# Signals and Event-Based I/O

CSE 536 Spring 2024

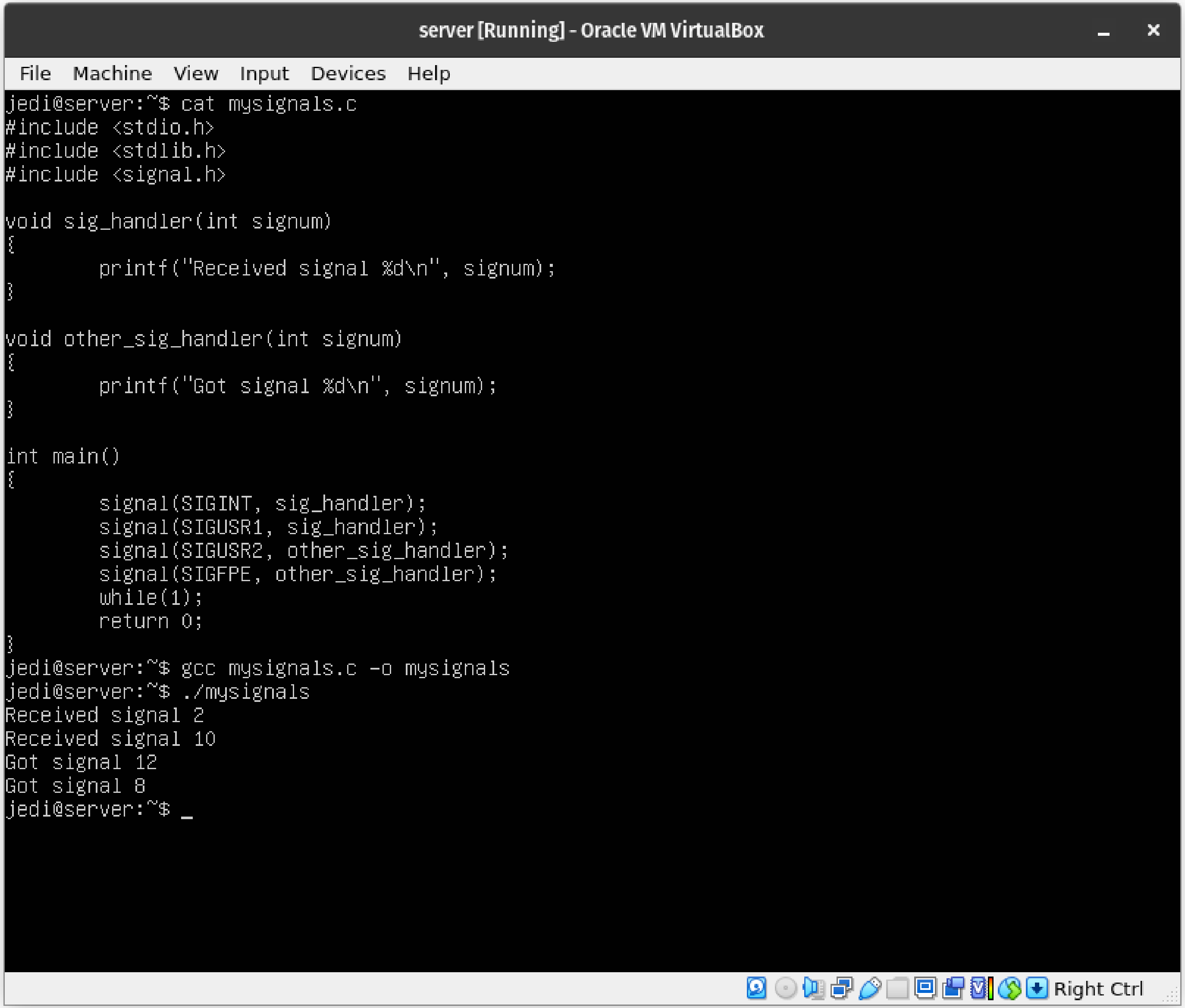
jedimaestro@asu.edu

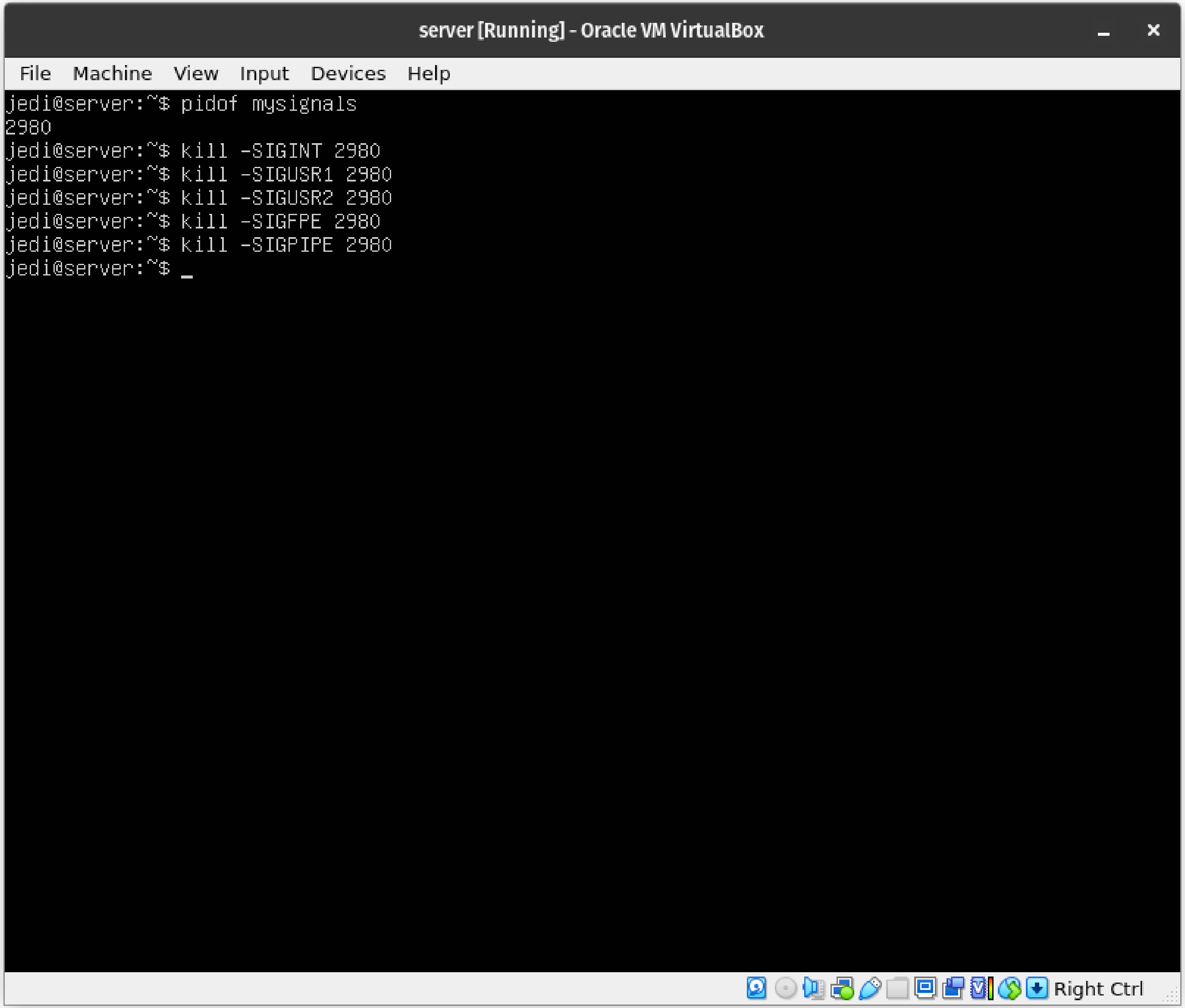
# Why Event-Based I/O?

