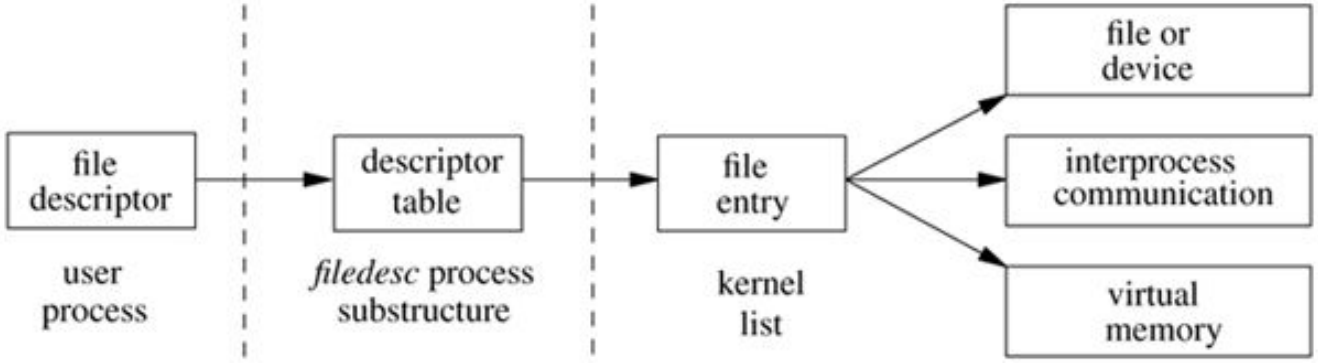
Struktura jądra UNIX

Wykład 14: Obsługa plików

Unix: wywołania systemowe

Każdy otwarty zasób plikowy ma **uchwyt** (ang. handle) zwany **deskryptorem pliku** (ang. file descriptor) czyli liczbę całkowitą ≥ 0.



Implementacja np. **`read`** odnajduje wpis w **tablicy deskryptorów plików** (`filedesc`) i skojarzony z nim **wpis pliku** (`file`). Każdy proces ma swoją tablicę deskryptorów, ale wpisy mogą być współdzielone. Typu pliku (`DTYPE_{VNODE, SOCKET, PIPE, ...}`) determinuje zestaw implementacji **operacji na pliku** (`fileops`).

FreeBSD: Typy plików

```
#define DTYPE_NONE        0   // not yet initialized
#define DTYPE_VNODE       1   // file or device
#define DTYPE_SOCKET      2   // communications endpoint
#define DTYPE_PIPE        3   // pipe
#define DTYPE_FIFO        4   // named pipe
#define DTYPE_KQUEUE      5   // event queue
#define DTYPE_CRYPTO      6   // cryptographic hardware
#define DTYPE_MQUEUE      7   // POSIX message queue
#define DTYPE_SHM         8   // POSIX shared memory
#define DTYPE_SEM         9   // POSIX semaphore
#define DTYPE_PTS         10  // pseudo-teletype master device
#define DTYPE_DEV         11  // device not referenced by a vnode
#define DTYPE_PROCDESC    12  // process
...
```

FreeBSD: Struktura pliku

```
struct file {
    void             *f_data;        // file descriptor specific data
    struct fileops   *f_ops;         // File operations
    struct ucred     *f_cred;        // Associated credentials
    struct vnode     *f_vnode;       // NULL or applicable vnode
    short             f_type;        // descriptor type
    ...
    volatile u_int    f_flag;        // see fcntl.h
    volatile u_int    f_count;       // reference count
    /* DTYPE_VNODE specific fields. */
    int               f_seqcount;    // (a) count of sequential accesses
    off_t             f_nextoff;     // next expected read/write offset
    ...
    /* DFLAG_SEEKABLE specific fields */
    off_t             f_offset;
    ...
};
```

FreeBSD: Otwieranie plików

Po utworzeniu struktury reprezentującej plik, jedną z poniższych procedur, jądro instaluje plik w tablicy deskryptorów.

- `vn_open`: pliki z przestrzeni nazw systemu plików → `vnops`
- `kern_socketpair`: gniazda → `socreate` + `socketops`
- `kern_pipe`: potoki → `pipe_create` + `pipeops`

Wszystkie operacje na plikach są przeprowadzane przez wspólny interfejs obiektów plikopodobnych. Wskaźnik na dane konkretnego pliku jest przechowywany w **`file::f_data`**.

Jeśli obiekt wspierający pliku zniknie (np. wyjęcie pendrive) to operacje na plikach zostaną zastąpione `badfileops`.

