CSC4508 – Operating Systems

François Trahay & Gaël Thomas

Institut Mines-Télécom

TELECOM SudParis

2021
Objectives of the class:
■ Understand the internals of operating systems
■ Know how to interact with the OS from a program

Structure of the class:
[U] “userland” oriented sessions
[K] “kernel” oriented sessions
[G] “more general” sessions

# 3

1.1 Organization
1. Processes
   CI1 [U] Threads
   CI2 [K] Concurrency programming
   CI3 [K] Multiprogramming
   CI4 [K] Scheduling
   CI5 [K] Interruption and scheduling
   CI6 [K] Sprints: Finalization of the scheduler

2. Memory
   CI7 [U] Virtual memory
   CI8 [K] Memory Management Unit
   CI9 [G] Architecture
   CI10 [K] Sprints

3. Input/Output
   CI11 [U] Input/Output
   CI12 [U] Synthesis: mini-project
   CI13 [K] File systems
   CI14 [K] Sprints
   CI15 Exam (lab)
1.2 Kernel sessions: XV6

During the [K] sessions, you will develop an OS
■ Based on the xv6 OS
■ Development of new OS mechanisms
■ Sprint sessions:
  ♦ Finalization of development
  ♦ Evaluation by teachers

1.3 Evaluation

Evaluation:
■ 20% - Continuous assessment during sprints:
  ♦ "how did you implement this mechanism of the OS?"
  ♦ "what happens if X?"
■ 80% - Graded lab exams with several parts:
  ♦ Course question(s)
  ♦ Explain how you implemented an OS mechanism
  ♦ Develop an application

1.4 Evaluation of the class

■ At the end of the class, students evaluate the class.
■ Objective: improve the class
1. Execution context of a process

- **Context**: execution context + kernel context
- **Address space**: code, data and stack

## Execution context

- **Data registers**
- **Status register**
- **Stack pointer (SP)**
- **Program counter (PC)**

## Kernel context

- **Virtual memory structures**
- **Descriptor table**
- **Brk pointer**

## Process context

- **Heap**
- **Libs**
- **Shared libraries**
- **Dynamic allocation** (malloc, ...)
- **Uninitialized variables**
- **Initialized variables**
- **Instructions**

## Resources

- **Execution flow**: execution context + stack
- **Execution flow (or thread)**: execution context + stack
- **Resources**: code, data, kernel context

1.1 Execution flows
In a multi-threaded process, each thread has a context (registers + stack). The rest of the memory (code, data, etc.) and resources (open files, etc.) are shared between threads.

The stacks of the different threads are located in memory so that they can grow. However, if a thread’s stack grows too much, it might overflow onto the stack of another thread. To avoid this problem, the size of the stack is limited (the command `ulimit -s` gives the maximum stack size). This size limit can be changed using command line (by example `ulimit -s 32768`), or from a program (in using the `setrlimit` function).

We present here the Pthread API (POSIX thread) which is the most used in C. The C11 standard defines another interface for manipulating threads. However, there are only few implementations of this interface. The de facto standard therefore remains Pthread.

Unlike the creation of processes which generates a hierarchy (i.e. each process has a parent process), there is no hierarchy between threads.

Technically, all the memory space is shared between the threads. It is therefore possible to share all the variables, including local variables.
2.1 Thread-safe source code

Thread-safe source code: gives a correct result when executed simultaneously by multiple threads:
- No call to non-thread-safe code
- Protect access to shared data

2.2 Reentrant source code

Reentrant source code: code whose result does not depend on a previous state:
- Do not maintain a persistent state between calls
- Example of a non-reentrant function: `fread` depends on the position of the stream cursor.

Example: `strtok`.

Another example of a non-reentrant function is the `char *strtok (char * str, char * delim)` function. This function extracts substrings from a string.

For example, the following code displays the different directories of the `PATH` variable:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
    char *string = getenv("PATH");
    printf("Parsing '%s'
", string);
    for (char *token = strtok(string, ":") ; token ;
        token = strtok(NULL, ":") ){
        printf("%s
", token);
    }
}

int main(int argc, char **argv) {
    extract_path();
    return 0;
}
```

Here is an example of result obtained with this program:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
/usr/local/bin
/usr/bin
/bin
/usr/local/games
/usr/games
```

The `strtok` function is not reentrant because it is based on a previous state (a pointer to the last character tested in the string). Thus, in this example, the processing applied to each token cannot use `strtok`. For example:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
    char *string = getenv("PATH");
    printf("Parsing '%s'
", string);
    for (char *token = strtok(string, ":") ;
        token ;
        token = strtok(NULL, ":") ){
        printf("%s contains: 
", token);
        for (char *word = strtok(token, "/") ;
            word ;
            word = strtok(NULL, "/") ){
            printf("%s ", word);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {
    extract_path();
    return 0;
}
```

Will result in:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
/usr/local/bin contains: usr local bin
```

Here the first token `/usr/local/bin` is split into words `usr local bin` by successive calls to `strtok` which modify the previous state of `strtok`, which prevents subsequent calls to `tokens = strtok (NULL, ":")` to iterate over the string.

Making a function reentrant. It is possible to make a non-reentrant function reentrant by adding a parameter corresponding to the state of the function. For example, the reentrant version of `char *strtok(char * str, char * delim);` is `char *strtok_r(char * str, char * delim, char ** saveptr );`.

Thus, the previous program can be corrected:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
    char *string = getenv("PATH");
    printf("Parsing '%s'
", string);
    for (char *token = strtok(string, ":") ;
        token ;
        token = strtok(NULL, ":") ){
        printf("%s
", token);
    }
}

int main(int argc, char **argv) {
    extract_path();
    return 0;
}
```

Here is an example of result obtained with this program:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
```

The previous program can be corrected:
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
    char *string = getenv("PATH");
    char *saveptr = NULL;
    printf("Parsing '%s'
", string);
    for (char *token = strtok_r(string, ":", &saveptr); token; 
         token = strtok_r(NULL, ":", &saveptr)) {
        printf("%s contains: ", token);
        char *saveptr_word = NULL;
        for (char *word = strtok_r(token, "/ ", &saveptr_word); 
             word; word = strtok_r(NULL, "/ ", &saveptr_word)) {
            printf("%s ", word);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {
    extract_path();
    return 0;
}
```

Which will result in:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
    /usr/local/bin contains:  
    /usr/bin contains:  
    /bin contains:  
    /usr/local/games contains:  
    /usr/games contains:  
```