- Multithreading can lead to a lot of errors, complexity
- Blocking is bad for performance
  - Blocking means your process is put in a wait queue because of a system call you made, basically

# Outline

- UNIX signals
- poll and ppoll()
- select() and pselect()
- epoll()
- kqueue()


server [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
jedi@server:~$ cat mysignals.c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void sig_handler(int signum)
{
printf("Received signal %d\n", signum);
}
void other_sig_handler(int signum)
{
printf("Got signal %d\n", signum);
}
int main()
{
signal(SIGINT, sig_handler);
signal(SIGUSR1, sig_handler);
signal(SIGUSR2, other_sig_handler);
signal(SIGFPE, other_sig_handler);
while(1);
return 0;
}
jedi@server:~$ gcc mysignals.c -o mysignals
jedi@server:~$ ./mysignals
Received signal 2
Received signal 10
Got signal 12
Got signal 8
jedi@server:~$ _
Right Ctrl


server [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
jedi@server:~$ pidof mysignals
2980
jedi@server:~$ kill -SIGINT 2980
jedi@server:~$ kill -SIGUSR1 2980
jedi@server:~$ kill -SIGUSR2 2980
jedi@server:~$ kill -SIGFPE 2980
jedi@server:~$ kill -SIGPIPE 2980
jedi@server:~$ _
Right Ctrl

# poll()

- Wait for one or more file descriptors to become ready for use
- Positives
  - POSIX (Portable Operating System Interface, from IEEE)
    - can be used on Linux, BSD flavors, *etc.*
- Negatives
  - Does not scale to many file descriptors

https://github.com/raoulmillais/linux-system-programming/blob/master/src/poll-example.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>

#define TIMEOUT 5

int main (void)
{
        struct pollfd fds[2];
        int ret;

        /* watch stdin for input */
        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;

        /* watch stdout for ability to write */
        fds[1].fd = STDOUT_FILENO;
        fds[1].events = POLLOUT;

        ret = poll(fds, 2, TIMEOUT * 1000);
```

# ppoll()

- A race condition can occur if there are any signal handlers registered, ppoll() atomically handles signals, applies a sigmask, and saves new incoming signals to the end
- More details below in description of pselect()

# select()

- Like poll(), but older and clunkier

jedi@tortuga: ~

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

       void FD_CLR(int fd, fd_set *set);
       int  FD_ISSET(int fd, fd_set *set);
       void FD_SET(int fd, fd_set *set);
       void FD_ZERO(fd_set *set);

       int pselect(int nfds, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, const struct timespec *timeout,
                   const sigset_t *sigmask);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       pselect(): _POSIX_C_SOURCE >= 200112L

DESCRIPTION
       select() allows a program to monitor multiple file descriptors, waiting
       until one or more of the file descriptors become "ready" for some class
       of I/O operation (e.g., input possible).  A file descriptor is  consid-
       ered  ready  if it is possible to perform a corresponding I/O operation
       (e.g., read(2), or a sufficiently small write(2)) without blocking.

       select() can monitor only file descriptors numbers that are  less  than
       FD_SETSIZE;  poll(2)  and  epoll(7)  do  not have this limitation.  See
       BUGS.
 Manual page select(2) line 9 (press h for help or q to quit)
```

jedi@tortuga:~

```
pselect()
       The  pselect()  system  call allows an application to safely wait until
       either a file descriptor becomes ready or until a signal is caught.

       The operation of select() and pselect() is identical, other than  these
       three differences:

       • select()  uses  a  timeout that is a struct timeval (with seconds and
         microseconds), while pselect() uses a struct timespec  (with  seconds
         and nanoseconds).

       • select()  may  update  the timeout argument to indicate how much time
         was left.  pselect() does not change this argument.

       • select() has no sigmask argument, and  behaves  as  pselect()  called
         with NULL sigmask.

       sigmask  is  a  pointer to a signal mask (see sigprocmask(2)); if it is
       not NULL, then pselect() first replaces the current signal mask by  the
       one  pointed  to  by sigmask, then does the "select" function, and then
       restores the original signal mask.  (If sigmask  is  NULL,  the  signal
       mask is not modified during the pselect() call.)

       Other than the difference in the precision of the timeout argument, the
       following pselect() call:

           ready = pselect(nfds, &readfds, &writefds, &exceptfds,
 Manual page select(2) line 132 (press h for help or q to quit)
```

jedi@tortuga: ~

Other than the difference in the precision of the timeout argument, the following **pselect**() call:

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds,
                timeout, &sigmask);
```

is equivalent to atomically executing the following calls:

```
sigset_t origmask;

pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

The reason that **pselect**() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of **select**() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, **pselect**() allows one to first block signals, handle the signals that have come in, then call **pselect**() with the desired sigmask, avoiding the race.)

**The timeout**

The timeout argument for **select**() is a structure of the following type:

Manual page select(2) line 154 (press h for help or q to quit)

# epoll()

- Negatives
  - Not POSIX, Linux-specific
  - Slightly more complex to use than poll()
- Positives

| Number of File Descriptors | poll() CPU time | select() CPU time | epoll() CPU time |
| --- | --- | --- | --- |
| 10 | 0.61 | 0.73 | 0.41 |
| 100 | 2.9 | 3 | 0.42 |
| 1000 | 35 | 35 | 0.53 |
| 10000 | 990 | 930 | 0.66 |

The Linux Programming Interface, section 63.4.5
https://suchprogramming.com/epoll-in-3-easy-steps/

Examples from
https://suchprogramming.com/epoll-in-3-easy-steps/
…

## Step 1: Create epoll file descriptor

First I'll go through the process of just creating and closing an epoll instance.

```
#include <stdio.h>      // for fprintf()
#include <unistd.h>     // for close()
#include <sys/epoll.h> // for epoll_create1()

int main()
{
        int epoll_fd = epoll_create1(0);

        if (epoll_fd == -1) {
                fprintf(stderr, "Failed to create epoll file descriptor\n");
                return 1;
        }

        if (close(epoll_fd)) {
                fprintf(stderr, "Failed to close epoll file descriptor\n");
                return 1;
        }

        return 0;
}
```

## Step 2: Add file descriptors for epoll to watch

The next thing to do is tell epoll what file descriptors to watch and what kinds of events to watch for. In this example I'll use one of my favorite file descriptors in Linux, good ol' file descriptor `0` (also known as Standard Input).

```
#include <stdio.h>     // for fprintf()
#include <unistd.h>    // for close()
#include <sys/epoll.h> // for epoll_create1(), epoll_ctl(), struct epoll_event

int main()
{
	struct epoll_event event;
	int epoll_fd = epoll_create1(0);

	if (epoll_fd == -1) {
		fprintf(stderr, "Failed to create epoll file descriptor\n");
		return 1;
	}

	event.events = EPOLLIN;
	event.data.fd = 0;

	if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0, &event)) {
		fprintf(stderr, "Failed to add file descriptor to epoll\n");
		close(epoll_fd);
		return 1;
	}

	if (close(epoll_fd)) {
		fprintf(stderr, "Failed to close epoll file descriptor\n");
		return 1;
	}
	return 0;

}
```

## Step 3: Profit

That's right! We're almost there. Now let epoll do it's magic.

```
while (running) {
        printf("\nPolling for input...\n");
        event_count = epoll_wait(epoll_fd, events, MAX_EVENTS, 30000);
        printf("%d ready events\n", event_count);
        for (i = 0; i < event_count; i++) {
                printf("Reading file descriptor '%d' -- ", events[i].data.fd
                bytes_read = read(events[i].data.fd, read_buffer, READ_SIZE)
                printf("%zd bytes read.\n", bytes_read);
                read_buffer[bytes_read] = '\0';
                printf("Read '%s'\n", read_buffer);

                if(!strncmp(read_buffer, "stop\n", 5))
                running = 0;
        }
}
```

# Why is epoll() faster?

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

**Red–black tree**

| | | |
|---|---|---|
| **Type** | Tree | |
| **Invented** | 1978 | |
| **Invented by** | Leonidas J. Guibas and Robert Sedgewick | |
| **Complexities in big O notation** | | |
| | **Space complexity** | |
| **Space** | $O(n)$ | |
| | **Time complexity** | |
| **Function** | **Amortized** | **Worst Case** |
| **Search** | $O(\log n)$[1] | $O(\log n)$[1] |
| **Insert** | $O(1)$[2] | $O(\log n)$[1] |
| **Delete** | $O(1)$[2] | $O(\log n)$[1] |

