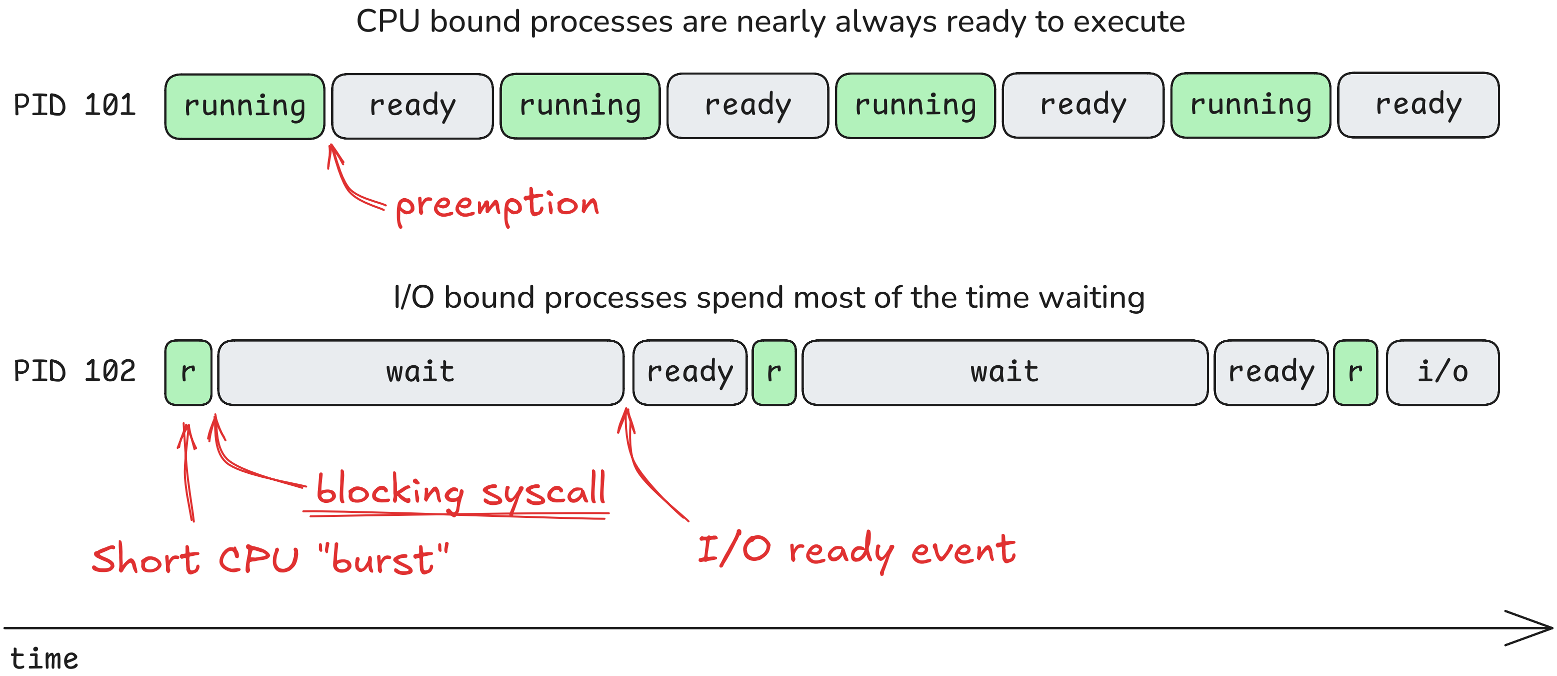
Lecture 5 - Event I/O

Operating Systems 1

Warsaw University of Technology - Faculty of Mathematics and Information Science

Application profiles

IO-bound vs. CPU-bound applications



CPU bound processes are nearly always ready to execute

PID 101 running | ready | running | ready | running | ready | running | ready

preemption

I/O bound processes spend most of the time waiting

PID 102 r | wait | ready | r | wait | ready | r | i/o

blocking syscall

Short CPU "burst"

I/O ready event

time

Blocking syscalls problem

I/O bound application main tool

Examples: stdin read(), sigsuspend(), wait().
It's a natural tool to wait-for (synchronize with) external events.
Events: "user put some text", "network data came", "other process finished".

```
while (run) {
int ret = read(fd, buf, N);
if (ret < 0) ERR();
handle_read(buf, ret);
}
```

caller blocks here

When invoked:
- calling process cooperatively yields CPU to others
- waits not consuming any CPU time

