BTI1341 – Operating Systems

Part 2: Concurrency – 9) IPC, Event-Based Concurrency

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- IPC and Locking
- Event-Based Concurrency
- Asynchronous I/O (AIO)
- Appendix

IPC and Locking

Multi-Programming and IPC

As we have seen in the last three parts, writing multi-threaded applications can be difficult.[1]

Another option is to use multi-programming , and if required, some inter-process communication (IPC) mechanism, e.g:

- Files and memory-mapped files
- Pipes and named pipes (FIFO)
- Signals
- Shared memory
- Unix domain sockets
- Message queues
- ...

[1] See also *06-concurrency-1* for a discussion of when to use threads.

IPC Synchronization

In general, issues of mutual exclusion and synchronization also hold in a multi-programming environment. Depending on the IPC mechanism used, different solutions apply.

Programs using e.g. shared memory or memory-mapped files for IPC need to take the same precautions for securing critical sections and avoiding deadlocks.

Consequently, it would be a good idea to be able to use the same primitives (e.g. mutexes and condition variables) also between multiple processes.[2]

[2] There are also additional mechanisms available, like e.g. file locking.

Memory-Mapped I/O I

Besides file I/O (e.g. `read()`, `write()`) and streams based I/O (e.g. `getc()`, `putc()`), memory-mapped I/O is another option for working with files.

Using `mmap()`, a file (or parts of it) can be mapped into a process' address space. It may then be accessed as any other memory region (e.g. through pointers) and the OS transparently handles reading and writing from or to the file. The corresponding C functions are:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Maps/unmaps **`length`** bytes from **`fd`**, starting at **`offset`** to address **`addr`** (**`NULL`** normally). **`prot`** is memory protection and **`flags`** contains options (e.g. **`MAP_SHARED`**).

Memory-Mapped I/O II

The following figure depicts the concept of memory-mapped I/O:

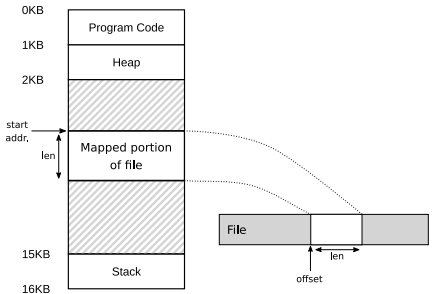

Figure: Memory-Mapped I/O

Courtesy of [ADAD18], Own Modification

Example: Memory-Mapped I/O

```
// open file
int fd = open("my-file", O_RDWR)));
assert(-1 != fd);

// map to memory and obtain pointer
void *ptr = mmap(NULL, len, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, file_offset);
assert(MAP_FAILED != ptr);

// fd is not required anymore
close(fd);


// ... do something with ptr ...


// unmap the file
assert(-1 != munmap(ptr, len))
```

Shared Memory

Using POSIX shared memory , two processes can share a memory region, eliminating the need to work with files:[3]

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Opens or creates a new shared memory region. The behavior is similar to `open()`, the function also returns a *file descriptor*, `name` should start with "`/`". Normally, the size of the region is then adjusted using `ftruncate()` and mapped using `mmap()` (the file descriptor may also be closed).

```
int shm_unlink(const char *name);
```

Removes the shared memory region and frees the resources after the last process has un-mapped it.

[3] On Linux, this works with files in the `/dev/shm` tmpfs file system (a sort of RAM disk). See `man shm_overview` for more details.

Example: Shared Memory

```c
struct shared_data {
  char c;
  int i;
}

// open shared memory region. name should start with '/'
int fd = shm_open("/data", O_CREAT | O_RDWR | O_TRUNC, S_IRUSR | S_IWUSR);
assert(-1 != fd);

// adjust size
assert(0 == ftruncate(fd, sizeof(struct shared_data)));

// mmap
struct shared_data *data = mmap(NULL, sizeof(struct shared_data),
                                PROT_READ | PROT_WRITE, MAP_SHARED,
                                fd, 0);
assert(MAP_FAILED != data);

// fd is not required anymore
close(fd);

data->i = 42;

// unmap and close shared memory region
assert(-1 != munmap(data, sizeof(struct shared_data)));
assert(-1 != shm_unlink("/data"));
```

Sharing Mutexes and CVs

Using shared memory between multiple processes, mutexes and condition variables (and also futexes, if available) may be exchanged for IPC synchronization:

```
struct shared_data {
  pthread_mutex_t mutex;
  pthread_cond_t cv;
  int i;
};
```

⚠ When shared, initialization must be done with `PTHREAD_PROCESS_SHARED`!

Example for `pthread_mutex_t`:

```
// prepare mutex attributes
pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_setpshared(&mutex_attr,
                             PTHREAD_PROCESS_SHARED);

// initialize mutex
pthread_mutex_t mutex;
pthread_mutex_init(mutex, &mutex_attr);
```

Named Semaphores

POSIX semaphores may also be exchanged via shared memory. However, named semaphores offer a more convenient API:[4]

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

Similar to `shm_open()`, `name` is required for naming the semaphore and should also start with "`/`", `oflag` and `mode` are used for options and permissions. `value` is the initial value if the semaphore is to be created.