---

## 2.3 TLS — Thread-Local Storage

**Global variable (or static local) specific to each thread**
- Examples:
  - `errno`
- Declaring a TLS variable
  - in C11: `_Thread_local int variable = 0;`
  - in C99 with Visual studio: `__declspec(thread) int variable = 0;`

**Another way (more portable, but much more painful to write) to declare a TLS variable** is to use a `pthread_key`:
- Creation:
  ```c
  int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
  ```
- Termination:
  ```c
  int pthread_key_delete(pthread_key_t *key);)
  ```
- Usage:
  ```c
  void *pthread_getspecific(pthread_key_t key);
  int pthread_setspecific(pthread_key_t key, const void *value);
  ```

---

## 3 Synchronization

- Guarantees data consistency
  - Simultaneous access to a shared read / write variable
- Several synchronization mechanisms exist
  - Mutex
  - Atomic instructions
  - Conditions, semaphores, etc. (see Lecture 03)

The following program illustrates the problem of simultaneous access to shared variables. Here, two threads each increment 1000000000 times the same variable:

```c
The following program illustrates the problem of simultaneous access to shared variables. Here, two threads each increment 1000000000 times the same variable:
```

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 — Operating Systems
#include <error.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

/* INT_MAX / 2 */
#define NBITER 1000000000

int counter = 0;

void* start_routine(void* arg) {
  int i;
  for (i = 0; i < NBITER; i++) {
    /* OOPS : WRONG ! Access to an unprotected shared variable */
    counter ++;
  }
  pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
  int rc;
  pthread_t thread1, thread2;
  rc = pthread_create(&thread1, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");
  rc = pthread_create(&thread2, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");
  rc = pthread_join(thread1, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  rc = pthread_join(thread2, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  if (counter != 2 * NBITER)
    printf("BOOM! counter = %d\n", counter);
  else
    printf("OK counter = %d\n", counter);
  exit(EXIT_SUCCESS);
}

While the counter should be 2 \times 1000000000 = 2000000000, running this program gives another result, for example:

$ ./compteurBOOM
BOOM! compteur = 1076588402

Using a mutex, we can correct the BOOM counter program by ensuring that the counter increments are done in mutual exclusion:

#include <error.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

/* INT_MAX / 2 */
#define NBITER 1000000000

int counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* start_routine(void* arg) {
  int i;
  for (i = 0; i < NBITER; i++) {
    pthread_mutex_lock(&mutex);
    counter ++;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
  int rc;
  pthread_t thread1, thread2;
  rc = pthread_create(&thread1, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");
  rc = pthread_create(&thread2, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");
  rc = pthread_join(thread1, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  rc = pthread_join(thread2, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  if (counter != 2 * NBITER)
    printf("BOOM! counter = %d\n", counter);
  else
    printf("OK counter = %d\n", counter);
  exit(EXIT_SUCCESS);
}
While the result is correct, the use of a mutex significantly slows down the program (144s with mutex, against 4.1s without mutex).

We can fix the counterBOOM program by using atomic operations. To do this, all we have to do is declare the counter like `atomic int`. The counter increment then uses the atomic operation `atomic_fetch_add`.

Here, the result is correct and the program is much faster than when using a mutex:

- without synchronization: 4.1s
- with a mutex: 144s
- with an atomic operation: 35s

Bibliography
### 1 Introduction

Content of this lecture:
- discovering existing synchronization mechanisms
- inter-process synchronization
- intra-process synchronization
- studying classic synchronization patterns

### 2 Inter-process synchronisation

- IPC: Inter Process Communication
  - based on IPC objects in the OS
  - usage: usually via an entry in the filesystem
  - provide data persistence
2.1 Pipes

- Special files managed in FIFO
  - int pipe(int pipefd[2]);
    - creates a pipe accessible by the current process
    - also accessible to future child processes
  - pipefd[0] for reading, pipefd[1] for writing

- Named pipes
  - int mkfifo(const char *pathname, mode_t mode);
    - creates an entry in the filesystem accessible by any process
  - Use (almost) like a "regular" file
    - blocking reading
    - lseek is impossible

You have already handled pipes without necessarily realizing it: in bash, the sequence of commands linked by pipes is done via anonymous pipes created by the bash process. So when we run `cmd1 | cmd2 | cmd3`, bash creates 2 anonymous pipes and 3 processes, then redirects (thanks to the `dup2` system call, see Lecture #11) standard and outputs of processes to the different tubes.

2.2 Shared memory

- Allows you to share certain memory pages between several processes
- Creating a zero-byte shared memory segment:
  - int shm_open(const char *name, int oflag, mode_t mode);
    - name is a key of the form "/key"
- Changing the segment size:
  - int ftruncate(int fd, off_t length);
- Mapping the segment into memory:
  - void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
    - flags must contain MAP_SHARED

We will see later (during lecture 11 on I/O) another use of mmap.

2.3 Semaphore

- Object consisting of a value and a waiting queue
- Creating a semaphore:
  - named semaphore: sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
    - name is a key of the form "/key"
  - anonymous semaphore: int sem_init(sem_t *sem, int pshared, unsigned int value);
    - if pshared != 0, can be used by several processes (using a shared memory segment)
- Usage:
  - int sem_wait(sem_t *sem);
  - int sem_trywait(sem_t *sem);
  - int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
  - int sem_post(sem_t *sem);

3 Intra-process synchronization

- Based on shared objects in memory
- Possible use of IPC
Concurrent programming 3 Intra-process synchronisation

3.1 Mutex

- Ensures mutual exclusion
- Type: pthread_mutex_t
- Initialisation:
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `int pthread_mutex_init(pthread_mutex_t *m, const pthread_mutexattr_t *attr);`
- Usage:
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- Destroying a mutex:
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Inter-process mutex

It is possible to synchronize threads from several processes with a pthread_mutex_t if it is in a shared memory area. For this, it is necessary to position the `PTHREAD_PROCESS_SHARED` attribute of the mutex with the function `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`

3.2 Monitors

- Allows you to wait for a condition to occur
- Consists of a mutex and a condition
- Example:
  ```
  pthread_mutex_lock(&l);
  while (!condition) {
    pthread_cond_wait(&c, &l);
  }
  process_data();
  pthread_mutex_unlock(&l);
  pthread_mutex_lock(&l);
  produce_data();
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&l);
  ```

Here are the prototypes of the functions associated with the conditions:

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
- `int pthread_cond_destroy(pthread_cond_t *cond);`
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`

The mutex ensures that between testing for the condition (`while (! condition)` and wait (`pthread_cond_wait()`), no thread performs the condition.

