EPOLLOUT;
  epoll_ctl(epfd, EPOLL_CTL_MOD, fd, &evt);
} while(!exit_loop);
```

- Na straně jádra O(n_active+n_filter_changed) na volání; O(1) na každou událost
- Na straně uživatele v podstatě stejné, přitom na jedno volání může být přeneseno tolik událostí, na kolik je nabídnut buffer
- Požadavky se registrují přímo ke zdrojům událostí v jádře, wait queue

Další systémy

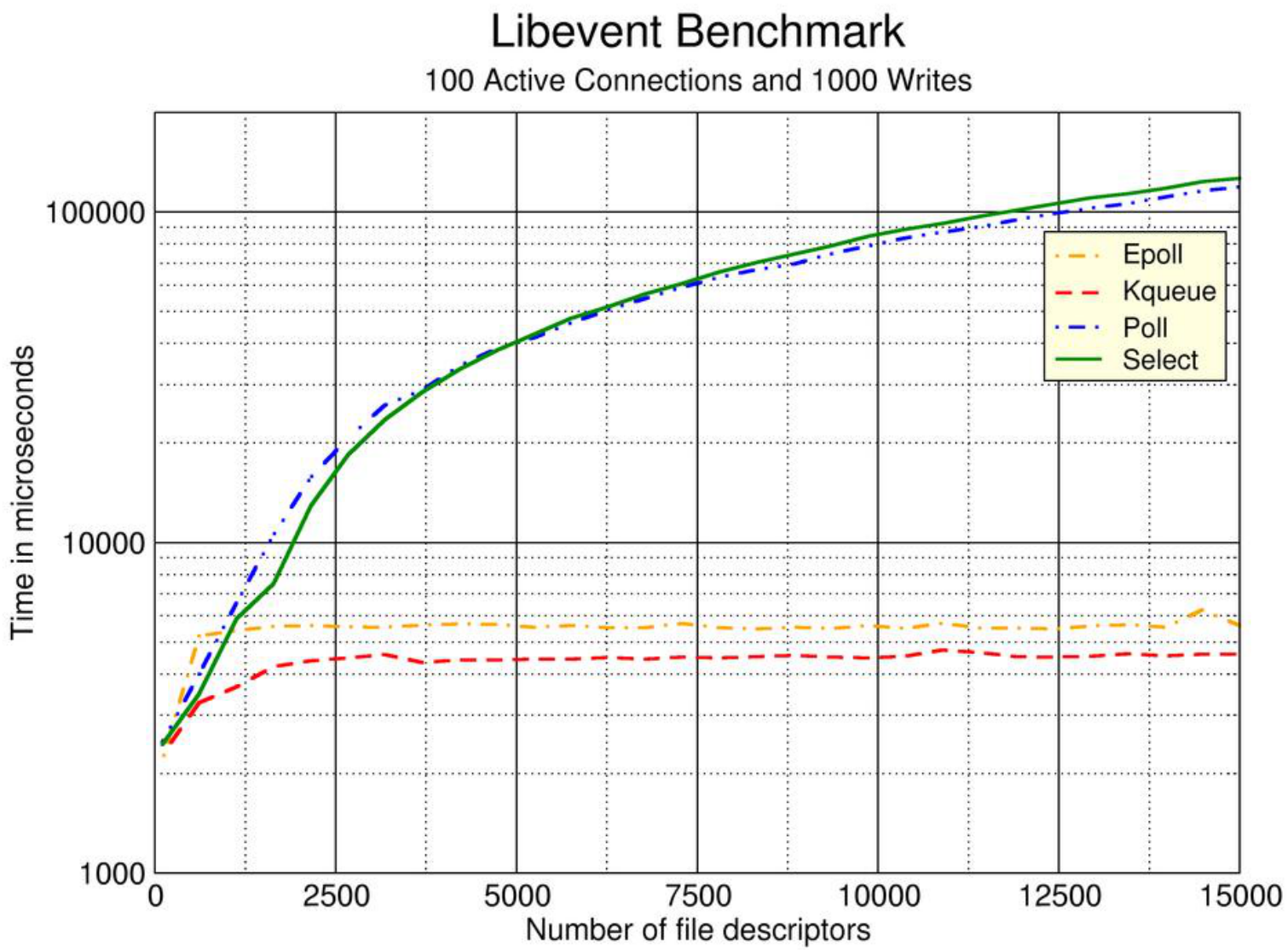
- BSD
  - kevent – náročnost stejná jako epoll, výhoda, že od svého vzniku podporuje přenos dalších typů událostí
- Solaris a další POSIX
  - /dev/poll
  - event ports
- Windows NT
  - IOCP – I/O Completion Port
  - Automatické řízení počtu aktivních vláken v threadpollu
- Java
  - Postupně implementovala tyto metody a zakrývá je pod java.nio.channels.SelectorProvider

Čas na obsluhu jednoho aktivního



zdroj: libevent - http://www.monkey.org/~provos/libevent/

Čas na obsluhu množství aktivních



zdroj: libevent - http://www.monkey.org/~provos/libevent/

Paralelní operace (nad soubory)

- Předchozí řešení jsou vhodná pro fronty nezávislých událostí
- **epoll** v kombinaci s více vlákny případně **I/O Completion Port** jsou pro tyto případy v podstatě optimálním řešením
- Velké databáze a paralelní práce nad velkými soubory vyžaduje zcela jiný přístup – i mmap() představuje problém, načítání (výpadek) stránky zcela blokuje daný thread, přitom zvyšování počtu threadů je řešení problematické (viz předchozí diskuze)
- Je potřeba přenést plánování I/O požadavků do user-space a zpracovávat data v tom pořadí, jak se je podaří načíst
- K tomu slouží služby pro **asynchronní I/O**, v Linuxu
  - standardizované **POSIX AIO**
  - nativní (pro Linux specifické) **libAIO**
  - Alternativa **mmap**() + **madvise**()

Asynchronní I/O – POSIX AIO (1)

- Požadavky popsané **struct aiocb** se vkládají do seznamu ke zpracování do určitého (nastaveného) limitu je jejich zpracování startováno okamžitě/paralelně
- Při dokončení požadavku může být procesu nebo threadu doručen asynchronní **signál**, thread může na dokončení alespoň jednoho ze skupiny požadavků čekat **aio_suspend**.