FreeBSD: Operacje na plikach

```
struct fileops {
    fo_rdwr_t       *fo_read;
    fo_rdwr_t       *fo_write;
    fo_truncate_t   *fo_truncate;
    fo_ioctl_t      *fo_ioctl;
    fo_poll_t       *fo_poll;
    fo_kqfilter_t   *fo_kqfilter;
    fo_stat_t       *fo_stat;
    fo_close_t      *fo_close;
    fo_chmod_t      *fo_chmod;
    fo_chown_t      *fo_chown;
    fo_sendfile_t   *fo_sendfile;    // kopiowanie plik -> gniazdo
    fo_seek_t       *fo_seek;
    fo_mmap_t       *fo_mmap;
    ...
    fo_flags_t       fo_flags;       // DFLAG_{PASSABLE,SEEKABLE}
};
```

Przykład: ustalanie rozmiaru pliku

Poniższe wywołanie systemowe:

```
int ftruncate(int fd, off_t length);
```

… jest obsługiwane w kern_ftruncate.

Najpierw na podstawie numeru deskryptora **fd** odnajdujemy strukturę pliku przy pomocy fget, by potem wywołać **file::f_fileops::fo_truncate**:

```
typedef int fo_truncate_t(struct file *, off_t,
        struct ucred *, struct thread *);
```

… z kredencjałami **ucred** wołającego wątku.

Do czego służą `fo_flags`?

`DFLAG_SEEKABLE` → plik o swobodnym dostępie, interfejs zarządza pozycją kursora pliku

`DFLAG_PASSABLE` → plik można przesyłać między procesami `cmsg(3)` przy pomocy gniazd domeny uniksowej

Bezpieczne kopiowanie danych

Wykorzystywane przez operacje **`fo_read`** i **`fo_write`**:

```
int uiomove(void *buf, int howmuch, struct uio *uiop);

struct uio {
  struct iovec * uio_iov;        // scatter/gather list
  int            uio_iovcnt;     // length of scatter/gather list
  off_t          uio_offset;     // offset in target object
  ssize_t        uio_resid;      // remaining bytes to copy
  enum uio_seg   uio_segflg;     // UIO_USERSPACE, UIO_SYSSPACE, ...
  enum uio_rw    uio_rw;         // UIO_READ, UIO_WRITE
  struct thread *uio_td;         // owner
};

struct iovec {
  void * iov_base;  // Base address
  size_t iov_len;   // Length
};
```

FreeBSD: Tablica deskryptorów plików (1)

```
struct filedesc {
  struct fdescenttbl *fd_files;  // open files table
  struct pwd *fd_pwd;            // directories
  u_long *fd_map;              // bitmap of free fds
  int fd_lastfile;             // high-water mark of fd_ofiles
  int fd_freefile;             // approx. next free file
  u_short fd_cmask;            // mask for file creation
  int fd_refcnt;               // thread reference count
  int fd_holdcnt;              // hold count on structure + mutex
  struct sx fd_sx;             // protects members of this struct
  struct kqlist fd_kqlist;     // list of kqueues on this filedesc
  int fd_holdleaderscount;     // block fdfree() for shared close()
  int fd_holdleaderswakeup;    // fdfree() needs wakeup
};
```

Zarządzanie **`fd_files`** niestety nudne → zwykłe realloc(9).

FreeBSD: Tablica deskryptorów plików (2)

```
struct fdescenttbl {
  int                 fdt_nfiles;     // num of open files allocated
  struct filedescent fdt_ofiles[0];  // open files
};

struct filedescent {
  struct file      *fde_file;         // file structure for open file
  struct filecaps   fde_caps;         // per-descriptor rights
  uint8_t           fde_flags;        // UF_EXCLOSE (FD_CLOEXEC)
  seqc_t            fde_seqc;         // keep file and caps in sync
};

struct pwd {
  volatile u_int pwd_refcount;
  struct vnode *pwd_cdir;             // current directory
  struct vnode *pwd_rdir;             // root directory
  struct vnode *pwd_jdir;             // jail root directory
};
```

`filedesc`: ciekawsze procedury interfejsu

`fget`: pobierz plik stojący za deskryptorem

`fdalloc`: przydziel nowy deskryptor

`finstall`: j.w. plus zainstaluj tam określony plik

`fdgrowtable`: zwiększ tablice deskryptorów (domyślnie 2x)

`fdfree`: zwolnij dany numer deskryptora i odpowiadający plik

`fdcopy`: kopiuje tablicę deskryptorów (`fork`)

`fdcloseexec`: zamyka oznaczone deskryptory (`execve`)

`dupfdopen`: klonuje deskryptor do nowego deskryptora (`dup`)

Inne wywołania systemowe

`umask`: maska bitów uprawnień nowo tworzonego pliku, przechowywana w **`filedesc::fd_cmask`**

`chdir`: bieżący katalog roboczy ustawiany przez procedurę `pwd_chdir`, przechowywany w **`filedesc::fd_pwd::pwd_cdir`**.