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

`sem_close()` only closes a named semaphore for the current process, while it remains available in the system. `sem_unlink()` removes the name and frees the associated resources *after* all processes using it have closed it.

[4] On Linux, again `/dev/shm` is used, as described on Slide 9.

Example: Named Semaphores

```
// create named semaphore and initialize it to 1
sem_t *sem = sem_open("/semaphore",
                      O_CREAT,
                      S_IRUSR | S_IWUSR,
                      1);
assert(SEM_FAILED != sem);

sem_wait(sem)

/* ... */

sem_post(sem)

// either only close semaphore after usage...
sem_close(sem);

// ... or free it
// (after all processes using it have closed it)
sem_unlink("/semaphore");
```

Event-Based Concurrency

Introduction

By now, we have introduced two different ways of implementing *concurrent applications:*

1. Threads
2. Multi-programming with IPC

Another option, event driven programming , or event-based concurrency can be used for writing concurrent applications with a *single thread* of execution. It is often found in GUI programming and has also gained popularity in modern server frameworks (e.g. node.js). It possibly solves the following problems:

- Inherent *concurrency problems* like mutual exclusion, deadlocks etc.
- Having no control over *scheduling decisions*.

Main Idea: Event Loop I

The main idea of event-based concurrency is very simple:

1. Have an (endless) event loop .
2. *Wait for events to occur.*
3. When an event occurs, process it accordingly.

To process events, typically event handlers are used (e.g. callback functions). A basic event loop in pseudocode looks as follows:

```
while (1) {
  events = get_events();
  for (e in events)
    eventhandler.process(e);
}
```

⚠ When an event handler processes an event, it is the only activity taking place at any instant. Thus, choosing in which order to handle events is a form of *in-process scheduling!*

Main Idea: Event Loop II

The following figure depicts the principle of an event loop:

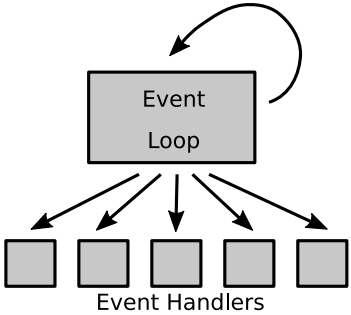

Figure: Event Loop Principle

Receiving Events

With an event loop, the question is *how to get notified about new events* (i.e. implement `get_events()`). The solution in general is I/O multiplexing : Instead of waiting for data, e.g. on a file descriptor (blocking the whole loop), it works as follows:

1. A list of relevant descriptors is passed to the OS.
2. Later (periodically), the OS is queried for changes in given descriptors.
3. The OS returns a list of changed descriptors.
4. The process reads/writes the changed descriptors etc.

The traditional API for I/O multiplexing is `select()` or `poll()`. *Modern interfaces* like `epoll()` (Linux) or `kqueue()` (BSD) address performance and other issues of those.

The `select()` Function I

`select()` works by passing different sets (bit-masks essentially) containing file descriptors of interest to the OS:

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

Three different sets can be passed:

`readfds`	Descriptors to be monitored for new data becoming available for reading.
`writefds`	Descriptors becoming ready to accept data written.
`exceptfds`	Descriptors to be monitored for exceptional conditions (e.g. TCP socket out-of-band data).

`nfds` has to be set to the highest-numbered descriptor in any set plus 1.

The `select()` Function II

File descriptor sets are manipulated using the macros `FD_ZERO()`, `FD_SET()`, `FD_CLR()` and `FD_ISSET()`:

```
// define two sets
fd_set rfds, wfds;

// initialize the sets (empty)
FD_ZERO(&rfds);
FD_ZERO(&wfds);

// add specific descriptors to a set
FD_SET(0, &rfds);  // stdin
FD_SET(1, &wfds);  // stdout

// nfds = highest fd + 1
int rc = select(2, &rfds, &wfds, NULL, NULL);

if (FD_ISSET(0, &rfds))
  printf("stdin is ready\n");

if (FD_ISSET(1, &wfds)) {
  printf("stdout is ready\n");
```

The `poll()` Function I

`poll()` works like `select()` but uses an array of `pollfd` structs, one per descriptor:

```
struct pollfd {
  int   fd;         // file descriptor
  short events;     // requested events
  short revents;    // returned events
};

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

Different constants for `events` and `revents` are provided, e.g. `POLLIN` (there is data to read) and `POLLOUT` (writing is now possible). See the man page for details.

The `poll()` Function II

The example from Slide 19 ff. using `poll()`:

```
// array of descriptors and events to monitor
struct pollfd fds[] = {
  {0, POLLIN, 0},
  {1, POLLOUT, 0}
};

// poll without timeout
poll(fds, 2, 0);