Inter-process monitors

To synchronize multiple processes with a monitor, it is necessary to set the following attributes:

- The attribute `PTHREAD_MUTEX_SHARED` of the mutex (using `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`)
- The attribute `PTHREAD_PROCESS_SHARED` of the condition (using `int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);`)

3.3 Barrier

- Allows you to wait for a set of threads to reach rendez-vous point
- Initialisation:
  - `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count);`
- Waiting:
  - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  - `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
- Example:
  ```
  pthread_barrier_init(&barrier, NULL, 2);
  pthread_mutex_lock(&l);
  while (!condition) {
    pthread_cond_wait(&c, &l);
  }
  process_data();
  pthread_mutex_unlock(&l);
  ```

Once all the threads have reached the barrier, they are all unblocked and `pthread_barrier_wait` returns 0 except for one thread which returns `PTHREAD_BARRIER_SERIAL_THREAD`.

Inter-process barrier

To synchronize threads from multiple processes with a barrier, it is necessary to set the attribute `PTHREAD_PROCESS_SHARED` with `int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);`
Concurrent programming 4 Classic synchronization patterns

3.3.1 Read-Write lock

- Type: `pthread_rwlock_t`
- `int pthread_rwlock_rdlock(pthread_rwlock_t* lock)`
  - Lock in read mode
  - Possibility of several concurrent readers
- `int pthread_rwlock_wrlock(pthread_rwlock_t* lock)`
  - Lock in write mode
  - Mutual exclusion with other writers and readers
- `int pthread_rwlock_unlock(pthread_rwlock_t* lock)`
  - Release the lock

4 Classic synchronization patterns

4 Classic synchronization patterns

In the literature, these problems are usually solved by using semaphores. This is because these problems have been theorized in the 1960s and 1970s by Dijkstra based on semaphores. In addition, semaphores have the advantage of being able to be used for inter-process synchronizations or intra-process.

However, modern operating systems implement many synchronization primitives which are much more efficient than semaphores. In the next slides, we will therefore rely on these mechanisms rather than semaphores.

4.1 Mutual exclusion synchronization pattern

- Allows concurrent access to a shared resource
- **Principle:**
  - A mutex is initialized
  - **Primitive** `mutex_lock(m)` at the start of the critical section
  - **Primitive** `mutex_unlock(m)` at the end of the critical section
- **Example:**
  - `mutex m` initialized
  - `prog1`, `prog2`
  - `mutex_lock(m)`
  - `x = x + 10` and `write(account=x)`
  - `mutex_unlock(m)`

Intra-process implementation

In a multi-threaded process, we just need to use a mutex of type `pthread_mutex_t`.

Inter-process implementation

To implement a mutual exclusion between several processes, several solutions exist:

- using a `pthread_mutex_t` in a shared memory segment between processes. For this, it is necessary to set the attribute `PTHREAD_MUTEX_SHARED` in the mutex (using `pthread_mutexattr_setpshared`);
- using a semaphore initialized to 1. The entry in section critical is protected by `sem_wait`, and we call `sem_post` when leaving the critical section.

4.2 Cohort synchronization pattern

- Allows the cooperation of a group of a given maximum size
- **Principle:**
  - A counter initialized to `N` and a monitor `m` to protect the counter
  - Decrement the counter at the start when needing a resource
  - Increment the counter at the end when releasing the resource

Intra-process implementation

- `mutex_lock(m)`
- `while(cpt == 0) cond_wait(m);`
- `mutex_unlock(m)`
- `cond_signal(m);`
- `mutex_unlock(m)`

Inter-process implementation

- `mutex_lock(m)`
- `while(cpt == 0) cond_wait(m);`
- `mutex_unlock(m)`
- `sem_wait();`
- `sem_post();`
- `cond_signal(m);`
- `mutex_unlock(m)`
4.3 Producer / Consumer synchronization pattern

- One or more threads produce data.
- One or more threads consume the data produced.
- Communication via a N-blocks buffer.

1. Executing Producer: produces info0

2. Executing Producer: produces info1

3. Executing Consumer: consumes info0

4. Executing Producer: produces info2

4.3.1 Implementation of a Producer / Consumer pattern

- A available_spots monitor initialized to N.
- A ready_info monitor initialized to 0.

Producer

- repeat
- ... ...
- mutex_lock(available_spots); mutex_lock(ready_info);
- while(available_spots<=0) while(ready_info<=0)
- cond_wait(available_spots); cond_wait(ready_info);
- reserve_slot(); extract(info)
- mutex_unlock(available_spots); mutex_unlock(ready_info);
- calcul(info) mutex_lock(available_spots);
- free_slot();
- mutex_lock(ready_info); cond_signal(available_spots)
- push(info); mutex_unlock(available_spots);
- cond_signal(ready_info);
- mutex_unlock(ready_info); ...
- endRepeat
- ...

Consumer

Inter-process Producer / Consumer

It is of course possible to implement a producer / consumer scheme between processes using conditions and monitors. Another simple solution is to use a pipe: since writing in a pipe being atomic, the deposit of a data boils down to writing into the pipe, and reading from the pipe extracts the data.

4.4 Reader / Writer pattern

- Allow a coherent competition between two types of process:
  - the "readers" can simultaneously access the resource
  - the "writers" access the resource in mutual exclusion with other readers and writers

4.4.1 Implementation of a Reader / Writer synchronization pattern

- Use a pthread_rwlock_t
  - int pthread_rwlock_rdlock(pthread_rwlock_t* lock) to protect read operations
  - int pthread_rwlock_wrlock(pthread_rwlock_t* lock) to protect write operations
  - int pthread_rwlock_unlock(pthread_rwlock_t* lock) to release the lock

Implementation with a mutex

It is possible to implement the reader / writer synchronization pattern using a mutex instead of rwlock: read and write operations are protected by a mutex. However, this implementation does not allow multiple readers to work in parallel.