Problems:
- cannot do any other computation in the meantime
- can't wait for other, unrelated external events

How to wait for "two things" at once?
Consider:
- reading from multiple data sources
- reading and doing some computation periodically

Alternative: busy waiting

How to unnecessarily consume CPU resources

One could refrain from using blocking syscalls at all and use ones which guarantee to return control immediately.

```
while (run) {
ret = read(fd1, buf, N, O_NONBLOCK);
if (ret >= 0) handle_read(buf, ret);
ret = read(fd2, buf, N, O_NONBLOCK);
if (ret >= 0) handle_read(buf, ret);
}
```

non-blocking read allows
to check multiple sources
in an interleaved manner

This kind of API must define to return something,
usually erros like EAGAIN, if it could not complete immediately.
Those are not real errors, handling them is wasted time.

The above loop is not I/O bound, it's CPU bound, never blocking,
always ready for execute, despite most of the time there's nothing to do.
As such, this is wasteful approach, discouraged in general purpose OS programming.

Asynchronous event-based programming

Solution to a blocking syscall problem

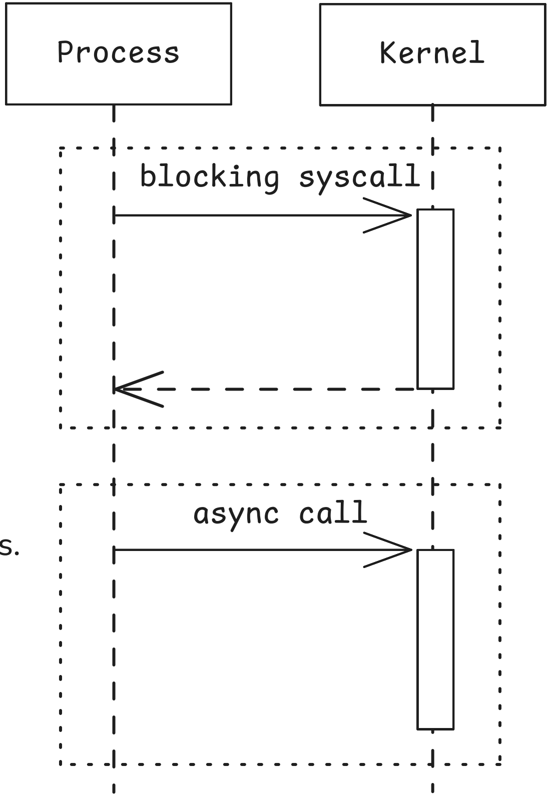
Instead of blocking just schedule an operation via syscall.
Ask the OS to notify calling process via event upon completion.

```
// configure notification
struct async_event ev = {};
// schedule operation(s)
int ret = async_read(fd, buf, N, &ev);
// do something else
```

never blocks!

Async API is far less natural to the programmer. It requires configuring not only the operation but also how it delivers notifications.

Later, scheduling process must somehow receive completion event(s). Put another way, it needs to synchronize with the async operation.



Event sources

What asynchronous events a program might be interested in?

timer ticks, timer elapse

run some computation every 3 seconds ← example functional requirement

I/O readiness

process command when user enters something into the terminal

Filesystem events

when user creates new file in a directory D run some computation to process it

Signal delivery

exit upon receiving C-c

Arbitrary application-level events

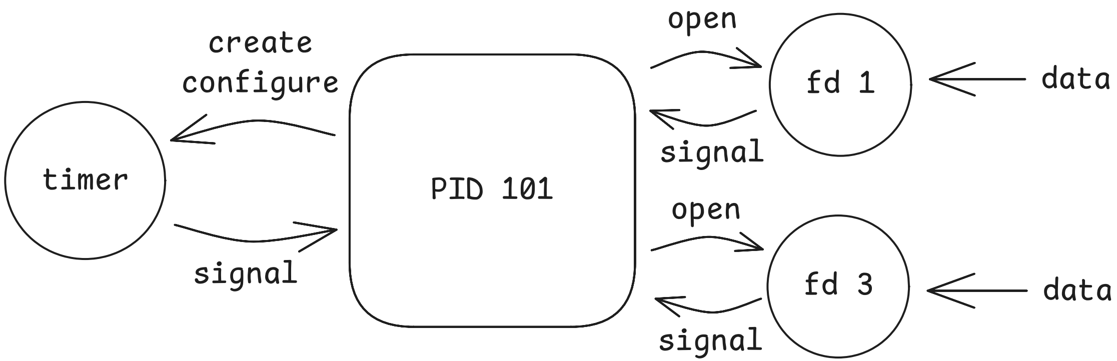
wait until other process/thread is done with some processing

Signal-based event handling

async_fd.c

Asynchronous way of handling asynchronous things

Signals are a classic way to handle asynchronous events. OS may be asked to send a signal when some event occurs, i.e. fd = 1 became readable or configured time has elapsed.