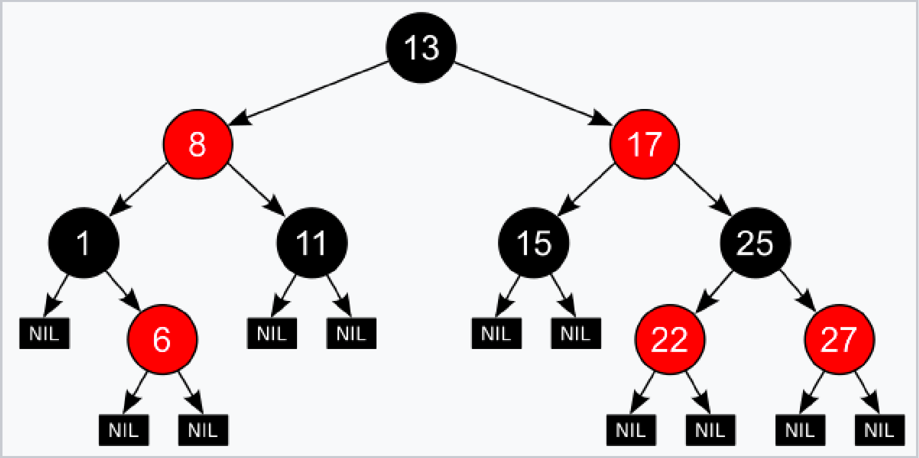


Figure 1: ... with explicit NIL leaves

# Edge- *vs.* level-triggered?

jedi@tortuga:~

```
Level-triggered and edge-triggered
       The  epoll event distribution interface is able to behave both as edge-trig‐
       gered (ET) and as level-triggered (LT).   The  difference  between  the  two
       mechanisms can be described as follows.  Suppose that this scenario happens:

       1. The file descriptor that represents the read side of a pipe (rfd) is reg‐
          istered on the epoll instance.

       2. A pipe writer writes 2 kB of data on the write side of the pipe.

       3. A call to epoll_wait(2) is done that will return rfd as a ready file  de‐
          scriptor.

       4. The pipe reader reads 1 kB of data from rfd.

       5. A call to epoll_wait(2) is done.

       If  the  rfd file descriptor has been added to the epoll interface using the
       EPOLLET (edge-triggered) flag, the call to epoll_wait(2) done in step 5 will
       probably  hang  despite  the  available data still present in the file input
       buffer; meanwhile the remote peer might be expecting a response based on the
       data  it  already sent.  The reason for this is that edge-triggered mode de‐
       livers events only when changes occur on the monitored file descriptor.  So,
       in  step  5  the  caller  might end up waiting for some data that is already
       present inside the input buffer.  In the above example, an event on rfd will
       be  generated because of the write done in 2 and the event is consumed in 3.
       Since the read operation done in 4 does not consume the whole  buffer  data,
 Manual page epoll(7) line 42 (press h for help or q to quit)
```

jedi@tortuga:~

5. A call to **epoll_wait**(2) is done.

If the rfd file descriptor has been added to the **epoll** interface using the **EPOLLET** (edge-triggered) flag, the call to **epoll_wait**(2) done in step **5** will probably hang despite the available data still present in the file input buffer; meanwhile the remote peer might be expecting a response based on the data it already sent. The reason for this is that edge-triggered mode delivers events only when changes occur on the monitored file descriptor. So, in step **5** the caller might end up waiting for some data that is already present inside the input buffer. In the above example, an event on rfd will be generated because of the write done in **2** and the event is consumed in **3**. Since the read operation done in **4** does not consume the whole buffer data, the call to **epoll_wait**(2) done in step **5** might block indefinitely.

An application that employs the **EPOLLET** flag should use nonblocking file descriptors to avoid having a blocking read or write starve a task that is handling multiple file descriptors. The suggested way to use **epoll** as an edge-triggered (**EPOLLET**) interface is as follows:

a) with nonblocking file descriptors; and

b) by waiting for an event only after **read**(2) or **write**(2) return **EAGAIN**.

By contrast, when used as a level-triggered interface (the default, when **EPOLLET** is not specified), **epoll** is simply a faster **poll**(2), and can be used wherever the latter is used since it shares the same semantics.

Manual page epoll(7) line 57 (press h for help or q to quit)

jedi@tortuga:~

Since even with edge-triggered **epoll**, multiple events can be generated upon receipt of multiple chunks of data, the caller has the option to specify the **EPOLLONESHOT** flag, to tell **epoll** to disable the associated file descriptor after the receipt of an event with **epoll_wait**(2). When the **EPOLLONESHOT** flag is specified, it is the caller's responsibility to rearm the file descriptor using **epoll_ctl**(2) with **EPOLL_CTL_MOD**.

If multiple threads (or processes, if child processes have inherited the **epoll** file descriptor across **fork**(2)) are blocked in **epoll_wait**(2) waiting on the same epoll file descriptor and a file descriptor in the interest list that is marked for edge-triggered (**EPOLLET**) notification becomes ready, just one of the threads (or processes) is awoken from **epoll_wait**(2). This provides a useful optimization for avoiding "thundering herd" wake-ups in some scenarios.

**Interaction with autosleep**

If the system is in **autosleep** mode via /sys/power/autosleep and an event happens which wakes the device from sleep, the device driver will keep the device awake only until that event is queued. To keep the device awake until the event has been processed, it is necessary to use the **epoll_ctl**(2) **EPOLLWAKEUP** flag.

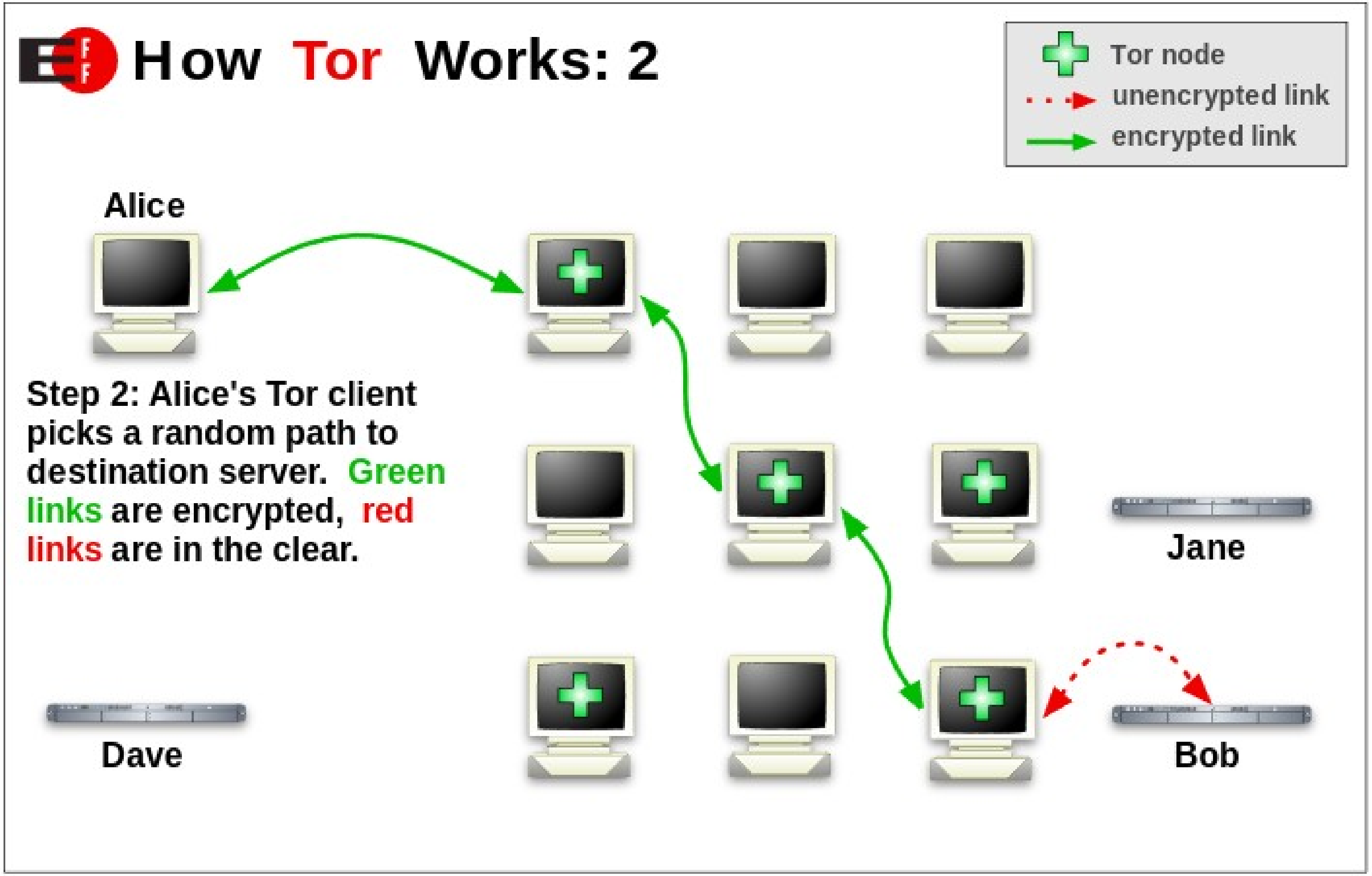
When the **EPOLLWAKEUP** flag is set in the **events** field for a struct epoll_event, the system will be kept awake from the moment the event is queued, through the **epoll_wait**(2) call which returns the event until the subsequent **epoll_wait**(2) call. If the event should keep the system awake

Manual page epoll(7) line 83 (press h for help or q to quit) https://en.wikipedia.org/wiki/Thundering_herd_problem

- “In computer science, the thundering herd problem occurs when a large number of processes or threads waiting for an event are awoken when that event occurs, but only one process is able to handle the event. When the processes wake up, they will each try to handle the event, but only one will win. All processes will compete for resources, possibly freezing the computer, until the herd is calmed down again.”