Implementation with a monitor

The implementation of the monitor-based reader / writer is more complex. It mainly requires:

- an integer readers which counts the number of threads reading
- a boolean writing which indicates that a thread is writing
- a cond condition to notify changes to these variables
4 Classic synchronization patterns 4.4 Reader / Writer pattern

- a mutex mutex to protect concurrent access

Here is an implementation of the reader / writer using a monitor:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

// This program simulates operations on a set of bank accounts
// Two kinds of operations are available:
// - read operation : compute the global balance (i.e. the sum of all accounts)
// - write operation : transfer money from one account to another

// Here's an example of the program output:

// $ ./rw_threads_condition
// Balance : 0 (expected : 0)
// 3982358 operation, including:
// 3581969 read operations (89.945932 %)
// 400389 write operations (10.054068 %)

#define N 200

int n_loops = 1000000;
int accounts[N];
int nb_read = 0;
int nb_write = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int readers = 0;
int writing = 0;

/* read all the accounts */
int read_accounts() {
    pthread_mutex_lock(&mutex);
    while (writing) {
        pthread_cond_wait(&cond, &mutex);
    }
    readers++;
    pthread_mutex_unlock(&mutex);
    nb_read++;
    int sum = 0;
    for (int i=0; i<N; i++) {
        sum += accounts[i];
    }
    pthread_mutex_lock(&mutex);
    readers--;
    if (!readers) {
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mutex);
    return sum;
}

/* transfer amount units from account src to account dest */
void transfer(int src, int dest, int amount) {
    pthread_mutex_lock(&mutex);
    while (writing || readers) {
        pthread_cond_wait(&cond, &mutex);
    }
    writing = 1;
    pthread_mutex_unlock(&mutex);
    nb_write++;
    accounts[dest] += amount;
    accounts[src] -= amount;
    pthread_mutex_lock(&mutex);
    writing = 0;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

void * thread_function(void *arg) {
    for (int i=0; i<n_loops; i++) {
        /* randomly perform an operation */
        /* threshold sets the proportion of read operation.
         * here, 90% of all the operations are read operation
         * and 10% are write operations */
        int threshold = 90;
        int x = rand()%100;
        if (x < threshold) {
            /* read */
            int balance = read_accounts();
            if (balance != 0) {
                fprintf(stderr, "Error : balance = %d !\n", balance);
                abort();
            }
        } else {
            /* write */
            int src = rand()%N;
            int dest = rand()%N;
            int amount = rand()%100;
            transfer(src, dest, amount);
        }
    }
    return NULL;
}

int main(int argc, char **argv) {
    for (int i=0; i<N; i++) {
        accounts[i] = 0;
    }
    int nthreads=4;
    pthread_t tid[nthreads];
    for (int i=0; i<nthreads; i++) {
        pthread_create(&tid[i], NULL, thread_function, NULL);
    }
    for (int i=0; i<nthreads; i++) {
        pthread_join(tid[i], NULL);
    }
    int balance = read_accounts();
    printf("Balance: %d (expected: 0)\n", balance);
    int nb_op = nb_read+nb_write;
    printf("%d operation, including:\n", nb_op);
    printf("\t%d read operations (%f %%%)\n", nb_read, 100.*nb_read/nb_op);
    printf("\t%d write operations (%f %%%)\n", nb_write, 100.*nb_write/nb_op);
    return EXIT_SUCCESS;
}
```

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 – Operating Systems
Synchronization

François Trahay

CSC4508 – Operating Systems

2020–2021

1 Introduction

Objectives of this lecture:
■ How are synchronization primitives implemented?
■ How to do without locks?

2 Introduction

Outlines

- 1 Introduction
- 2 Atomic operations
- 3 Synchronization primitives
- 4 Using synchronization

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 – Operating Systems
2 Atomic operations

2.1 Motivation

- By default, an instruction modifying a variable is not atomic.
- Example: `x++`
gives:

  1. Register = load(x)
  2. Register ++
  3. x = store(register)

→ Problem if the variable is modified by another thread concurrently.

C11 provides a set of atomic operations, including:

- `atomic_flag_test_and_set`
- `atomic_compare_exchange_strong`
- `atomic_fetch_add`
- `atomic_thread_fence`

2.2 Atomic operations

Test and set

- _Bool atomic_flag_test_and_set(volatile atomic_flag* obj)
  - Sets a flag and returns its previous value.

Performs atomically:

```c
int atomic_flag_test_and_set(int* flag) {
    int old = *flag;
    *flag = 1;
    return old;
}
```

Implementing a lock:

```c
thread lock:
void lock(int* lock) {
    while (atomic_flag_test_and_set(lock) == 1);
}
```

Here is an example of a program using a `test_and_set` based lock:

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

volatile int lock=0;
int x = 0;
```
Here is an example of a program that may suffer from overly aggressive optimization by the compiler:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

# ifdef USE_VOLATILE
volatile int a = 0;
# else
int a = 0;
# endif

void* thread1(void* arg) {
    while (a == 0) ;
    printf("Hello\n");
    return NULL;
}

void* thread2(void* arg) {
    a = 1;
    return NULL;
}

int main(int argc, char** argv) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return EXIT_SUCCESS;
}
```

When compiled with the optimization level `-O0` (i.e. without any optimization), thread1 spins waiting, and when thread2 modifies the variable `a`, it unlocks thread1 which displays "Hello ":

```
$ gcc -o volatile volatile.c -Wall -pthread -O0
$ ./volatile
Hello
```

When compiled with the optimization level `-O1`, the generated code no longer works:

```
$ gcc -o volatile volatile.c -Wall -pthread -O1
$ ./volatile
^C
```

Analyzing the code generated by the compiler reveals the problem:

```
$ gcc -o volatile volatile.c -Wall -pthread -O2
$ gdb ./volatile
[...]
(gdb) disassemble thread1
Dump of assembler code for function thread1:
0x00000000000011c0 <+0>: mov 0x2e7e(%rip),%eax # 0x4044 <a>
0x00000000000011c6 <+6>: test %eax,%eax
0x00000000000011c8 <+8>: jne 0x11d0 <thread1+16>
0x00000000000011ca <+10>: jmp 0x11ca <thread1+10>
0x00000000000011cc <+12>: nopl 0x0(%rax)
0x00000000000011d0 <+16>: sub $0x8,%rsp
0x00000000000011d4 <+20>: lea 0xe29(%rip),%rdi # 0x2004
0x00000000000011db <+27>: callq 0x1040 <puts@plt>
0x00000000000011e0 <+32>: xor %eax,%eax
0x00000000000011e2 <+34>: add $0x8,%rsp
0x00000000000011e6 <+38>: retq
```

When compiled with the optimization level `-O2`, the generated code no longer works:

```
$ gcc -o volatile volatile.c -Wall -pthread -O2
$ gdb ./volatile
[...]
```

When compiled with the optimization level `-O3`, the generated code no longer works:

```
$ gcc -o volatile volatile.c -Wall -pthread -O3
$ gdb ./volatile
[...]
```

Analyzing the code generated by the compiler reveals the problem:

```
$ gcc -o volatile volatile.c -Wall -pthread -O3
$ gdb ./volatile
[...]
```

This example demonstrates how volatile can be used to prevent the compiler from optimizing code in a way that leads to race conditions.
We see here that at the address 0x11ca, the program jumps to the address 0x00000000000011e4. So it jumps in place indefinitely. This is explained by the fact that the variable is not volatile. The compiler therefore thinks it can optimize access to this variable: since the thread1 function only accesses the variable in read-mode, the program loads the variable in a register (here, the eax register). When thread2 modifies the variable a, the modification is therefore not perceived by thread1!

Declaring the variable as volatile forces the compiler to read the variable each time:

```c
gcc -o volatile volatile.c -Wall -pthread -O2 -DUSE_VOLATILE=1
```

Here, the loop variable `a` is translated to the base from `a` to `a+1`. At each loop iteration, the value of `a` is loaded, then tested.

---

### 2.5 Compare And Swap (CAS)

![Diagram of Compare And Swap (CAS)](image)

#### Perform atomically:

```
bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);  
```

- `obj`: The object to compare.
- `expected`: The expected value.
- `desired`: The desired value.

- `obj` is loaded, then tested.
- `expected` is loaded, then tested.
- If both `obj` and `expected` equal `desired`, return `true`.
- If both `obj` and `expected` do not equal `desired`, return `false`.

---

Here is an example of a program handling an lock-free list thanks to compare_and_swap:

```c
int compare_and_swap(int address, int desired, int expected) {
    int obj = *(volatile int*) address;
    int expected_value = *(volatile int*) expected;
    if (obj == desired && expected_value == expected) {
        *(volatile int*) address = desired;
        return true;
    } else {
        return false;
    }
}
```
return NULL;
}

int main(int argc, char **argv) {
    pthread_t tids[NTHREADS];
    for (int i = 0; i < NTHREADS; i++) {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }
    for (int i = 0; i < NTHREADS; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("sum = %d\n", sum);
    return EXIT_SUCCESS;
}

2.6 Fetch and Add

- C atomic_fetch_add volatile A* obj, M arg;
- replace obj with arg+obj
- return the old value of obj

Performs atomically:

```c
int fetch_and_add(int *obj, int value) {
    int old = *obj;
    *obj = old + value;
    return old;
}
```

Here is an example of a program using fetch_and_add to atomically increment a variable:

```
fetched_add.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

volatile int x = 0;

// ifdef NOT_THREAD_SAFE
/* thread - unsafe version */
void inc(volatile int *obj) {
    *obj = (*obj) + 1;
}
// else
/* thread - safe version */
void inc(volatile int *obj) {
    atomic_fetch_add(obj, 1);
}
// endif /* NOT_THREAD_SAFE */

void *thread_function(void *arg) {
    for (int i = 0; i < NITER; i++) {
        inc(&x);
    }
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t tids[NTHREADS];
    for (int i = 0; i < NTHREADS; i++) {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }
    for (int i = 0; i < NTHREADS; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("x = %d\n", x);
    return EXIT_SUCCESS;
}
```

2.7 Memory Fence (Barrière mémoire)

- C atomic_thread_fence(memory_order order);
- performs a memory synchronization
- ensures that all past memory operations are "visible" by all threads according to the memory model chosen (see C11 memory model)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

volatile int x = 0;

// ifdef NOT_THREAD_SAFE
/* thread - unsafe version */
void inc(volatile int *obj) {
    *obj = (*obj) + 1;
}
// else
/* thread - safe version */
void inc(volatile int *obj) {
    atomic_fetch_add(obj, 1);
}
// endif /* NOT_THREAD_SAFE */

void *thread_function(void *arg) {
    for (int i = 0; i < NITER; i++) {
        inc(*arg);
    }
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t tids[NTHREADS];
    for (int i = 0; i < NTHREADS; i++) {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }
    for (int i = 0; i < NTHREADS; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("x = %d\n", x);
    return EXIT_SUCCESS;
}
```
3 Synchronization primitives

Properties to consider when choosing a synchronization primitive:

- **Reactivity**: time spent between the release of a lock and the unblocking of a thread waiting for this lock.
- **Contention**: memory traffic generated by threads waiting for a lock.
- **Equity and risk of famine**: if several threads are waiting for a lock, do they all have the same probability of acquiring it? Are some threads likely to wait indefinitely?

3.1 Busy-waiting synchronization

- **int pthread_spinlock(pthread_spinlock_t *lock);**
  - Tests the value of the lock until it becomes free, then acquires the lock.
- **Benefits**:
  - Simple to implement (with `test_and_set`).
- **Disadvantages**:
  - Consumes CPU while waiting.
  - Consumes memory bandwidth while waiting.

It is also possible to implement a spinlock using an atomic operation:

```c
// spin_lock.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>
#include <assert.h>

#define NITER 1000000
#define NTHREADS 4

struct lock {
  /* if flag = 0, the lock is available */
  volatile int flag;
};
typedef struct lock lock_t;

void lock(lock_t *l) {
  /* try to set flag to 1. If the flag is already 1, loop and try again */
  while (atomic_flag_test_and_set(&l->flag)) ;
}

void unlock(lock_t *l) {
  l->flag = 0;
}

void lock_init(lock_t *l) {
  l->flag = 0;
}

lock_t l;
int x;

void* thread_function(void* arg) {
  for (int i = 0; i < NITER; i++) {
    lock(&l);
    x++;
    unlock(&l);
  }
  return NULL;
}

int main(int argc, char** argv) {
  lock_init(&l);
  pthread_t tids[NTHREADS];
  int ret;
  for (int i = 0; i < NTHREADS; i++) {
    ret = pthread_create(&tids[i], NULL, thread_function, NULL);
    assert(ret == 0);
  }
  for (int i = 0; i < NTHREADS; i++) {
    ret = pthread_join(tids[i], NULL);
    assert(ret == 0);
  }
  printf("x = %d\n", x);
  printf("expected: %d\n", NTHREADS * NITER);
  return EXIT_SUCCESS;
}
```
3.2 Futex