This approach suffers from a number of pitfalls:
- signals sticking
- handler code restritions (async-signal-safe)
- does not scale (limited signal number)
- interrupts syscalls

Synchronous signal handling via sigsuspend() or sigwait() also suffers from these

Timer API

Standard POSIX interface for creating signalling timers

```
timer_create.c
```

```
> man 3p timer_create
```

system clock ID
monotonic/realtime/...

what happens when
timer elapses?

```
int timer_create(clockid_t clockid, struct sigevent *restrict evp,
timer_t *restrict timerid);
```

returned timer ID

```
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
int timer_settime(timer_t timerid, int flags,
const struct itimerspec* new_value,
struct itimerspec* old_value);
```

timer config
initial wait + interval

The sigevent structure descibes notification. Process specifies there signal number which it wants devlivered. Notification by thread is also supported.

Syscall based event handling

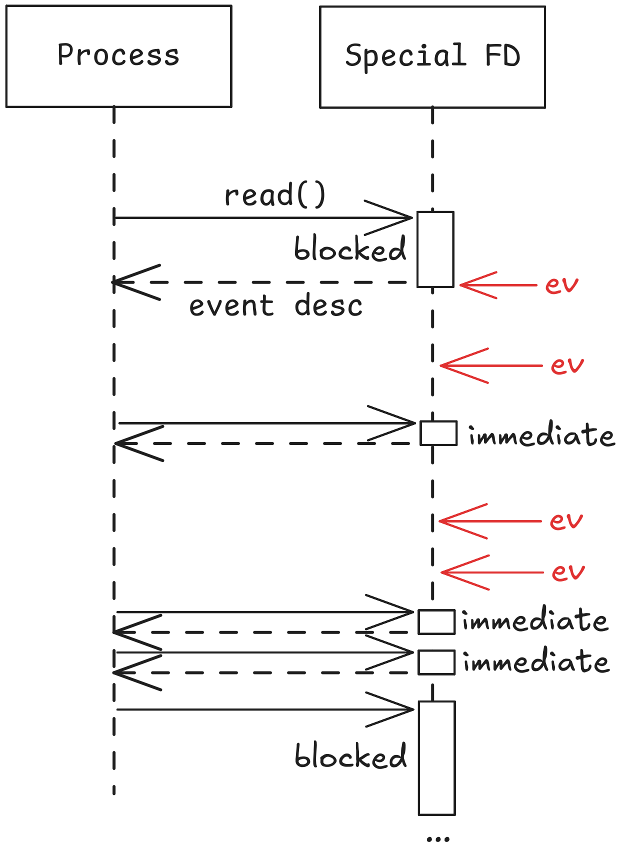
Synchronous way of handling events

OS exposes different way of handling async events. It provides syscalls blocking until event occurs and returning info about it.

It frequently takes form of blocking read() from a (special) file descriptor. Reads return structures describing the event.

```
int fd = make_event_source();
struct event ev;
read(fd, &ev, sizeof(ev));
// process ev
```

In the spirit of "everything is a file" Linux provides special FDs for receiving timer events, signals, filesystem changes, networking I/O events, IPC, and others.