# Use case: Tor

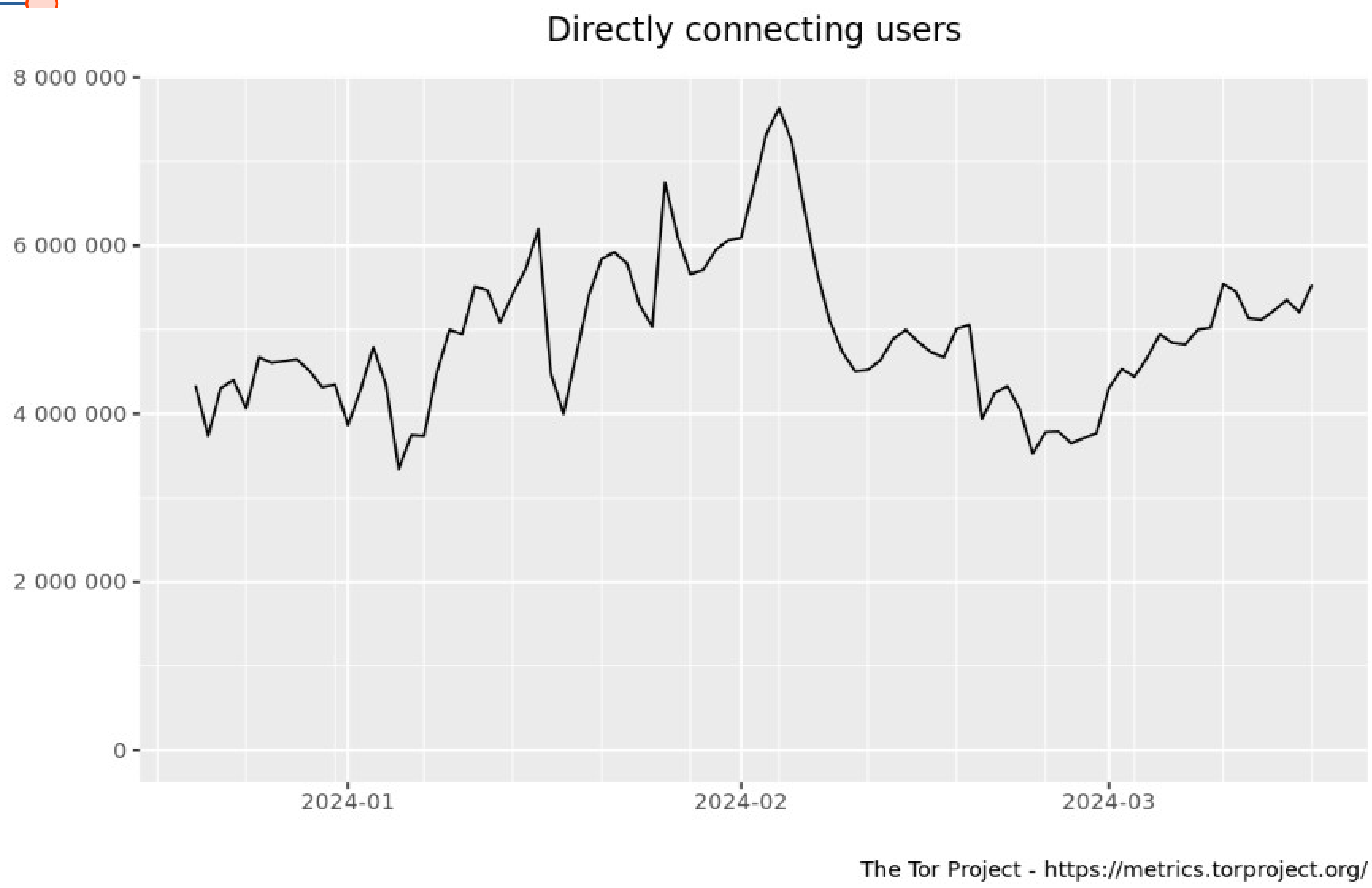
- Overlay network that provides anonymity and censorship resistance
- Easy to use (Tor browser), open source, friendly to academics (and lots of data)
- Uses epoll() for Linux and kqueue() for BSD flavors



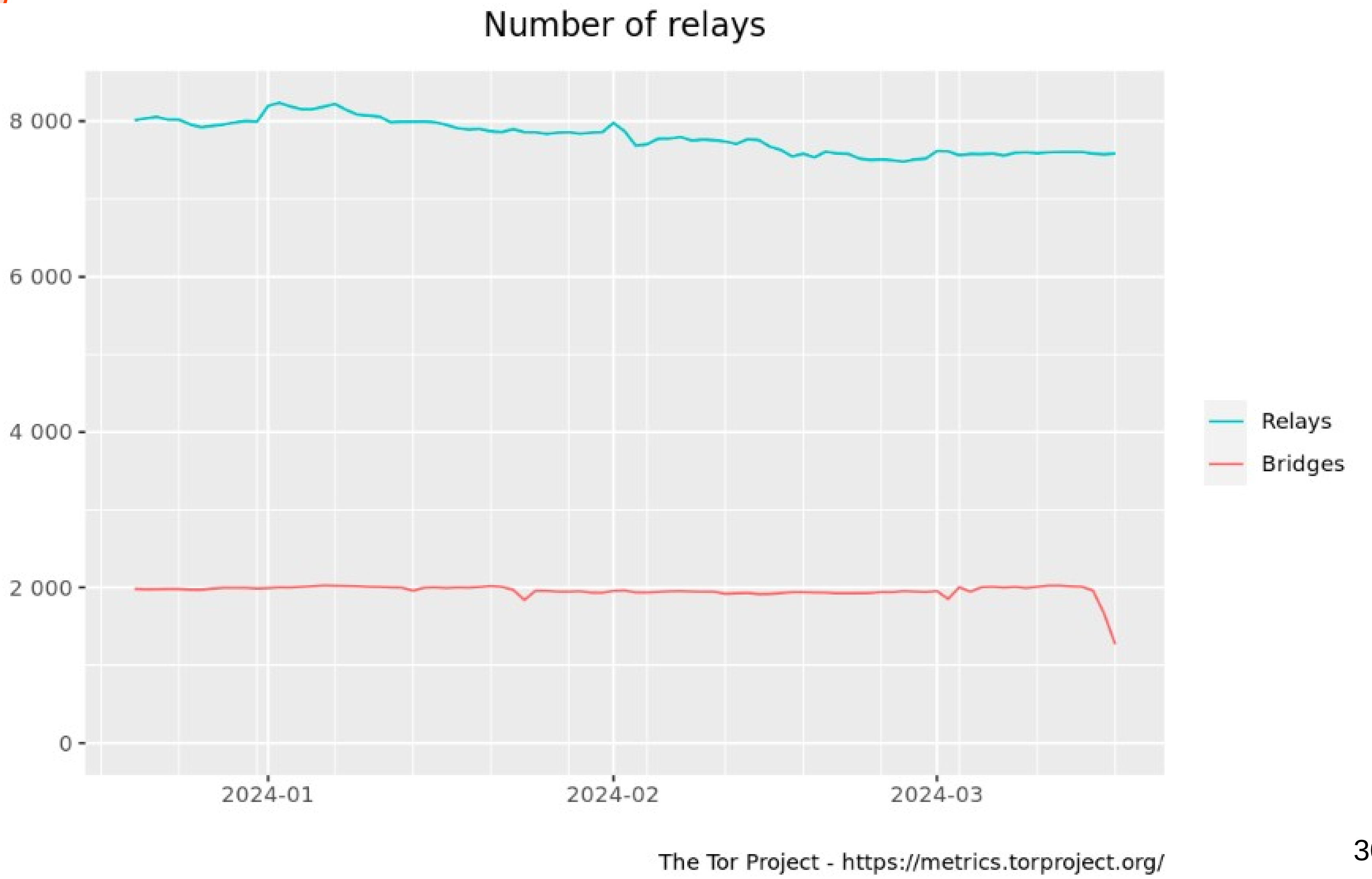

https://en.wikipedia.org/wiki/Tor_(network)