Fast Userspace Mutex
- System call allowing to build synchronization mechanisms in userland
- Allows waiting without monopolizing the CPU
- A futex is made up of:
  - a value
  - a waiting list
- Available operations (among others)
  - `WAIT(int *addr, int value)`
    - while(*addr == value) sleep();
    - add the current thread to the waiting list
  - `WAKE(int *addr, int value, int num)`
    - *addr = value
    - wake up num threads waiting on addr

3.3 Implementing a mutex using a futex

- mutex: an integer with two possible values: 1 (unlocked), or 0 (locked)
- `mutex_lock(m)`
  - Test and unset the mutex
  - if mutex is 0, call `FUTEX_WAIT`
- `mutex_unlock(m)`
  - Test and set the mutex
  - call `FUTEX_WAKE` to wake up a thread from the waiting list

Here is an example of a program implementing a mutex using futex:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>
#include <linux/futex.h>
#include <sys/time.h>
#include <sys/syscall.h>
#include <errno.h>
#include <assert.h>
#define NITER 1000000
#define NTHREADS 4

struct lock {
    int flag;
};
typedef struct lock lock_t;

static int futex(int *uaddr, int futex_op, int val, const struct timespec *timeout, int *uaddr2, int val3) {
    return syscall(SYS_futex, uaddr, futex_op, val, timeout, uaddr, val3);
}

void lock(lock_t *l) {
    while (1) {
        /* Is the futex available? */
        int expected = 1;
        if (atomic_compare_exchange_strong(&l->flag, &expected, 0))
            return;
        /* Yes */
        /* Futex is not available; wait */
        int s = futex(&l->flag, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN) {
            perror("futex_wait failed");
            abort();
        }
    }
}

void unlock(lock_t *l) {
    int expected = 0;
    atomic_compare_exchange_strong(&l->flag, &expected, 1);
    int s = futex(&l->flag, FUTEX_WAKE, 1, NULL, NULL, 0);
    if (s == -1) {
        perror("futex_wake failed");
        abort();
    }
}

void lock_init(lock_t *l) {
    l->flag = 1;
}

lock_t l;
int x;
void *thread_function(void *arg) {
    for (int i=0; i<NITER; i++) {
        lock(&l);
        x++;
        unlock(&l);
    }
    return NULL;
}

int main(int argc, char **argv) {
    lock_init(&l);
    pthread_t tids[NTHREADS];
    int ret;
    for (int i = 0; i < NTHREADS; i++) {
        ret = pthread_create(&tids[i], NULL, thread_function, NULL);
        assert(ret == 0);
    }
    for (int i = 0; i < NTHREADS; i++) {
        ret = pthread_join(tids[i], NULL);
        assert(ret == 0);
    }
    printf("x = %d\n", x);
    printf("expected: %d\n", NTHREADS*NITER);
    return EXIT_SUCCESS;
}
```
3.4 Implementing a monitor using a futex

Here is an example of a program implementing a condition using futex:

```c
condition: a counter
void cond_init(cond_t *c) {
    c->cpt = 0;
}
void cond_wait(cond_t *c, pthread_mutex_t *m) {
    pthread_mutex_lock(m);
    if (atomic_load(&c->cpt) > 0) {
        atomic_fetch_add(&c->cpt, -1);
        pthread_mutex_unlock(m);
        return;
    }
    pthread_mutex_lock(m);
    int value = atomic_load(&c->value);
    if (value > 0) {
        atomic_fetch_add(&c->value, -1);
        futex(&c->value, FUTEX_WAKE, 0);
        pthread_mutex_unlock(m);
        return;
    }
    pthread_mutex_lock(m);
    int value = atomic_load(&c->value);
    if (value > 0) {
        atomic_fetch_add(&c->value, -1);
        futex(&c->value, FUTEX_WAKE, 0);
        pthread_mutex_lock(m);
        futex(&c->value, FUTEX_WAKE, 0);
        pthread_mutex_unlock(m);
        return;
    }
    atomic_fetch_add(&c->cpt, 1);
    pthread_mutex_lock(m);
    futex(&c->value, FUTEX_WAKE, 0);
    pthread_mutex_unlock(m);
    return;
}
void cond_signal(cond_t *c) {
    atomic_fetch_add(&c->cpt, 1);
    pthread_mutex_lock(m);
    futex(&c->value, FUTEX_WAKE, 0);
    pthread_mutex_unlock(m);
    return;
}
void init_monitor(monitor *m, int *prods, int *cons) {
    pthread_mutex_t mutex;
    pthread_mutex_init(&m->mutex, NULL);
    pthread_mutex_lock(&m->mutex);
    m->cond = cond_init(NULL);
    m->places_dispo = cond_init(NULL);
    pthread_mutex_unlock(&m->mutex);
}
int main(int argc, char **argv) {
    int n_loops = 20;
    int i_depot, i_extrait;
    int nb_produits = 0;
    struct monitor info_prete;
    struct monitor places_dispo;
    int ret;
    nb_produits = 0;
    i_depot = 0;
    i_extrait = 0;
    for (my_rank = nb_threads++;
        my_rank < nb_threads; my_rank++) {
        product_id = nb_produits++;
        cur_indice = i_depot++;
        printf("P%d produit %d dans %d\n", my_rank, product_id, cur_indice);
        usleep(500000);
        i_depot = i_depot % N;
        printf("C%d consomme %d depuis %d\n", my_rank, product_id, cur_indice);
        usleep(100000);
    }
    for (my_rank = nb_threads;
        my_rank < nb_threads; my_rank++) {
        product_id = nb_produits++;
        cur_indice = i_extrait;
        product_id = infos[i_extrait];
        printf("P%d produit %d dans %d\n", my_rank, product_id, cur_indice);
        usleep(500000);
        cur_indice = (cur_indice+1) % N;
        printf("C%d consomme %d depuis %d\n", my_rank, product_id, cur_indice);
        usleep(100000);
    }
    return 0;
}
```
Using synchronization