TimerFD API

Modern, signal-free timers

```
timerfd_create.c
```

```
> man 2 timerfd_create
```

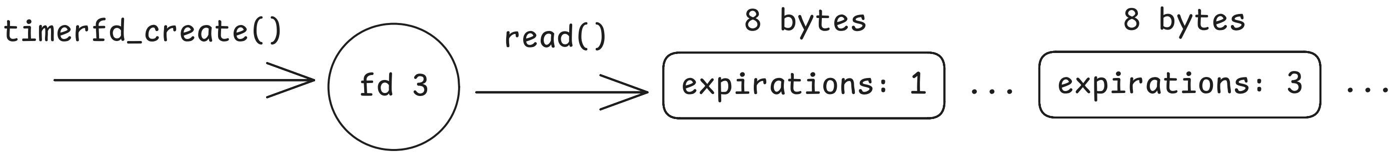
```
int timerfd_create(int clockid, int flags);
```

creates special descriptor
referencing the new timer

```
int timerfd_settime(int fd, int flags,
const struct itimerspec* new_value,
struct itimerspec* old_value);
```

```
int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

Having a timer fd process can read() from it to obtain 8-byte integers representing elapse events. Until timer elapses, read blocks. When elapsed, read completes immediately. Returned integer value holds number of timer expirations since last read.



timerfd_create()

fd 3

read()

8 bytes

expirations: 1

...

8 bytes

expirations: 3

...

SignalFD API

Signals without signal handler restrictions

```
signalfd.c
```

```
> man 2 signalfd
```

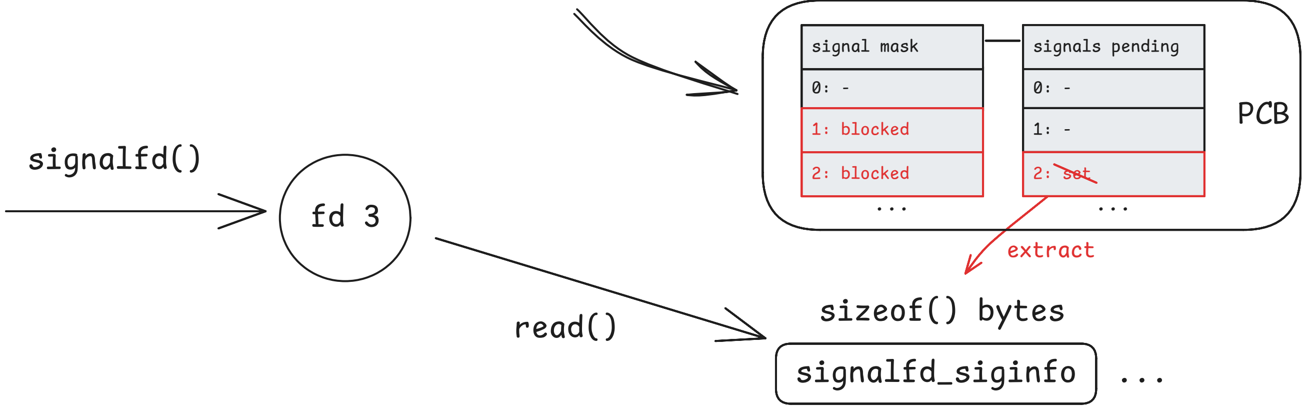
Signals also may be received by regular reads. Signalfd provides alternative method to signal handlers or sigwait(). Special descriptor reads return struct signalfd_siginfo instances upon receiving a signal.

-1 creates and returns new FD

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

set of awaited signals

Awaited signals shall be blocked using sigprocmask().



signal mask
0: -
1: blocked
2: blocked
...

signals pending
0: -
1: -
2: ~~set~~
...

SignalFD API

```
> man 2 signalfd
```

Signals without signal handler restrictions

Data structure instances received via read() contain vairous attributes related to the consumed signal instance.