Multipleksowane wejście–wyjście: motywacja

Proces komunikatora ustala połączenie z serwerem. Jeśli czeka na wejście od użytkownika to nie może czekać na komunikaty z serwera i vice versa. Jak to naprawić?

- **nonblocking I/O**: proces w pętli sprawdza czy *FDs* gotowe do użycia → zbędnie zużywa czas CPU,
- **signal-driven I/O**: jeśli na deskryptorze pojawiły się dane wyślij SIGIO, niepraktyczne dla dużej liczby deskryptorów!
- **polling I/O**: usypiamy proces do momentu pojawienia się zdarzeń na przekazanej grupie deskryptorów
- **kernel-event polling**: optymalizacja wcześniejszego

Multipleksowane wejście–wyjście: `select` i `poll`

Ustawiamy flagi deskryptory plików na `O_NONBLOCK`.

Wywołania `read` / `write` zwracają `EAGAIN` jeśli bufor pusty albo odpowiednio pełny. Jeśli nie można kontynuować pracy to wołamy `select` albo `poll`.

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

int poll(struct pollfd fds[], nfds_t nfds,
         int timeout);
```

Problemy z `select` i `poll`

Są bezstanowe → za każdym razem jądro musi zbudować dużą strukturę danych do śledzenia zdarzeń!

```
struct pollfd {
  int   fd;       /* file descriptor */
  short events;   /* events to look for */
  short revents;  /* events returned */
};
```

Programista nie wie ile bajtów może przeczytać lub zapisać bez otrzymania `EAGAIN`! A co jeśli chcemy czekać na inne zdarzenia: sygnały, czasomierze, modyfikacje sys. plików?

Problem: Tysiące połączeń na sekundę → nie skaluje się!

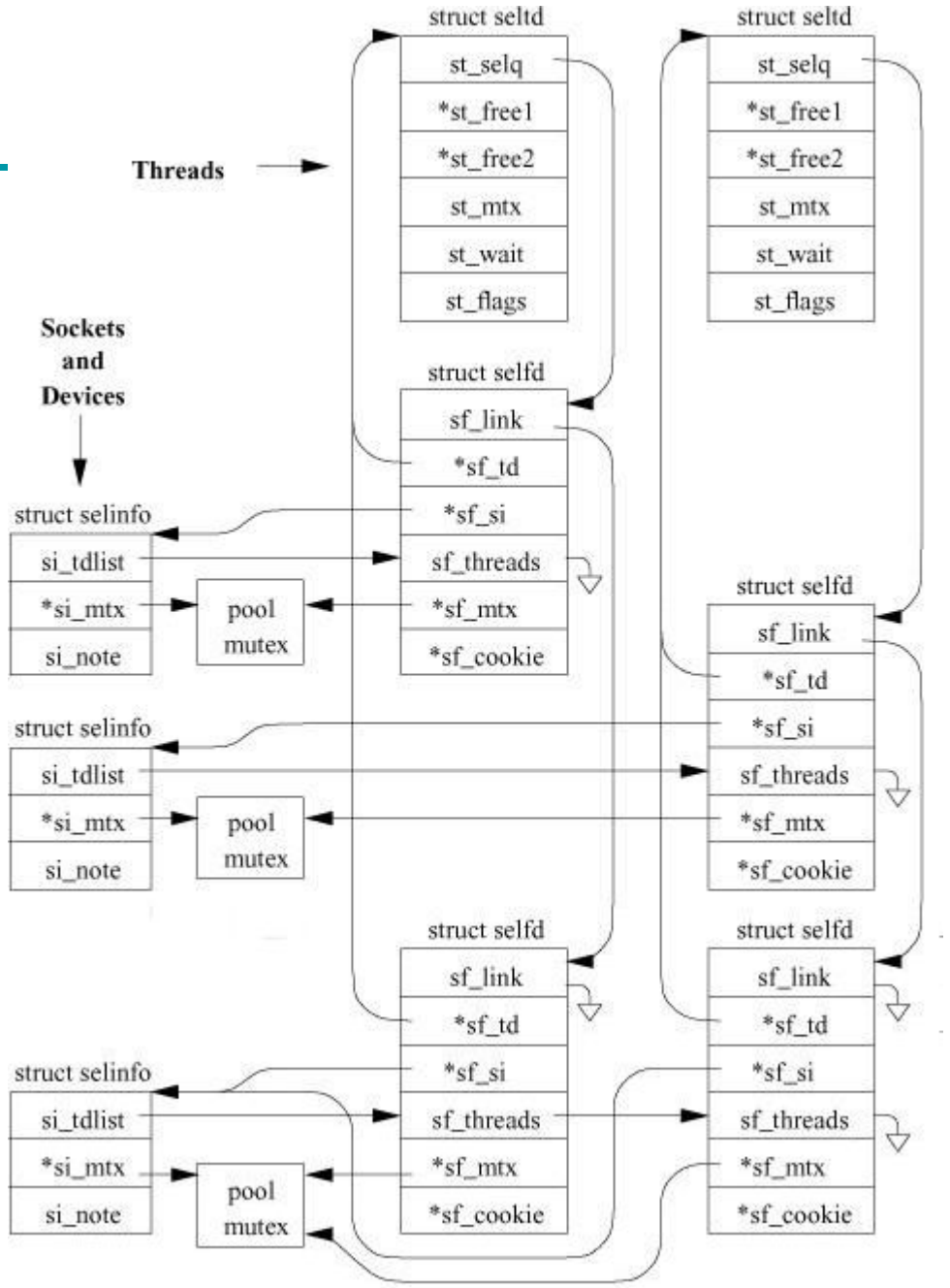
Struktura dla `select(2)`

Struktura budowana
w trakcie obsługi wywołania systemowego. Dla poll(2) analogiczne zachowanie.

`seltd`: instancja wywołania

`selfd`: referencja do pliku związana z wywołaniem

`selinfo`: opis zdarzenia



`seltd`: stan wątku w trakcie obsługi `select(2)`

`kern_select` tworzy poniższą strukturę przy pomocy `seltdinit` i podczepia ją pod `thread::td_sel`. Skanując zbiór deskryptorów `selscan` buduje strukturę z poprzedniego slajdu. Jeśli znaleziono aktywny deskryptor to kończymy. Jeśli nie, to idziemy spać w `seltdwait`, a po wybudzeniu skanujemy ponownie `selrescan`. Jeśli znaleziono to demontujemy strukturę w `seltdclear`.