```c
for (int i = 0; i < nthreads_prod; i++) {
    ret = pthread_create(&tid_prod[i], NULL, function_prod, NULL);
    assert(ret == 0);
}
for (int i = 0; i < nthreads_cons; i++) {
    ret = pthread_create(&tid_cons[i], NULL, function_cons, NULL);
    assert(ret == 0);
}
for (int i = 0; i < nthreads_prod; i++) {
    ret = pthread_join(tid_prod[i], NULL);
    assert(ret == 0);
}
for (int i = 0; i < nthreads_cons; i++) {
    ret = pthread_join(tid_cons[i], NULL);
    assert(ret == 0);
}
return EXIT_SUCCESS;
```

### 4 Using synchronization

#### 4.1 Deadlock

- Situation such that at least two processes are each waiting for a non-shareable resource already allocated to the other.
- Necessary and sufficient conditions (Coffman, 1971 [7]):
  1. Resources accessed under mutual exclusion (non-shareable resources)
  2. Waiting processes (processes keep resources that are acquired)
  3. Non-preemption of resources
  4. Circular chain of blocked processes

- Strategies:
  - Prevention: acquisition of mutexes in the same order
  - Deadlock detection and resolution (e.g., with pthread_mutex_timedlock)

#### 4.2 Lock granularity

- Coarse grain locking
  - A lock protects a large portion of the program
  - Advantage: easy to implement
  - Disadvantage: reduces parallelism
- Fine grain locking
  - Each lock protects a small portion of the program
  - Advantage: possibility of using various resources in parallel
  - Disadvantages:
    - Complex to implement without bug (e.g., deadlocks, memory corruption)
    - Overhead (locking comes at a cost)
4.3 Scalability of a parallel system

- Scalability = ability to reduce execution time when adding processing units
- Sequential parts of a program reduce the scalability of a program (Amdahl’s law \( \frac{1}{f} \))
- In a parallel program, waiting for a lock introduced sequentiality
  \( \rightarrow \) Locks can interfere with scalability

The notion of scalability is discussed in more detail in the module CSC5001, High Performance Systems.
# Operating systems (2/2)

The operating system is responsible for operating various hardware. It therefore includes drivers capable of interacting with a particular material. The different drivers for a same type of peripheral offer the same interface, which allows the upper layers of the OS to use the hardware interchangeably.

The transition from user space to kernel space is done via a system call (syscall). The kernel processes the request for the application and returns a positive or zero integer on success, and -1 on failure.

From the application point of view, system calls are exposed as functions (defined in libc) in charge of executing the system call.
1.2 Testing the return value of system calls and functions

Let us consider the C standard function `stat()`. It returns a success status code on success, or a negative value (including `-1`) on error. If you do not test the return value, you cannot catch errors.

```c
int rc = stat(file, &buf);
if (rc < 0) {
    perror("Error while accessing file '");
    // -> message "Error while accessing file 'plop': No such file or directory"
    exit(EXIT_FAILURE);
}
```

You must always test the return value of a system call and deal with errors:

- Prevent the propagation of errors (the discovery of the error can take much later).
- Use the fail-fast approach presented in CSC4102.
- Use an error variable indicating the cause of the last error.

It is possible to define a macro displaying an error message and indicating where the error occurred. For example:

```c
#define FATAL(errnum, ...) do { 
    fprintf(stderr, __VA_ARGS__); 
    fprintf(stderr, __FILE__, __LINE__); 
} while(0)
```

The macro `assert()` tests the expression passed in parameter and, if false, displays a message describing the error, and terminates the process.

```c
int rc = stat(file, &buf);
assert(rc>=0);
```

The philosophy of error checking is to prevent the propagation of errors, and to catch them as early as possible. This is why we test the return value of system calls.

### Test a system call

When a program calls the `stat()` function, a core dump (that describes the process when the error occurred) can be generated in order to debug the program.

To activate the generation of a core dump:

```bash
ulimit -c unlimited
```

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot open file '");
    return;
}
stat(f, &buf);
```

When a program calls the `stat()` function in order to terminate the process, a core dump file (that describes the process when the error occurred) can be generated in order to debug the program.

To activate the generation of a core dump:

```bash
ulimit -c unlimited
```

### Do not propagate errors

To suppress the propagation of errors, the `assert()` function can be used.

```c
int main() {
    FILE* f = fopen(argv[1], "r");
    assert(f != NULL);
}```

However, the `assert()` macro should be used with caution because it is disabled when the program is compiled in optimized mode (with `gcc -O3` for example).

### The command line

Some functions have a list of error codes returned. For example, `fopen()`.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot open file '");
    return;
}
```

The error message associated with a value of `errno` can be obtained with the `strerror()` function:

```c
strerror(errno);"Cannot open file '");
```

### Generic error handling

It is possible to define a macro displaying an error message and indicating where the error occurred. For example:

```c
#define FATAL(errnum, ...) do { 
    fprintf(stderr, __VA_ARGS__); 
    fprintf(stderr, __FILE__, __LINE__); 
} while(0)
```

### Debugger

When a program calls the `abort()` function in order to terminate the program, it generates a core dump which can be supplied to `gdb`.

```c
while(0) {
    FATAL(errno, "Cannot access file '");
}
```

To activate the generation of a core dump, run the command `ulimit -c unlimited`. Therefore, the function `abort()` generates a core dump which can be supplied to `gdb`.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot open file '");
    return;
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```

To debug a program, it is possible to do `gdb` on the core dump.

```c
FILE* f = fopen(argv[1], "r");
if (f == NULL) {
    FATAL(errno, "Cannot access file '");
}
```
Program terminated with signal SIGABRT, Aborted.
#0 __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#0 __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
#1 0x00007ffff7dfb535 in __GI_abort () at abort.c:79
#2 0x0000555555555232 in main (argc=2, argv=0x7fffffffdcd8) at fatal.c:21

Each function call creates a stack frame:
- A stack frame contains:
  - local variables
  - a backup of the modified registers
  - the arguments of the function (specific to 32-bit x86 architectures)
  - the return address of the function (specific to x86 architectures)

Function call convention. Depending on the CPU architecture (and sometimes the compiler), the way of making a function call may vary.

On 32-bit x86 architectures, parameters are placed on the stack so that the first argument is at the address ebp + 8, the second at the address ebp + 12 (if the first argument is stored on 4 bytes), etc.

The return address (i.e. the address of the instruction to run after function) is stored on the stack at the address ebp+4.

On 64-bit x86 architectures, the parameters are passed via the rdi, rsi, rdx, rcx, r8 and r9 registers. If there are more than 6 parameters, the next parameters are placed on the stack.

The return address is also stored in a register.