```
struct signalfd_siginfo {
uint32_t ssi_signo; /* Signal number */
uint32_t ssi_pid; /* PID of sender */
int32_t ssi_fd; /* File descriptor (SIGIO) */
int32_t ssi_status; /* Exit status or signal (SIGCHLD) */
int32_t ssi_int; /* Integer sent by sigqueue(3) */
uint64_t ssi_utime; /* User CPU time consumed (SIGCHLD) */
uint64_t ssi_stime; /* System CPU time consumed (SIGCHLD) */
/* ... lots of other attributes */
};
```

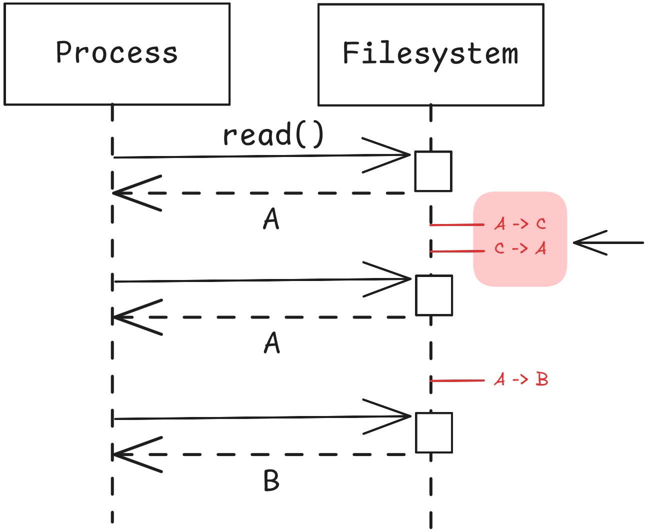
Signalfd approach still suffers from sticking and other issues within signals nature.
Everlasting block on mask protects main code from signals, void signal handler code related problems.

It is very similar to what sigwait() does but provides FD-based interface. ← this is gonna prove extremely useful later today!

Change events

Observing state of system resources

Frequently apps need to execute some action when some OS object changes.
Example: buildserver, observing contents of the source tree, rebuilding when anything changes.



Without state change events process would need to periodically poll and diff state (wasting CPU time).

This solution also suffers from ABA problem.
It might miss a change if polling is not frequent enough.

Ultimately, the OS knows best when something changes, since it executes the change. It should notify us!

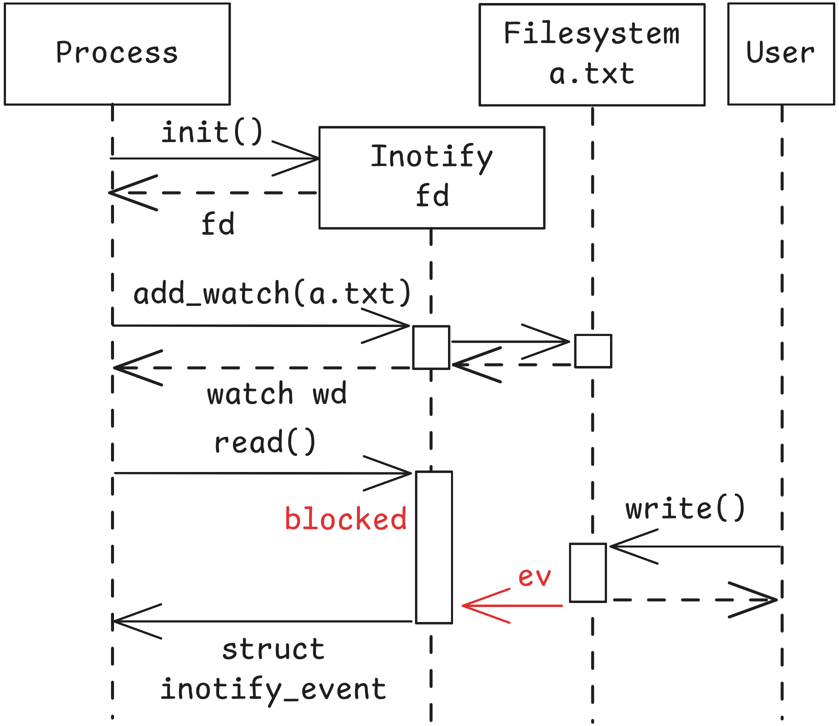
inotify API

