Bahir Dar Institute of Technology

Faculty of Computing

Department of Software Engineering

Operating System and System Programming

Individual Assignment

System Call

By Netsanet Alemu: BDU1311647

Submitted to: Teacher Wendimu Baye

Submission Date: 17/11/2014 E.C

A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

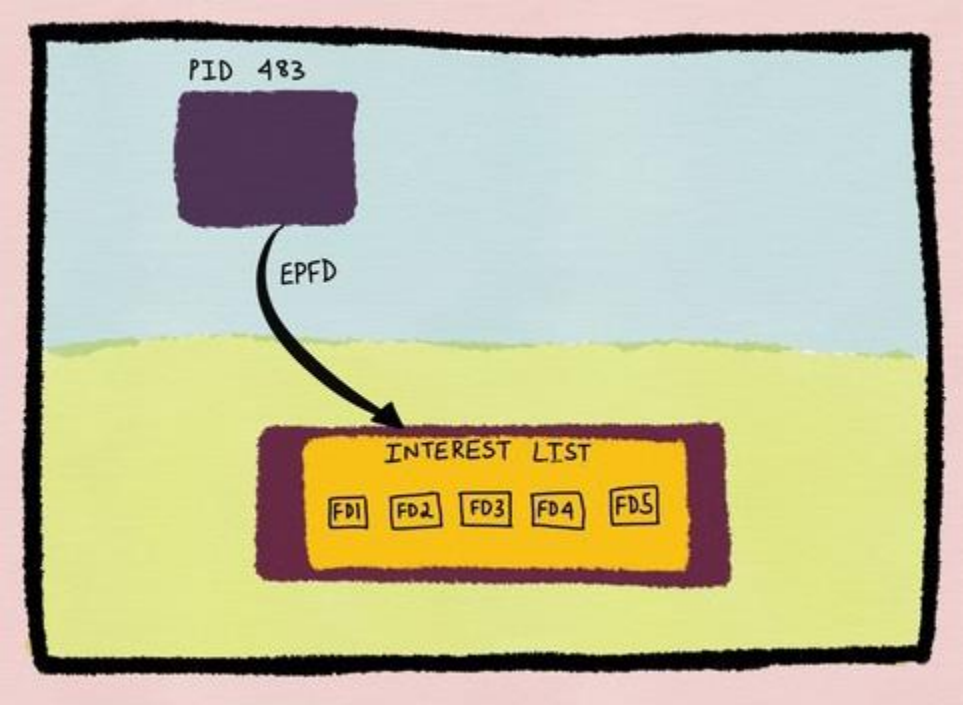
System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.

Given System call:

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)

epoll stands for event poll and is a Linux specific construct. It allows for a process to monitor multiple file descriptors and get notifications when I/O is possible on them. It allows for both edge-triggered as well as level-triggered notifications.

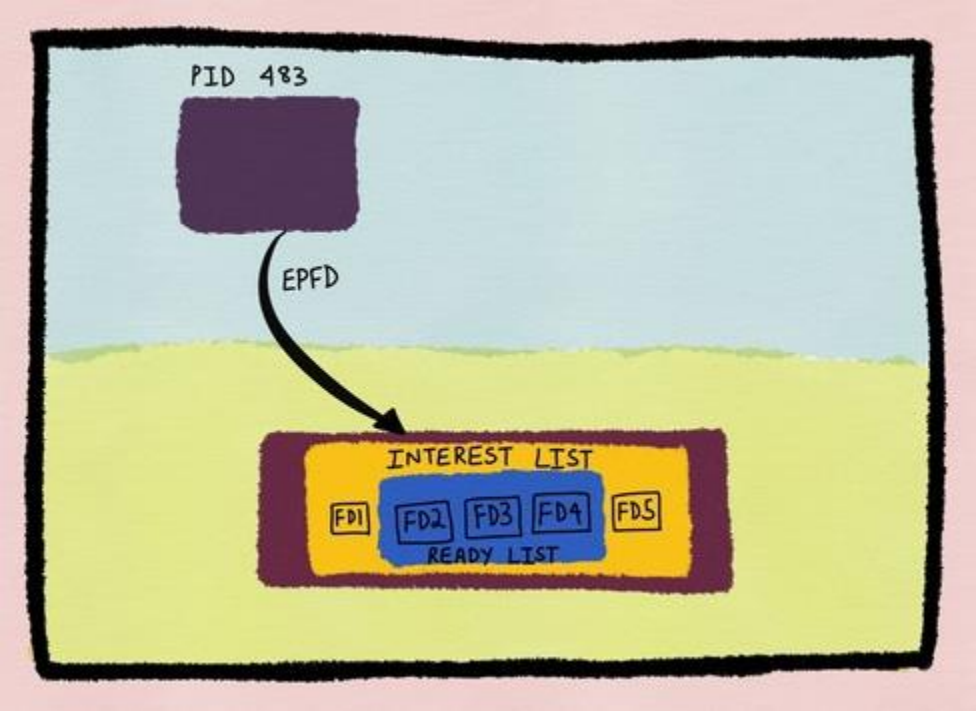
A process can add file descriptors it wants monitored to the *epoll* instance by calling epoll_ctl. All the file descriptors registered with an *epoll* instance are collectively called an ***epoll set*** or the ***interest list***.