```
/* Asynchronous I/O control block.  */
struct aiocb
{
  int aio_fildes;         /* File desriptor.  */
  int aio_lio_opcode;     /* Operation to be performed.  */
     /* LIO_READ/LIO_WRITE/LIO_NOP */
  int aio_reqprio;        /* Request priority offset.  */
  volatile void *aio_buf;/* Location of buffer.  */
  size_t aio_nbytes;      /* Length of transfer.  */
  struct sigevent aio_sigevent;/* Signal number and value.  */

  /* Internal members. __error_code, __return_value/size */

  off_t aio_offset;       /* File offset.  */
};
```

Asynchronní I/O – POSIX AIO (2)

```
#include <aio.h>

int aio_read     (struct aiocb *aiocbp);
int aio_write    (struct aiocb *aiocbp);
int lio_listio   (int wait_mode, struct aiocb * const list[],
                  int nent, struct sigevent *notice);
int aio_cancel   (int fildes, struct aiocb *aiocbp);
int aio_suspend  (struct aiocb *const list[],
                  int nent, const struct timespec *timeout);
int aio_fsync    (int operation, struct aiocb *aiocbp);
  /* O_DSYNC/O_SYNC */
int aio_error    (const struct aiocb *aiocbp);
  /* EINPROGRESS/0(OK)/ENOSYS/EINVAL */
ssize_t aio_return (struct aiocb aiocbp); /* size/fsync ret */
```

- Současná implementace v **GLIBC** je založená na thread-pool(u) v user-space
- Toto řešení není výhodné a je snaha vytvořit pro Linux nativní podporu pro asynchronní operace v jádře – systémové volání **io_submit**

Nativní asynchronní I/O

```
#include <libaio.h>

struct iocb iocbq[MAX_PARALLEL]; int iocbq_used;
struct stat st;
io_context_t myctx;

fstat(srcfd, &st);
length = st.st_size;
memset(&myctx, 0, sizeof(myctx));
io_queue_init(aio_maxio, &myctx);
offset = 0;
/* Prepare memory for requests 0 .. iocbq_used-1*/
iocbq[i] = (struct iocb *) malloc(sizeof(struct iocb));
posix_memalign(&buf, alignment, iosize)
io_prep_pread(iocbq[i], -1, buf, iosize, offset);
offset += iosize;
io_set_callback(iocbq[i], rd_done);

rc = io_submit(myctx, iocbq_used, iocbq);
rc = io_queue_run(myctx); /* nonblocking */
rc = io_queue_wait(myctx, NULL /*&timeout*/);
```

Příjem dat ze sítě bez volání jádra

- Mapování příjmového a vysílacího bufferu přímo do adresního prostoru aplikace (PF_PACKET protocol family)

```
sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
int val = TPACKET_V2;
setsockopt(sock, SOL_PACKET, PACKET_VERSION, &val, sizeof(val));
strncpy(ifr.ifr_name, dev, sizeof(ifr.ifr_name);
ioctl(sock, SIOCGIFINDEX, &ifr);
addr.sll_family = AF_PACKET;
addr.sll_protocol = htons(ETH_P_CAN);
addr.sll_ifindex =  ifr.ifr_ifindex;
bind(sock, (struct sockaddr *)&addr, sizeof(struct sockaddr_ll));
struct tpacket_req req = {
     .tp_block_size = BLOCK_SIZE,
     .tp_frame_size = FRAME_SIZE,
     .tp_block_nr   = BLOCK_NR,
     .tp_frame_nr   = FRAME_NR,
};
setsockopt(sock, SOL_PACKET, PACKET_TX_RING, (char *)&req, sizeof(req));
tx_ring_buffer = (void*)mmap(0, BLOCK_SIZE*BLOCK_NR, PROT_READ|PROT_WRITE,
MAP_SHARED, sock, 0);
```

- Vlastní aktivace vysílání

```
send(sock, NULL, 0, 0)
```

Další možnosti

- Odeslání celého/části souboru bez jeho načtení/mapování do adresního prostoru procesu

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Odkazy na další informace

- Dan Kegel: The C10K problem
  http://www.kegel.com/c10k.html
- Nick Mathewson a Niels Provos
  http://www.monkey.org/~provos/libevent/
  /dev/poll, kqueue(2), event ports, select(2), poll(2), epoll(4), Win IOCP
- uLEvPoll Library - uLan/Universal Light Event Poll Library
  http://ulan.git.sourceforge.net/git/gitweb.cgi?p=ulan/ulevpoll
  http://ulan.sourceforge.net/pdf/ulevpoll.pdf