```
> man 7 inotify
```

Meet the mighty filesystem monitoring interface

Inotify is an OS object, which needs to be created and is referred to via special file descriptor. Process has to register watches within the inotify instance and then can await change events with read().

Inotify can hold multiple watches: on individual files, or whole directories. Directory watches are non recursive.

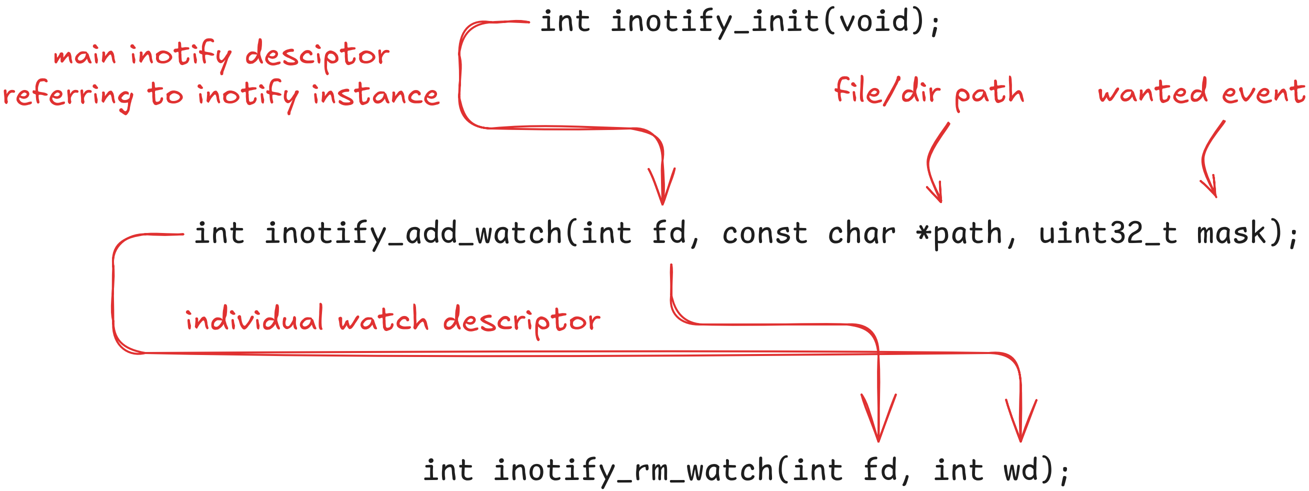


inotify API

Initializing and managing watches

```
> man 2 inotify_init
```

```
> man 2 inotify_add_watch
```



```
int inotify_init(void);

int inotify_add_watch(int fd, const char *path, uint32_t mask);

int inotify_rm_watch(int fd, int wd);
```

inotify API

```
> man 7 inotify
```

Reading filesystem events

After registering watches, read(fd) will return instances of struct inotify_event for each detected file/dir change:

```
struct inotify_event {
int wd; /* Watch descriptor */
uint32_t mask; /* Mask describing event */
uint32_t cookie; /* Unique cookie (for rename(2)) */
uint32_t len; /* Size of name field */
char name[]; /* Optional null-terminated name */
};
```

allows for watch identification

what has happened?

only for directory watches

Instances for directory events vary in size. Application should be ready to handle largest possible:

```
sizeof(struct inotify_event) + NAME_MAX + 1
```

Many may be read in a single read() call.

inotify API

Event types

Mask lists event types which happened. Different events may happen depending if watch refers to dir or file.

File

data changed

IN_MODIFY

IN_ATTRIB

IN_DELETE_SELF*

IN_MOVE_SELF*

Directory

inode changed

IN_ATTRIB

IN_DELETE_SELF*

IN_MOVE_SELF*

watch subject
moved/deleted

File in watched directory

IN_ATTRIB

IN_CREATE

IN_DELETE

IN_MODIFY

IN_MOVED_FROM

IN_MOVED_TO

IN_ISDIR will be set in mask if event relates to directory (i.e. dir created)

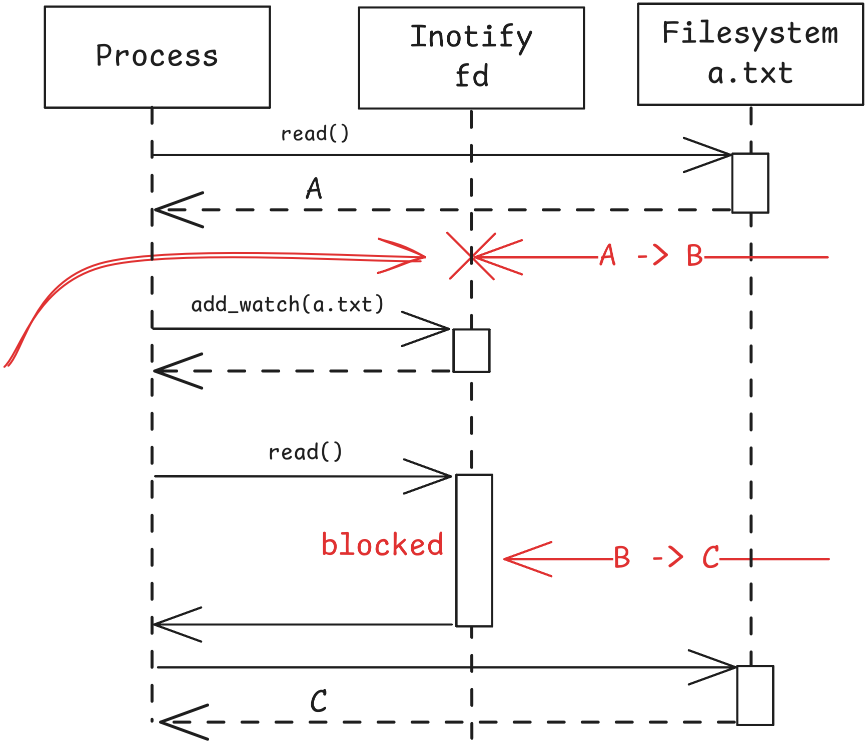
(*) When watched object or it's watch is removed special IN_IGNORED event is generated.

important to handle it!

Initial scan race condition

If an applications intends to cache contents of the watched object and update the cache when change is reported care must be taken on startup.

Initial scan must be executed AFTER adding watches. Otherwise app might miss change which happens before adding watch.



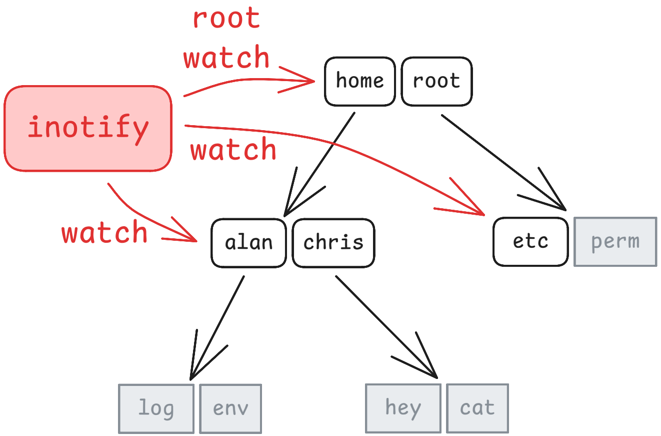
Directory tree watching

`watch_tree.c`

Watching directory trees requires registering individual watches for each subdirectory recursively.

After detecting that new sub-directory was created a corresponding watch must be registered.

Deleting a directory at some level generates bottom-up cascade of events, deleting watches automatically.



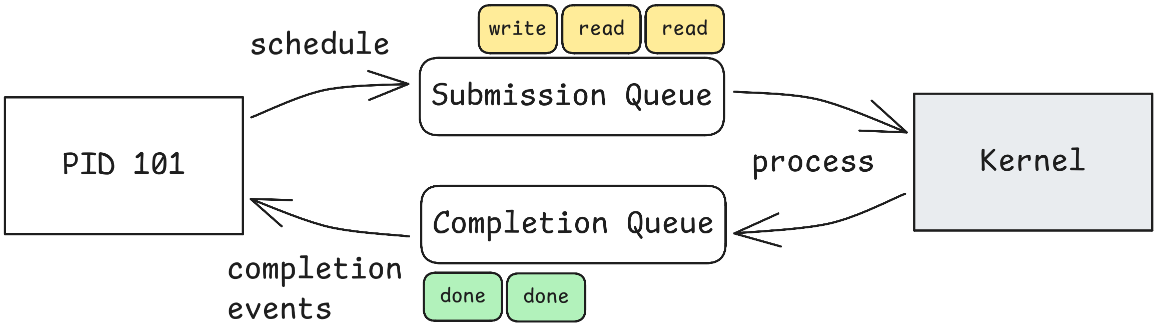
io_uring API

Modern async I/O interface

WARNING!
ADVANCED STUFF

```
> man 7 io_uring
```

For massive asynchronous read/write operations modern linux kernel introduced new API called io_uring IO userspace ring. It allows to submit multiple batched I/O operations to the kernel and read completion events. Communication happens over two queues: submission and completion.



Low-level syscall interface is very complex as it requires establishing queues in special memory segments shared between userspace and kernel.
Application usually consume io_uring API via a wrapper liburing library .

```
> man 3 io_uring_queue_init
```

io_uring API

Modern async I/O interface

First thing that needs to happen is creation of queues:

queues size

constructed ring (out)

```
int io_uring_queue_init(unsigned entries,
struct io_uring *ring,
unsigned flags);
```

Corresponding cleanup shall be called at the end of an application to free the queues:

```
void io_uring_queue_exit(struct io_uring *ring);
```

io_uring API

Submission queue management

```
> man 3 io_uring_get_sqe
```

```
> man 3 io_uring_prep_read
```

Sending I/O request is a 2-step process: populate queue entry (or entries) and then submit the queue.