# Some metrics from https://metrics.torproject.org/

…

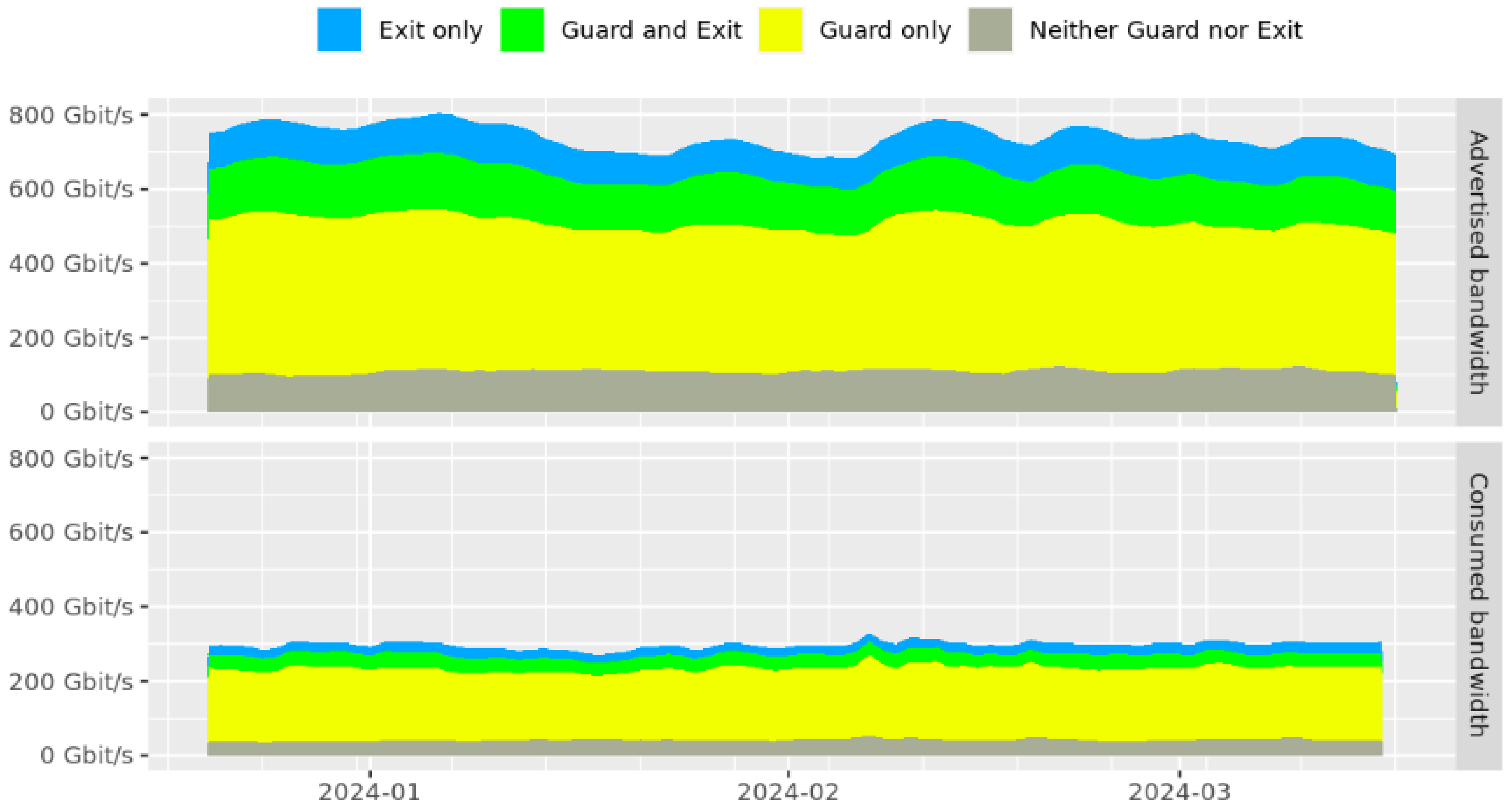


Directly connecting users
8 000 000
6 000 000
4 000 000
2 000 000
0
2024-01
2024-02
2024-03
The Tor Project - https://metrics.torproject.org/



Number of relays
8 000
6 000
4 000
2 000
0
2024-01
2024-02
2024-03
Relays
Bridges
The Tor Project - https://metrics.torproject.org/