```
struct seltd {
  STAILQ_HEAD(, selfd) st_selq;    // (k) List of selfds
  ...
  struct mtx           st_mtx;     // Protects struct seltd
  struct cv            st_wait;    // (t) Wait channel
  int                  st_flags;   // (t) SELTD_* flags
};
```

Powiadomienie o zdarzeniu

Każdy obiekt, na którym robimy **`select`** ma skojarzoną strukturę:

```
struct selinfo {
  struct selfdlist si_tdlist;    // List of sleeping threads
  struct knlist    si_note;      // kernel note list
  struct mtx      *si_mtx;       // Lock for si_tdlist
};
```

Po odebraniu pakietu na gnieździe wołana procedura `sowakeup`, która woła `selwakeup`. Ta przechodzi po wszystkich **`seltd`** na liście **`si_tdlist`**. W **`st_flags`** ustawia flagę **`PENDING`** i wybudza wątki śpiące na **`st_wait`**. Wątki w trakcie `selscan` mogły nie załapać się na zauważenie zmiany stanu deskryptora, ale nie pójdą spać w `seltdwait`, bo zauważą flagę **`PENDING`**.

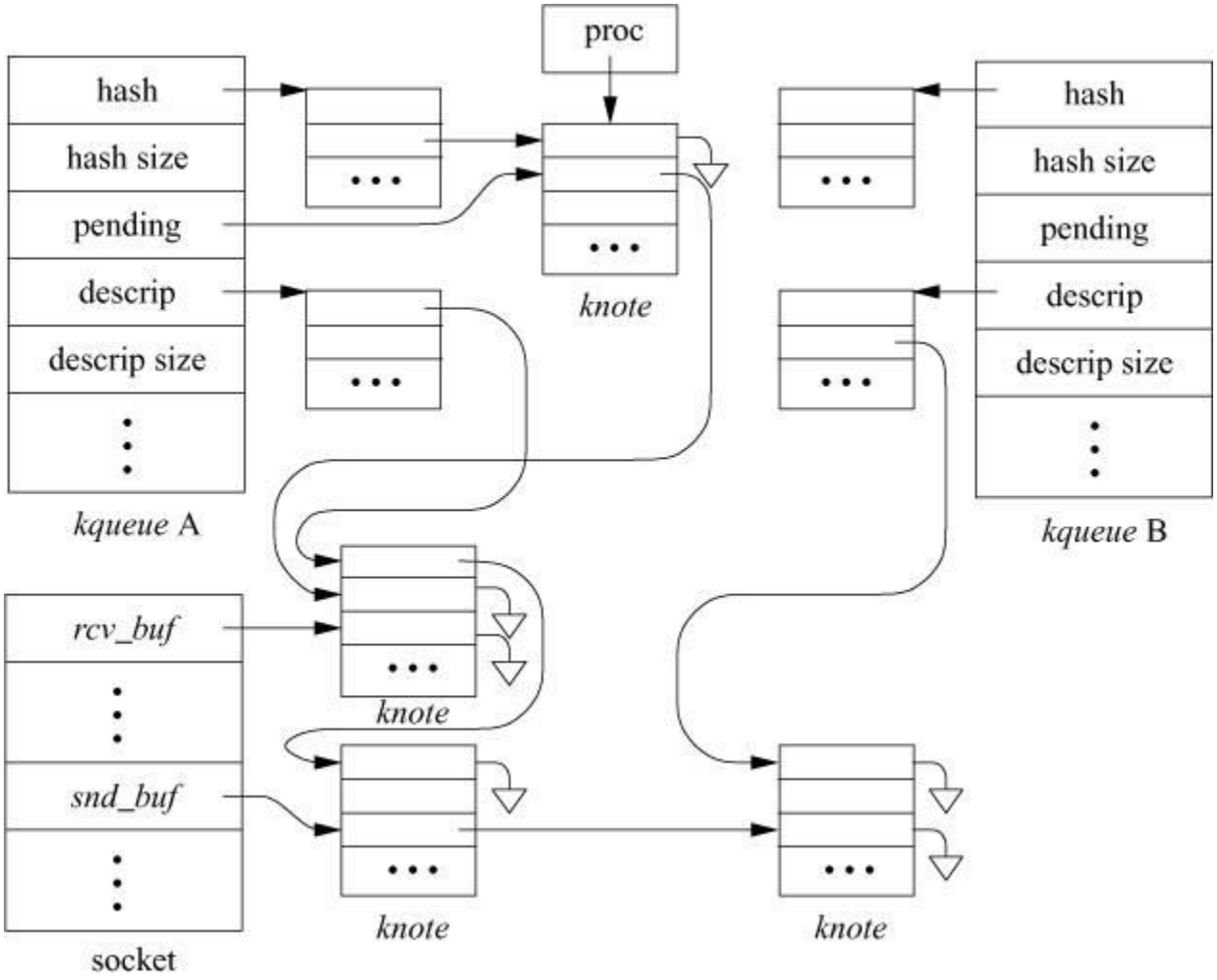
kqueue i kevent: monitorowanie zdarzeń jądra

Kqueue to obiekt jądra osiągalny przez deskryptor pliku.
Będziemy z nim wiązać stan, tj. nasłuchiwane zdarzenia,
a potem czekać na jego modyfikację.

Event name	Operation tracked