```
struct io_uring_sqe *io_uring_get_sqe(struct io_uring *ring);
```

read()
args

```
void io_uring_prep_read(struct io_uring_sqe *sqe, int fd, void *buf,
unsigned nbytes, __u64 offset);
```

Scheduling whole queue is done via single call which translates to a non-blocking syscall:

```
> man 3 io_uring_submit
```

```
int io_uring_submit(struct io_uring *ring);
```

io_uring API

Completion queue management

```
> man 3 io_uring_wait_cqe
```

```
> man 3 io_uring_cqe_seen
```

Later, the event loop waits for and retrieves completion events:

```
struct io_uring_cqe cqe;
int io_uring_wait_cqe(struct io_uring *ring,
struct io_uring_cqe **cqe_ptr);
```

... and mark it as consumed to free the spot in CQ:

```
void io_uring_cqe_seen(struct io_uring *ring,
struct io_uring_cqe *cqe);
```

Completion event structure contains status code (errno) and optional arbitrary u64 passed during submission.

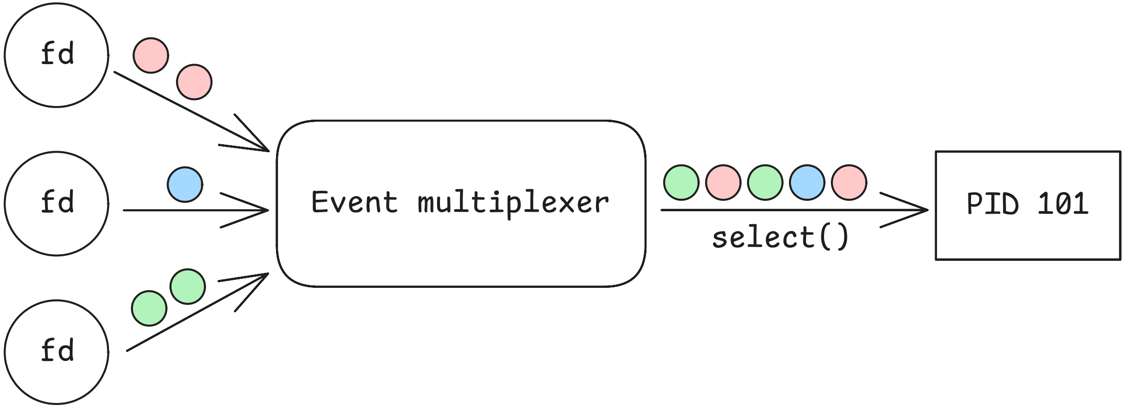
```
struct io_uring_cqe {
__u64 user_data; /* sqe->data submission passed back */
__s32 res; /* result code for this event */
__u32 flags;
};
```

Event multiplexing

How to process multiple event sources

Reading from a single data source blocks. How to run an application which awaits timers, stdin, filesystem, signals and other events at the same time?

Having coherent file desciriptor API, OS provides interface to await I/O events of mutliple file descriptors at once - the event multiplexer.



There are several different APIs for multiplexing events available: select(), poll(), epoll().

The select() syscall

```
> man 3p pselect
```

```
select_basic.c
```

How to read from mutliple file descriptors at once?

Select is a blocking syscall which takes mutliple file descriptors and returns when at least one is ready for I/O operation.

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

max fd value

descriptor sets

optional timeout

Three descriptor sets are like signal sets (bitmask) and must be constructed via macros before the call:

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

Select works with many I/O object types, including all mentioned in this lecture.

The event loop

`select_multiple.c`

How to read from mutliple file descriptors at once?

Select and other multiplexers are usually the heart of an event-driven application.
Blocking call is made in "the event loop". Each time multiplexer returns,
events are iteratively processed. Then app goes back to sleep awaiting new events.

```
while(run) {
FD_ZERO(&read_fds);
FD_SET(stdin_fd, &read_fds);
FD_SET(timer_fd, &read_fds);
// ... other fds
int n = select(max_fd + 1, &read_fds, NULL, NULL, NULL);
if (n < 0) ERR();
if (FD_ISSET(stdin_fd, &read_fds)) {
// read stdin
}
if (FD_ISSET(timer_fd, &read_fds)) {
// read timer events
}
}
```

prepare call

block

process events