# Advertised and consumed bandwidth by relay flags



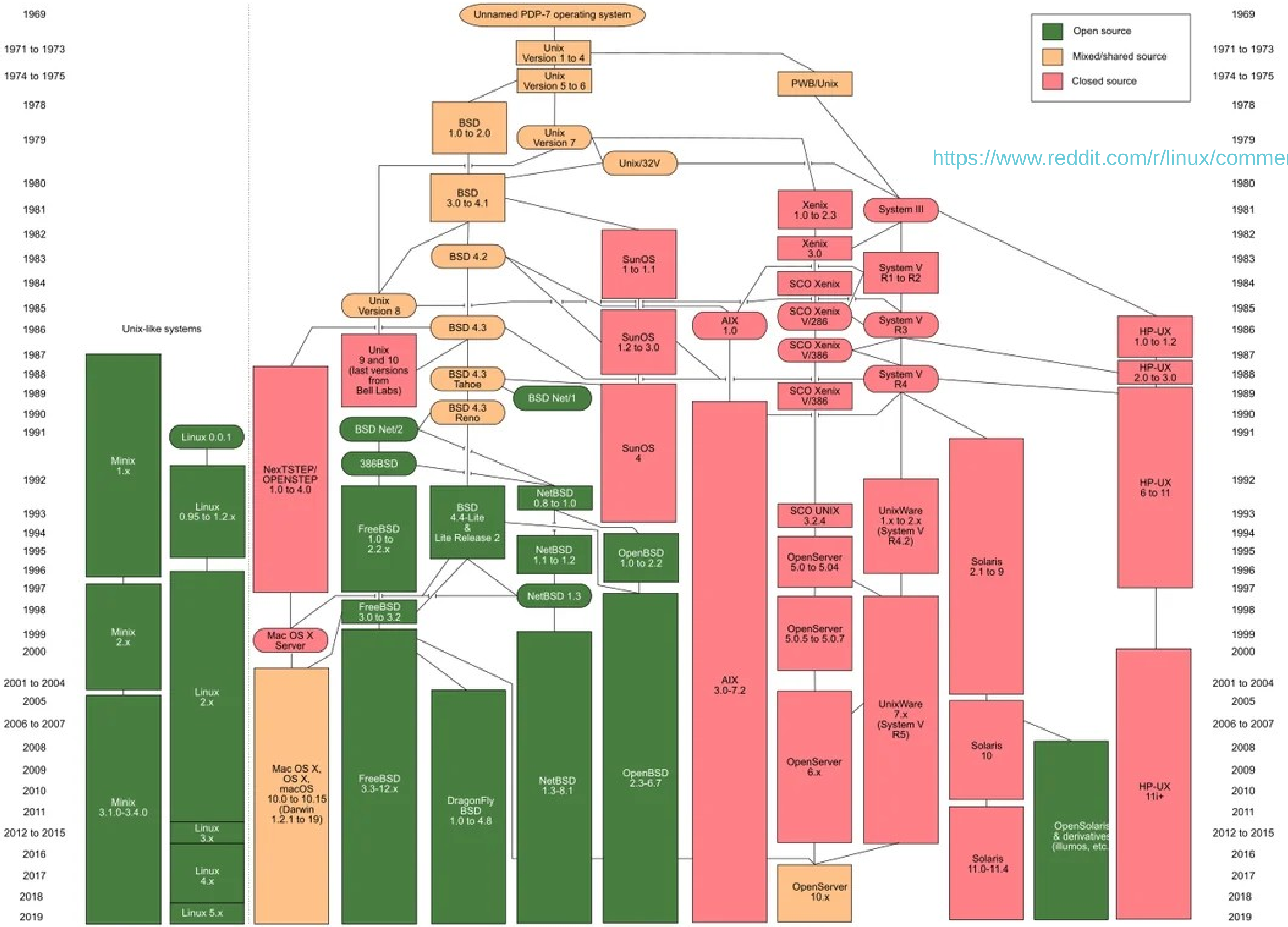

The Tor Project - https://metrics.torproject.org/

# What is kqueue()

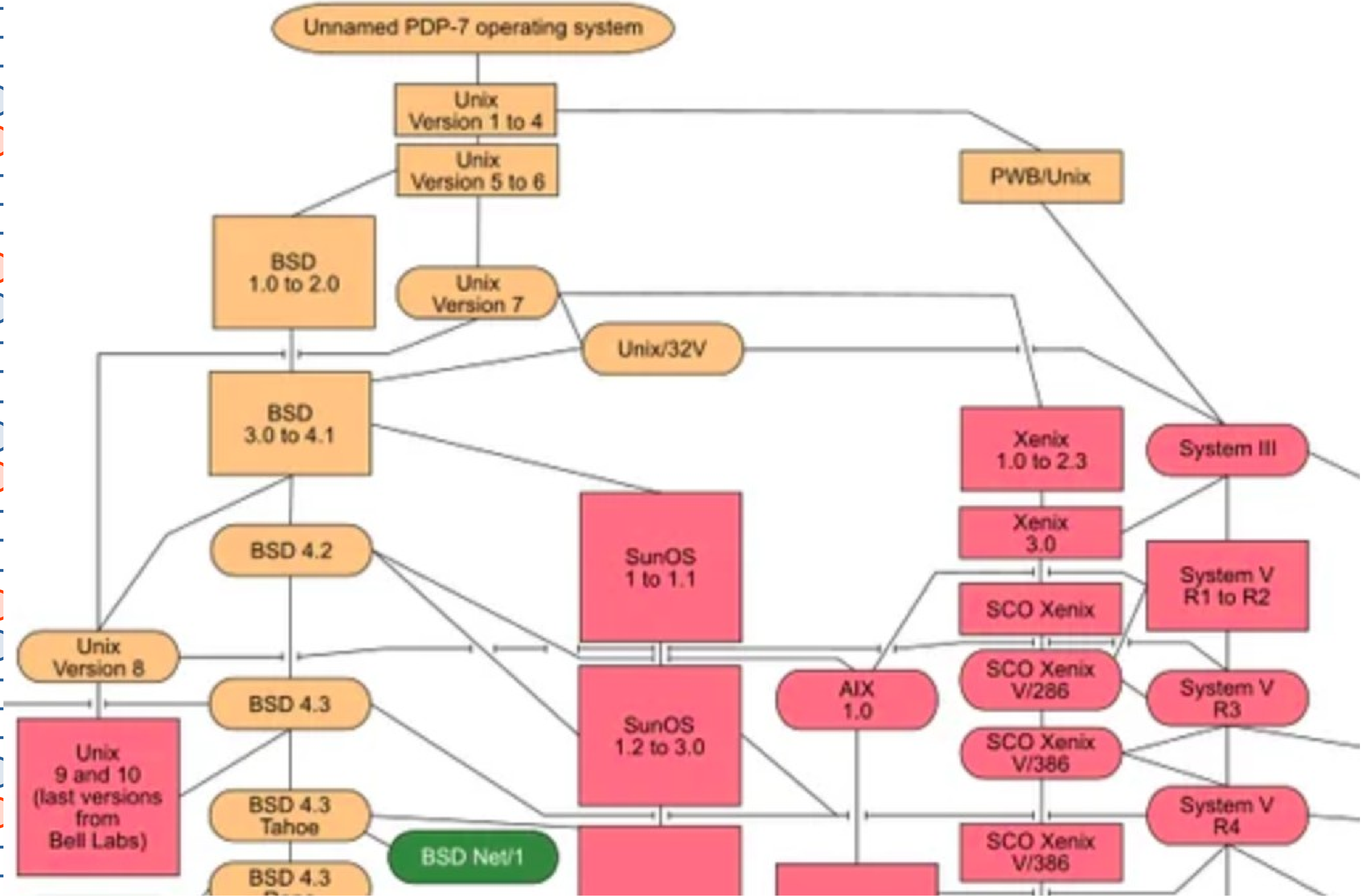
- Similar to epoll(), but for BSD flavors
- “Kqueue allows one to batch modify watcher states and to retrieve watcher states in a single system call. With epoll, you have to call a system call for every modification. Kqueue also allows one to watch for things like filesystem changes and process state changes, epoll is limited to socket/pipe I/O only.”

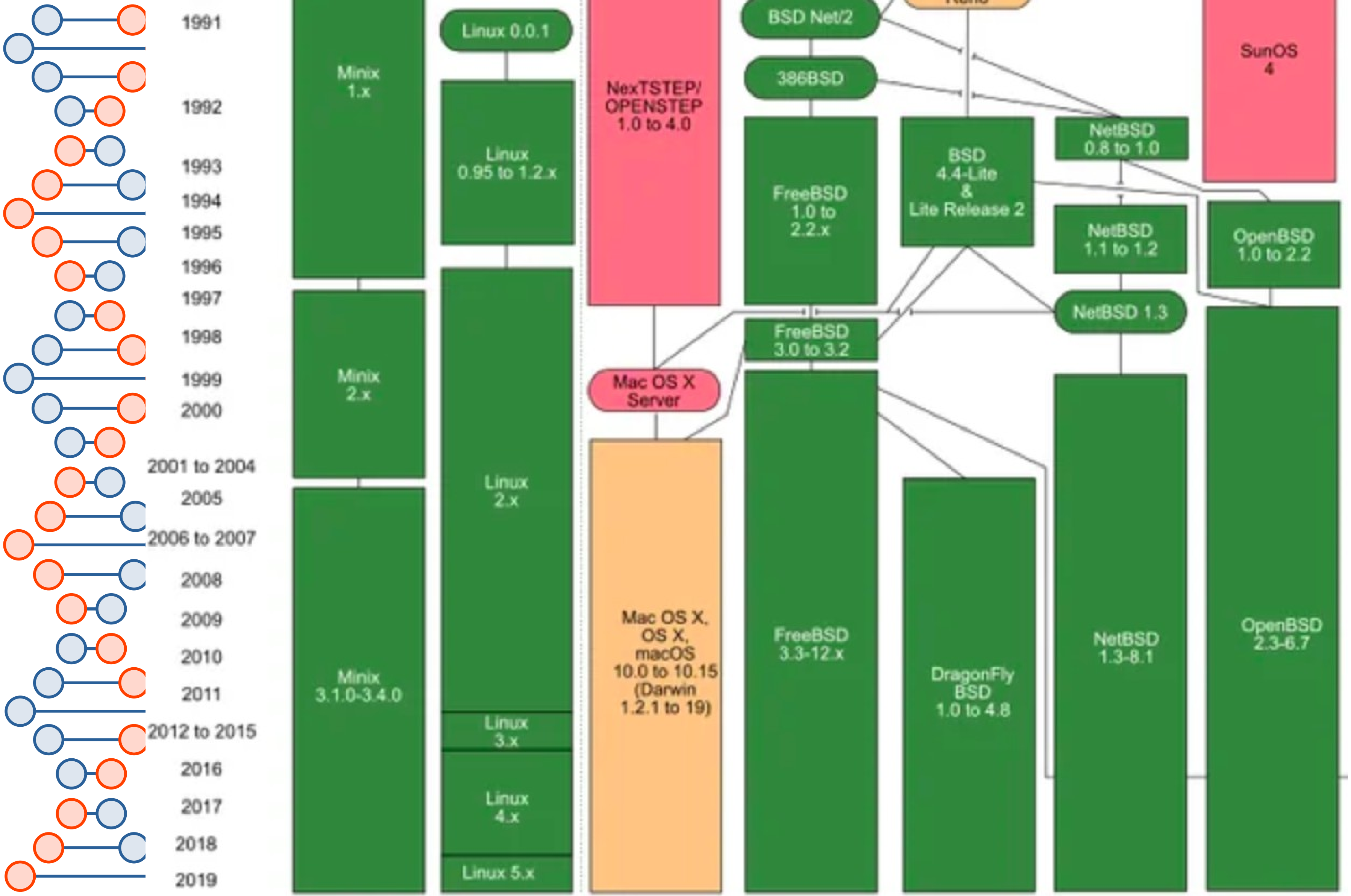
  --asomiv, https://news.ycombinator.com/item?id=3028687
- Linux has inotify()
- libuv and libevent support kqueue(), epoll(), and alternatives such as Solaris I/O completion ports, Windows IOCP, *etc.*