EVFILT_READ	Descriptor has data to read
EVFILT_WRITE	Descriptor has buffer space to write
EVFILT_AIO	Asynchronous I/O associated with descriptor has completed
EVFILT_VNODE	Information associated with a file has changed
EVFILT_PROC	Status of a process has changed
EVFILT_SIGNAL	A signal has been posted for a process
EVFILT_TIMER	An event-based timer has expired
EVFILT_USER	Application defined and triggered events

```
int kevent(int kq, const struct kevent *changelist, int nchanges,
                   struct kevent *eventlist, int nevents,
                   const struct timespec *timeout);
```

FreeBSD: Struktury **kqueue** i **knote**



Przetwarzanie zdarzenia

Dla każdego zdarzenia na obiekcie zdefiniowana poniższa struktura, np. soread_filtops, sig_filtops, proc_filtops.

```
struct filterops {
  int f_isfd;
  int (*f_attach)(struct knote *kn);
  void (*f_detach)(struct knote *kn);
  int (*f_event)(struct knote *kn, long hint);
  void (*f_touch)(struct knote *kn, struct kevent *kev, u_long type);
};
```

Odpowiednia procedura (np. sowakeup, exit1, tdsendsignal) wywołuje knote przegląda wszystkie liściki przyczepione do danego obiektu odpytując czy dane zdarzenie **f_event** zaszło.

Pytania?