// read out the returned events from revents field

if (fds[0].revents == POLLIN)
  printf("stdin is ready\n");

if (fds[1].revents == POLLOUT)
  printf("stdout is ready\n");
```

Synchronous and Asynchronous APIs

In general, two types of APIs can be distinguished: synchronous and asynchronous.

Definition

A synchronous API / interface is *blocking*. It performs all of its work *before returning* to the caller.

An asynchronous API / interface is *non-blocking*. It *returns immediately* to the caller and performs its work in the background.

⚠ Typically, some form of I/O is the reason for blocking. *With event-based concurrency, only asynchronous APIs must be used!* Note that `select()`, `poll()` etc. are synchronous APIs!

Problem: Blocking Syscalls

Having only a single thread/process simplifies development: None of the common concurrency problems can occur. However, with event-based concurrency, other problems arise:

- *An important problem are synchronous syscalls:* Any syscall which blocks while being used in an event handler, blocks the whole event loop![5]
  Classical I/O APIs as well as any other type of blocking syscall are not allowed in this context. See Slide 28 ff. for a discussion of asynchronous I/O.
- *Long (i.e. CPU-intensive) computations are typically a problem as well:* For good performance, event handlers must return fast.

[5] NB: One of the obvious reasons for the earlier introduction of threads was to solve this kind of problems…

Problem: State Management

In event-based concurrency, managing state is more complicated. Compare the hypothetical flow in a normal application:

```
int rc = read(fd1, buffer1, size1);
rc = write(fd2, buffer1, size1);
rc = read(fd3, buffer2, size2);
rc = write(fd4, buffer2, size2);
```

...to an event-based version:

```
rc = read(fd1, buffer1, size1);
rc = read(fd3, buffer2, size2);
some_op();
// ...
rc = write(fd2, buffer1, size1);
some_other_op();
// ...
rc = write(fd4, buffer2, size2);
even_more_ops();
// ...
```

Other Difficulties

Besides blocking syscalls and state management, more difficulties are prevalent in event-based concurrency. Amongst them:

- Scalability
  A single event loop on a single CPU is fine. However, to scale and increase efficiency, support for *multiple cores* is required. For this, event handlers must be run on different cores – the return of classical concurrency problems!
- Implicit blocking
  Issues may arise due to *external activity* on the system. If for example a page fault occurs in an event-handler, the whole process is blocked implicitly.
- Maintainability
  Increased code and *API complexity* may make event-driven code more difficult to understand and maintain.

Asynchronous I/O (AIO)

Asynchronous I/O

Different APIs for Asynchronous I/O (AIO) offer the possibility to have true, non-blocking I/O running in the background. *I/O operations are scheduled* by the process and the OS will notify it upon their completion/status change.

In POSIX AIO , so-called *control blocks* describe all I/O requests:

```
struct aiocb {
  /* The order of these fields is implementation-dependent */

  int             aio_fildes;     /* File descriptor */
  off_t           aio_offset;     /* File offset */
  volatile void  *aio_buf;        /* Location of buffer */
  size_t          aio_nbytes;     /* Length of transfer */
  int             aio_reqprio;    /* Request priority */
  struct sigevent aio_sigevent;   /* Notification method */
  int             aio_lio_opcode; /* Operation to be performed;
                                     lio_listio() only */

  /* Various implementation-internal fields not shown */
};
```

Source: `man aio`

Common POSIX AIO Functions

Here are the most common POSIX AIO functions:

```
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
```

Operations for asynchronous reads and writes. Analogous to the classic **`read()`** and **`write()`** functions.

```
int aio_error(const struct aiocb *aiocbp);
```

Returns the current status of an operation, e.g. ongoing (**`EINPROGRESS`**) or completed (**`0`**).

```
int aio_suspend(const struct aiocb * const aiocb_list[],
                int nitems, const struct timespec *timeout);
```

Can be used to block the caller until one of the given AIO operations has completed or the timeout has been reached.

👉 See **`man aio`** for more details.

POSIX AIO Notifications

An important question when performing I/O in the background is how notification for completion or status changes works. In POSIX AIO, different options exist:

- AIO signaling
  Using the `aio_sigevent` field in `aiocb`, notification via thread or signal (or no notification at all) can be chosen.[6]
- Polling
  Polling may be implemented by verifying the return value of `aio_error()`, to see if an operation is still in progress.
- Suspending
  Using `aio_suspend()`, a process may explicitly wait for a status change.

[6] ☞ Signals are a powerful UNIX idea not treated further in this course. Invest some time (e.g. by reading "`man 7 signal`") to familiarize yourself!

AIO in Practice

Asynchronous I/O is complicated in practice. The APIs are not straightforward to use and the resulting code is in general complex, due to state management (cf. Slide 25) and other issues. Because of complexity and other deficiencies, often alternatives to AIO are considered.

One possible solution is to combine different approaches:

- Have an event loop to handle the overall concurrency (e.g. network requests).
- Use threads (possibly also a *thread pool*) to handle I/O using classical I/O mechanisms.

Reinventing the Wheel (Not)

As we have seen, implementing event-based concurrency has its own complexities. Due to this, event loops are rarely written from scratch using I/O multiplexing and asynchronous I/O APIs.

👉 Instead of reinventing the wheel, it makes sense to use libraries which abstract the basic event loop implementation.

Examples include:

- `libevent`, e.g. used by Google Chrome.
- `libuv`, primarily developed for Node.js, but also used by others.

Appendix

Bibliography I

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.