In the above diagram, process 483 has registered file descriptors ***fd1***, ***fd2***, ***fd3***, ***fd4*** and ***fd5*** with the *epoll* instance. This is the ***interest list*** or the ***epoll set*** of that particular *epoll* instance.

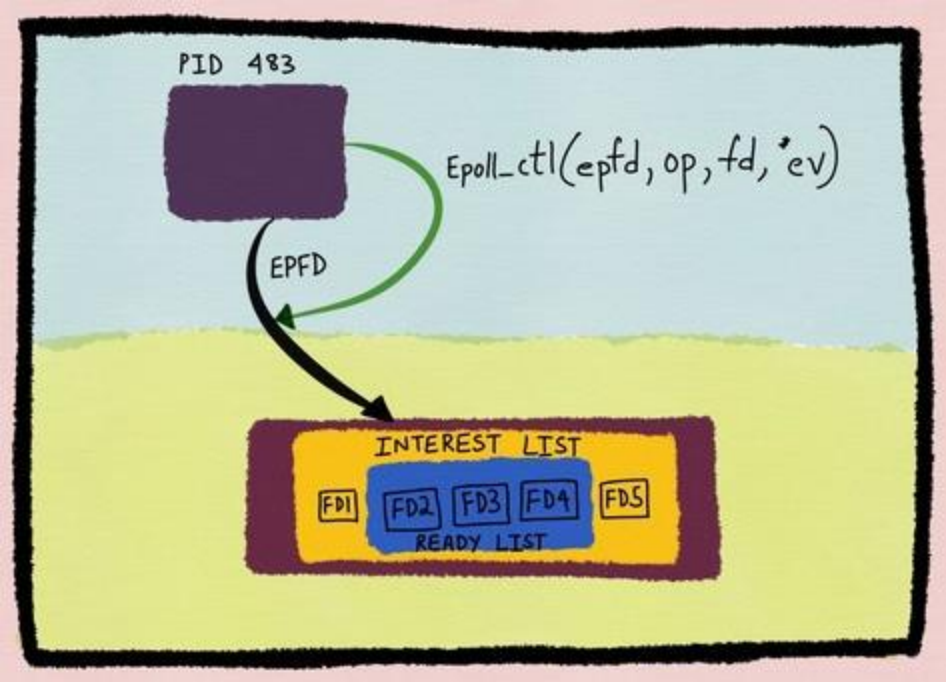
Subsequently, when any of the file descriptors registered become ready for I/O, then they are considered to be in the ***ready list***.

The ***ready list*** is a subset of the ***interest list***.