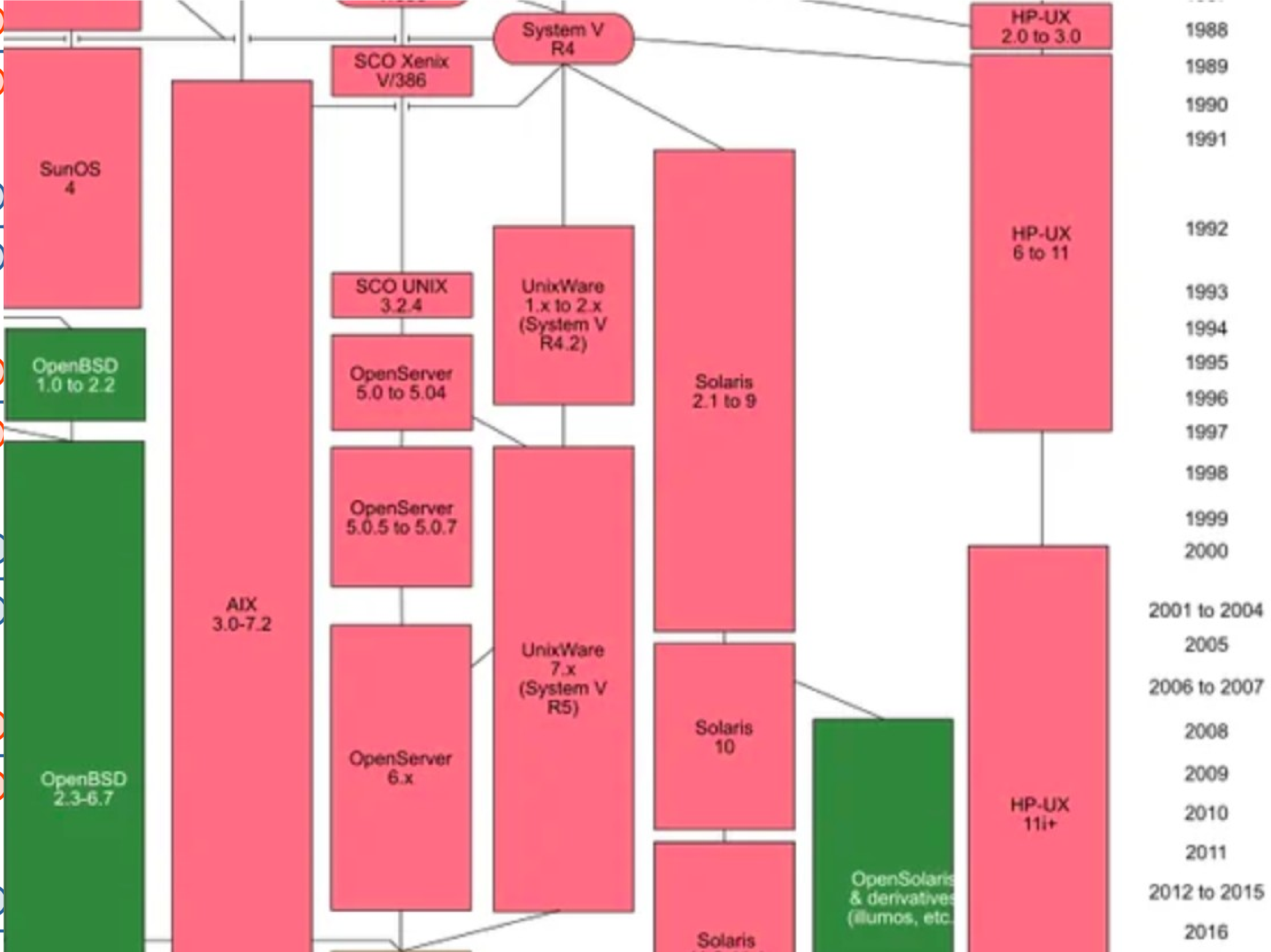
We’ve mentioned Solaris a few times, now seeing a difference from Linux *vs*. BSD, and we’ve largely ignored Windows this semester. Now’s a good time for a 5-minute digression into OS history...

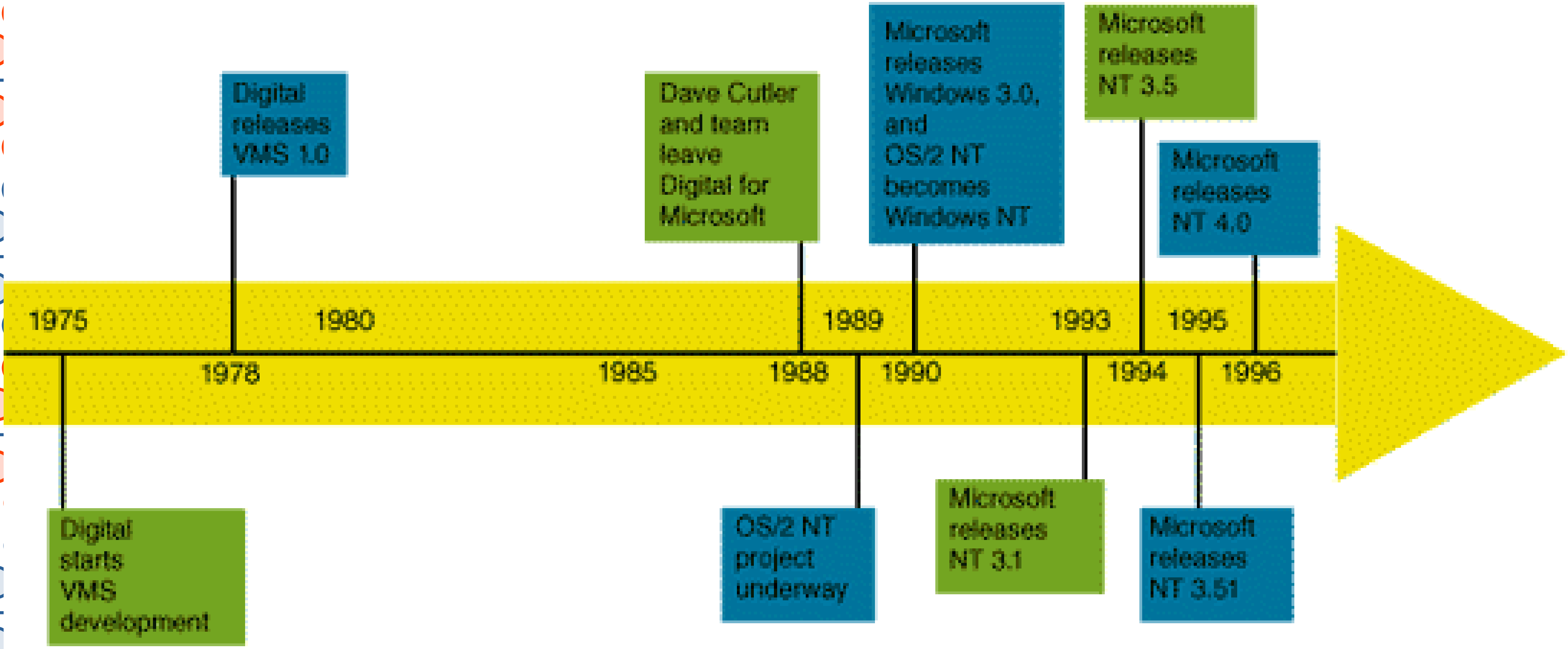



https://www.reddit.com/r/linux/comments/huhqrh/unix_family_tree/


Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
SunOS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
AIX 1.0
System V R3
BSD 4.3
SunOS 1.2 to 3.0
Unix 9 and 10 (last versions from Bell Labs)
SCO Xenix V/386
BSD 4.3 Tahoe
System V R4
BSD Net/1
SCO Xenix V/386
BSD 4.3


1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2015
2016
2017
2018
2019
Minix 1.x
Minix 2.x
Minix 3.1.0-3.4.0
Linux 0.0.1
Linux 0.95 to 1.2.x
Linux 2.x
Linux 3.x
Linux 4.x
Linux 5.x
NexTSTEP/ OPENSTEP 1.0 to 4.0
Mac OS X Server
Mac OS X, OS X, macOS 10.0 to 10.15 (Darwin 1.2.1 to 19)
BSD Net/2
386BSD
FreeBSD 1.0 to 2.2.x
FreeBSD 3.0 to 3.2
FreeBSD 3.3-12.x
BSD 4.4-Lite & Lite Release 2
DragonFly BSD 1.0 to 4.8
NetBSD 0.8 to 1.0
NetBSD 1.1 to 1.2
NetBSD 1.3
NetBSD 1.3-8.1
SunOS 4
OpenBSD 1.0 to 2.2
OpenBSD 2.3-6.7