The signature of the epoll_ctl system call is as follows:

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```



***epfd* —** is the file descriptor returned by epoll_create which identifies the *epoll* instance in the kernel.

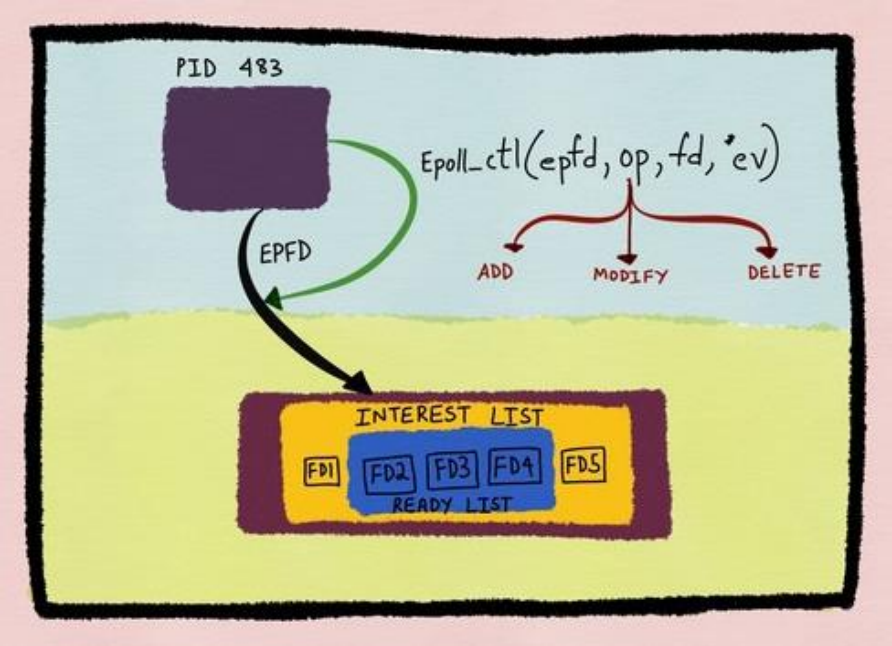
fd* —** is the file descriptor we want to add to the ***epoll list/***interest list***.

***op* —** refers to the operation to be performed on the file descriptor *fd*. In general, three operations are supported:

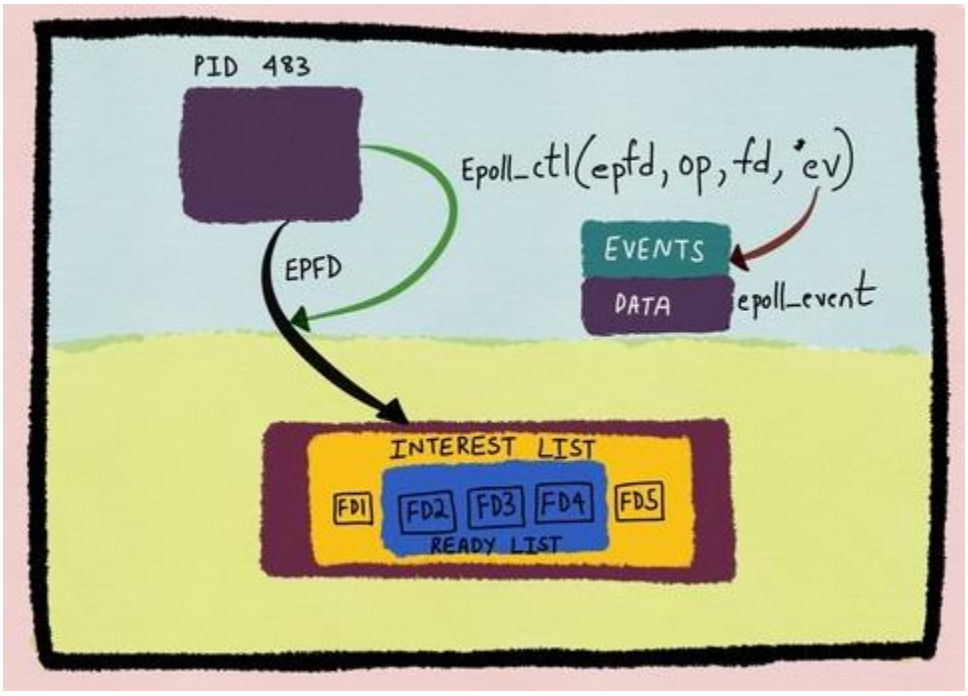
- **Register *fd*** with the *epoll* instance (**EPOLL_CTL_ADD**) and get notified about events that occur on ***fd***

- **Delete**/deregister ***fd*** from the *epoll* instance. This would mean that the process would no longer get any notifications about events on that file descriptor (**EPOLL_CTL_DEL**). If a file descriptor has been added to multiple *epoll* instances, then closing it will remove it from all of the *epoll* interest lists to which it was added.

- **Modify** the events ***fd*** is monitoring (**EPOLL_CTL_MOD**)



event* —** is a pointer to a structure called ***epoll_event which stores the *event* we actually want to monitor ***fd*** for.



The *event* argument describes the object linked to the file descriptor *fd*. The *struct epoll_event* is defined as:

```
typedef union epoll_data {
   void        *ptr;
```

```
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;

struct epoll_event {
    uint32_t    events;      /* Epoll events */
    epoll_data_t data;       /* User data variable */
};
```

The *data* member of the *epoll_event* structure specifies data that the kernel should save and then return (via epoll_wait(2)) when this file descriptor becomes ready.
The *events* member of the *epoll_event* structure is a bit mask composed by ORing together zero or more of the following available event types:

- **EPOLLIN:** indicates that the corresponding file descriptor is readable (including that the opposite socket is normally closed)
- **EPOLLOUT:** indicates that the corresponding file descriptor is writable;
- **EPOLLPRI**: indicates that the corresponding file descriptor has urgent data readability (this should indicate the arrival of out of band data)
- **EPOLLERR**: indicates that the corresponding file descriptor has an error;
- **EPOLLHUP**: indicates that the corresponding file descriptor is hung up;
- **EPOLLLET**: set epoll to edge trigger (ET) mode;
- **EPOLLONESHOT:** only listen to the event once. After listening to the event, if you need to continue listening to the socket, you need to add the socket to the epoll queue again.
- **EPOLLWAKEUP:** If **EPOLLONESHOT** and **EPOLLET** are clear and the process has the **CAP_BLOCK_SUSPEND** capability, ensure that the system does not enter "suspend" or "hibernate" while this event is pending or being processed.
- **EPOLLEXCLUSIVE:** Sets an exclusive wakeup mode for the epoll file descriptor that is being attached to the target file descriptor, *fd*.

Code implementation of epoll

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/epoll.h>
#include <errno.h>

#define MAXEVENTS 64

static int
make_socket_non_blocking (int sfd)
{
```

```
  int flags, s;

  flags = fcntl (sfd, F_GETFL, 0);
  if (flags == -1)
    {
      perror ("fcntl");
      return -1;
    }

  flags |= O_NONBLOCK;
  s = fcntl (sfd, F_SETFL, flags);
  if (s == -1)
    {
      perror ("fcntl");
      return -1;
    }

  return 0;
}

static int
create_and_bind (char *port)
{
  struct addrinfo hints;
  struct addrinfo *result, *rp;
  int s, sfd;

  memset (&hints, 0, sizeof (struct addrinfo));
  hints.ai_family = AF_UNSPEC;     /* Return IPv4 and IPv6 choices */
  hints.ai_socktype = SOCK_STREAM; /* We want a TCP socket */
  hints.ai_flags = AI_PASSIVE;     /* All interfaces */

  s = getaddrinfo (NULL, port, &hints, &result);
  if (s != 0)
    {
      fprintf (stderr, "getaddrinfo: %s\n", gai_strerror (s));
      return -1;
    }

  for (rp = result; rp != NULL; rp = rp->ai_next)
    {
      sfd = socket (rp->ai_family, rp->ai_socktype, rp->ai_protocol);
      if (sfd == -1)
        continue;

      s = bind (sfd, rp->ai_addr, rp->ai_addrlen);
      if (s == 0)
        {
          /* We managed to bind successfully! */
          break;
        }

      close (sfd);
    }

  if (rp == NULL)
    {
```

```
      fprintf (stderr, "Could not bind\n");
      return -1;
    }

  freeaddrinfo (result);

  return sfd;
}

int
main (int argc, char *argv[])
{
  int sfd, s;
  int efd;
  struct epoll_event event;
  struct epoll_event *events;

  if (argc != 2)
    {
      fprintf (stderr, "Usage: %s [port]\n", argv[0]);
      exit (EXIT_FAILURE);
    }

  sfd = create_and_bind (argv[1]);
  if (sfd == -1)
    abort ();

  s = make_socket_non_blocking (sfd);
  if (s == -1)
    abort ();

  s = listen (sfd, SOMAXCONN);
  if (s == -1)
    {
      perror ("listen");
      abort ();
    }

  efd = epoll_create1 (0);
  if (efd == -1)
    {
      perror ("epoll_create");
      abort ();
    }

  event.data.fd = sfd;
  event.events = EPOLLIN | EPOLLET;
  s = epoll_ctl (efd, EPOLL_CTL_ADD, sfd, &event);
  if (s == -1)
    {
      perror ("epoll_ctl");
      abort ();
    }