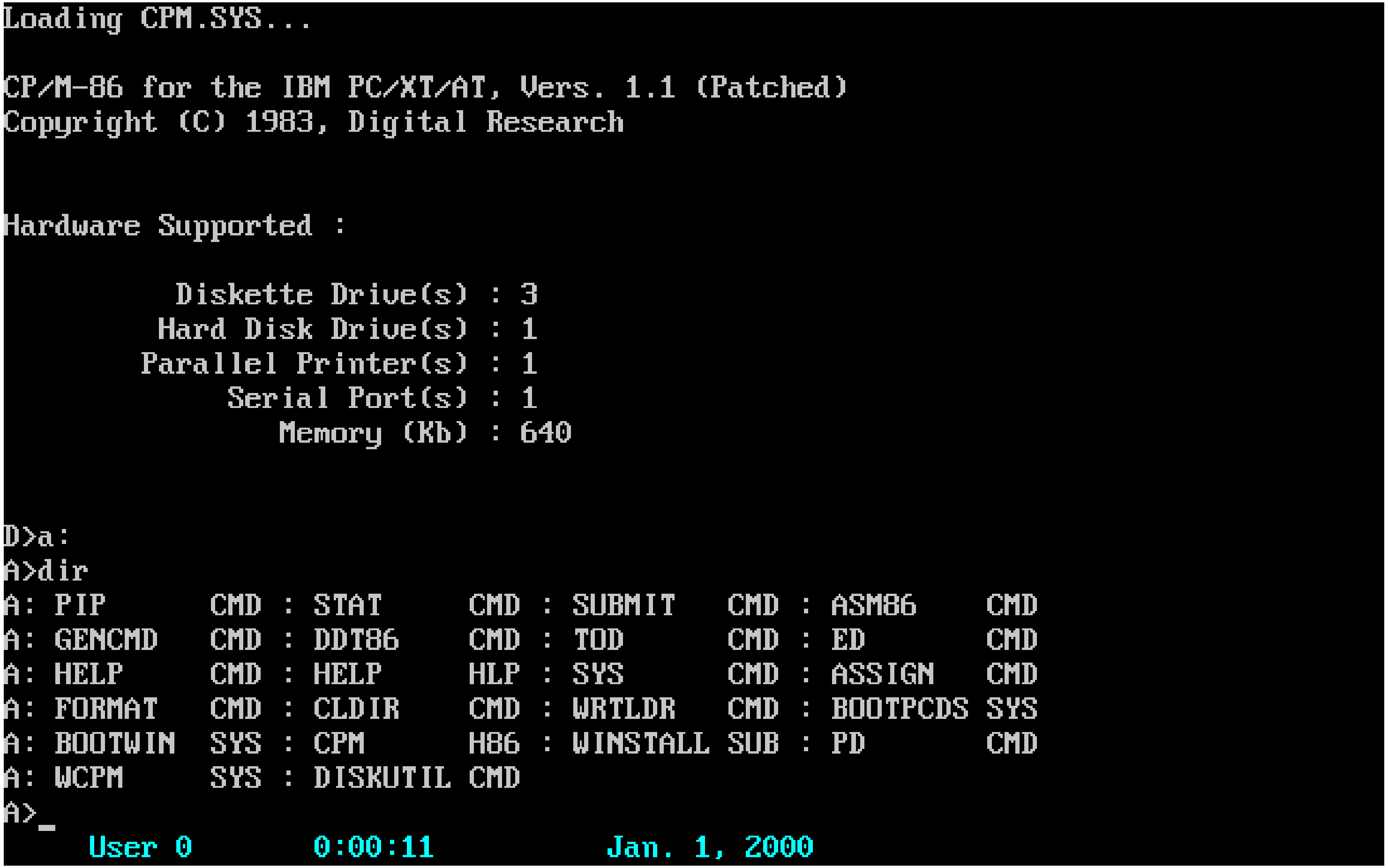

System V R4
HP-UX 2.0 to 3.0
1988
SCO Xenix V/386
1989
1990
1991
SunOS 4
HP-UX 6 to 11
1992
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
1993
1994
Solaris 2.1 to 9
1995
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
1996
1997
1998
OpenServer 5.0.5 to 5.0.7
1999
2000
AIX 3.0-7.2
2001 to 2004
2005
UnixWare 7.x (System V R5)
2006 to 2007
Solaris 10
2008
OpenServer 6.x
2009
OpenBSD 2.3-6.7
HP-UX 11i+
2010
2011
2012 to 2015
2016




https://www.tech-insider.org/windows/research/1998/images/figure_01.gif

# CPM - 1974

https://en.wikipedia.org/wiki/CP/M#/media/File:CPM-86.png



February 3, 1976

An Open Letter to Hobbyists

To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software and an owner who understands programming, a hobby computer is wasted. Will quality software be written for the hobby market?

Almost a year ago, Paul Allen and myself, expecting the hobby market to expand, hired Monte Davidoff and developed Altair BASIC. Though the initial work took only two months, the three of us have spent most of the last year documenting, improving and adding features to BASIC. Now we have 4K, 8K, EXTENDED, ROM and DISK BASIC. The value of the computer time we have used exceeds $40,000.

[See the fulll letter at https://en.wikipedia.org/wiki/File:Bill_Gates_Letter_to_Hobbyists_ocr.pdf]

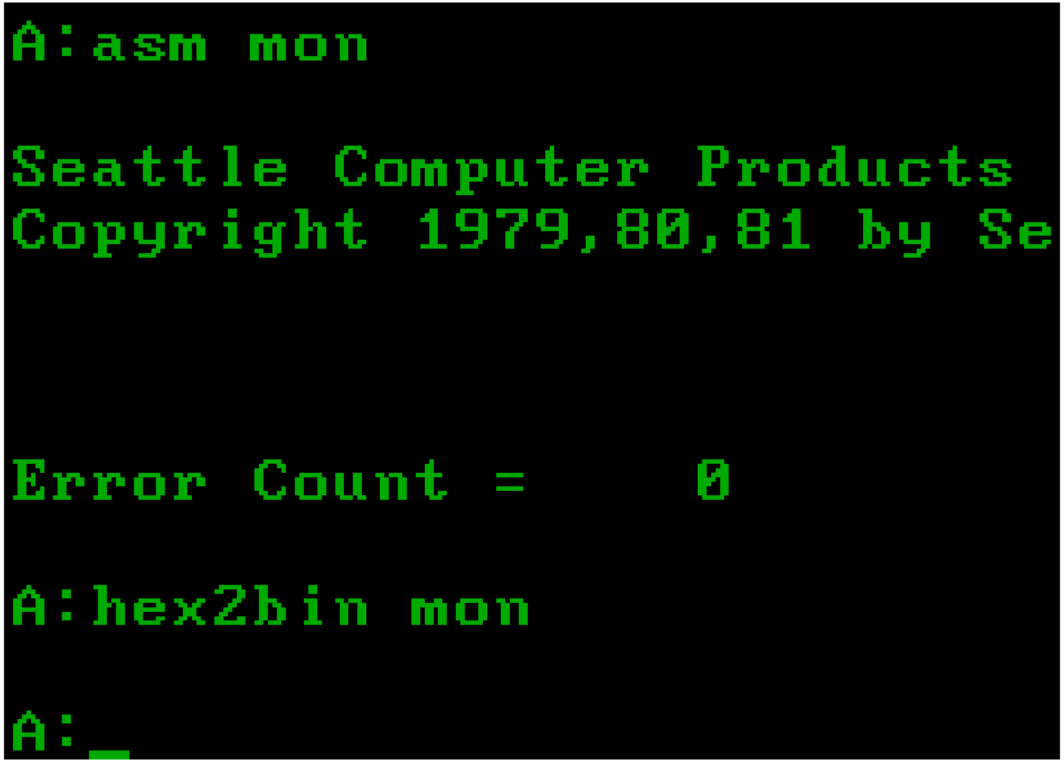
I would appreciate letters from any one who wants to pay up, or has a suggestion or comment. Just write me at 1180 Alvarado SE, #114, Albuquerque, New Mexico, 87108. Nothing would please me more than being able to hire ten programmers and deluge the hobby market with good software.

Bill Gates

Bill Gates
General Partner, Micro-Soft

# QDOS - 1979

https://en.wikipedia.org/wiki/86-DOS#/media/File:86-DOS_running_assembler_and_HEX2BIN_(screenshot).png



# MS-DOS - 1981

https://upload.wikimedia.org/wikipedia/commons/b/b6/StartingMsdos.png