  /* Buffer where events are returned */
  events = calloc (MAXEVENTS, sizeof event);
```

```
/* The event loop */
while (1)
  {
    int n, i;

    n = epoll_wait (efd, events, MAXEVENTS, -1);
    for (i = 0; i < n; i++)
      {
        if ((events[i].events & EPOLLERR) ||
            (events[i].events & EPOLLHUP) ||
            (!(events[i].events & EPOLLIN)))
          {
            /* An error has occured on this fd, or the socket is not
               ready for reading (why were we notified then?) */
            fprintf (stderr, "epoll error\n");
            close (events[i].data.fd);
            continue;
          }

        else if (sfd == events[i].data.fd)
          {
            /* We have a notification on the listening socket, which
               means one or more incoming connections. */
            while (1)
              {
                struct sockaddr in_addr;
                socklen_t in_len;
                int infd;
                char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

                in_len = sizeof in_addr;
                infd = accept (sfd, &in_addr, &in_len);
                if (infd == -1)
                  {
                    if ((errno == EAGAIN) ||
                        (errno == EWOULDBLOCK))
                      {
                        /* We have processed all incoming
                           connections. */
                        break;
                      }
                    else
                      {
                        perror ("accept");
                        break;
                      }
                  }

                s = getnameinfo (&in_addr, in_len,
                                 hbuf, sizeof hbuf,
                                 sbuf, sizeof sbuf,
                                 NI_NUMERICHOST | NI_NUMERICSERV);
                if (s == 0)
                  {
                    printf("Accepted connection on descriptor %d "
                           "(host=%s, port=%s)\n", infd, hbuf, sbuf);
                  }
```

```
          /* Make the incoming socket non-blocking and add it to the
             list of fds to monitor. */
          s = make_socket_non_blocking (infd);
          if (s == -1)
            abort ();

          event.data.fd = infd;
          event.events = EPOLLIN | EPOLLET;
          s = epoll_ctl (efd, EPOLL_CTL_ADD, infd, &event);
          if (s == -1)
            {
              perror ("epoll_ctl");
              abort ();
            }
        }
      continue;
    }
  else
    {
      /* We have data on the fd waiting to be read. Read and
         display it. We must read whatever data is available
         completely, as we are running in edge-triggered mode
         and won't get a notification again for the same
         data. */
      int done = 0;

      while (1)
        {
          ssize_t count;
          char buf[512];

          count = read (events[i].data.fd, buf, sizeof buf);
          if (count == -1)
            {
              /* If errno == EAGAIN, that means we have read all
                 data. So go back to the main loop. */
              if (errno != EAGAIN)
                {
                  perror ("read");
                  done = 1;
                }
              break;
            }
          else if (count == 0)
            {
              /* End of file. The remote has closed the
                 connection. */
              done = 1;
              break;
            }

          /* Write the buffer to standard output */
          s = write (1, buf, count);
          if (s == -1)
            {
              perror ("write");
```

```
                    abort ();
                  }
              }

            if (done)
              {
                printf ("Closed connection on descriptor %d\n",
                        events[i].data.fd);

                /* Closing the descriptor will make epoll remove it
                   from the set of descriptors which are monitored. */
                close (events[i].data.fd);
              }
          }
      }
  }

  free (events);

  close (sfd);

  return EXIT_SUCCESS;
}
```

References

- ✓ Using epoll? Might wanna think about batching epoll_ctl - Granulate
- ✓ System Call in OS (Operating System): What is, Types and Examples (guru99.com)
- ✓ The method to epoll's madness. My previous post covered the… | by Cindy Sridharan | Medium
- ✓ Analysis of epoll function | Develop Paper
- ✓ c-example/epoll-example.c at master · millken/c-example · GitHub
- ✓ epoll_ctl(2) - Linux manual page (man7.org)