On Arm architectures, parameters are passed via registers (r0 to r7 on arm 64 bit). The return address is also stored in a register.
2.2 Buffer overflow

- (in French dépassement de tampon)
- Writing data outside the space allocated for a buffer
- Risk of overwriting other data
- Security vulnerability: overwriting data may change the behavior of the application

Example. Here, the bug comes from the loop in charge of filling the array which iterates too many times (because of "< 17"). After the first 4 iterations, here is the memory status:

```
new_str = tab[0]
```

During the 5th iteration, the modification of tab[4] modifies one byte of the variable a:

```
new_str = tab[1]
```

The variable a is therefore no longer equal to 17, but 89 (or 0x45).

Security vulnerabilities. Buffer overflows are potentially serious for the security of a system, because depending on an input (e.g., a string entered by the user), the bug may modify the behavior of the application (without necessarily crashing the program). In our example, if the variable modifies the instruction pointer, the bug could allow attackers to pretend to be someone else (for example, an administrator).

Buffer overflows are among the most common security vulnerabilities. To be convinced of this, just look at the vulnerability announcements that mention "buffer overflow" (https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer%20overflow) (around 780 faults in 2017).

2.2.1 Stack overflow

- Using a buffer overflow to change the program execution flow
- The return address of a function is on the stack
  - Possibility of changing the code to be executed afterwards

Example. Here is an example of stack overflow:
System calls

Here, the foo function does not check that new_str is large enough to hold str. So if str is too long, strcpy overflows and may overwrite the return address of foo.

To avoid bugs:

- To avoid bugs
- To avoid stacks
- For this, the processor offers two operating modes
- The system mode: access to all the memory and to all the processor instructions
- The user mode: access only to the process memory and to a restricted set of instructions
- In particular, to direct access to peripherals and instructions that manage the permissions associated with the memory.

2.2.2 How to prevent buffer / stack overflow?

- Check the boundaries of buffers
  - done automatically in Java
  - not done in C / C++ because it is too expensive
- Do not use the "unsafe" functions (strcpy, gets ...)
- Use their safe counterpart instead ( strncpy, fgets ...)
- Non-executable stack (enabled by default by Linux)
- Avoid the execution of an arbitrary code
- Stack canaries
  - A canary (a specific value) is placed on the stack when entering a function
  - If when exiting the function, the canary has been modified, there has been a stack overflow
- Use the -fstack-protector-all option in gcc
- Address space layout randomization (ASLR) (enabled by default by Linux)
- Load the application code to a random address

$ gdb -stack_overflow
The program being debugged has been started already.
Start it from the beginning? (y or n)? y
Starting program: stack_overflow coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
3.2 User/system interface

- **Solution**: special processor instruction called **trap**
  - The kernel associates the address of a **syscall** function to a **trap**
  - To call a kernel function
    - The process gives the function number to call via a **parameter**
    - The process executes the **trap** instruction
  - The processor changes mode and executes the **syscall** instruction
  - **Syscall** uses the **parameter** to select the kernel function to be executed

Depending on the type of processor, the way of making a system call may vary. The way to pass the parameters is a convention which can vary from one OS to another. For example, for Linux:

**x86_32**:
- The parameters of the system call are stored in the **ebx ecx edx esi edi** and **ebp** registers;
- The syscall number is loaded into the **eax** register;
- Switching to kernel mode is done by generating the interrupt 128: **INT 0x80**;
- At the end of the system call, the return value is stored in the **eax** register.

**x86_64**:
- The parameters of the system call are stored in the **rdi rsi rdx rcx r8 r9** registers;
- The system call number is loaded in the **rax** register;
- Switching to kernel mode is done with the **syscall** instruction;
- The return value of the system call is stored in the **rax** register.

**ARM 64 bits**:
- The parameters of the system call are stored in the **x0 to x5** registers;
- The system call number is loaded in the **x8** register;
- Switching to kernel mode is done with the **svc 0** instruction;
- The return value of the system call is stored in the **x0** register.
Interruptions and communication
Gaël Thomas
CSC4508 – Operating Systems
2020–2021

Outlines

1 Communication buses ........................................................... 4
2 Interruptions ................................................................... 10

1 Communication buses

1.1 Communication buses .......................................................... 4
1.2 The memory bus ............................................................... 6
1.3 The input / output bus ........................................................ 8
2.1 The interrupt bus - principle ................................................. 10
1 Communication buses

1.1 Communication buses

- Hardware components communicate via buses
- Communication bus

- From a software point of view, there are 3 main buses:
  - Memory bus: mainly to access memory
  - Input / output bus: messages from CPUs to devices
  - Interrupt bus: messages from peripherals to CPUs

- From a hardware point of view, there is a set of hardware buses with different protocols that can multiplex the software buses.

1.2 The memory bus

- Processors use the memory bus for reads / writes
  - Sender: the processor or a peripheral
  - Receiver: most often memory, but can also be a device (memory-mapped I/O)

- The DMA controller manages the transfer between peripherals or memory
  - The processor configures the DMA controller
  - The DMA controller performs the transfer
  - When finished, the DMA controller generates an interrupt

- The processor can execute instructions during an I/O operation.

1.2.1 DMA: Direct Memory Access

- Devices use the memory bus for reads / writes
  - Sender: a processor or a peripheral
  - Receiver: memory, or can also be a device (memory-mapped I/O)

- The DMA controller manages the transfer between peripherals or memory
  - The processor configures the DMA controller
  - The DMA controller performs the transfer
  - When finished, the DMA controller generates an interrupt

- The processor can execute instructions during an I/O operation.

1.2.2 MMIO: Memory-Mapped IO

- Processors use memory bus to access devices
  - Sender: the processor or a peripheral
  - Receiver: most often memory, but can also be a device (memory-mapped I/O)

- Device memory is mapped in memory
  - When the processor accesses this memory area, the data is transferred from / to the device.
1.3 The input / output bus

- Request / response protocol, special instructions in/out
- Sender: a processor
- Receiver: a peripheral
- Examples: activate the caps-lock LED, start a DMA transfer, read the key pressed on a keyboard...

1.4 The interrupt bus - principle

- Used to signal an event to a processor
- Sender: a peripheral or a processor
- Receiver: a processor
- Examples: keyboard key pressed, end of a DMA transfer, millisecond elapsed...
- IRQ (Interrupt ReQuest): interruption number. Identifies the sending device

2.1 Receiving an interrupt

- Two tables configured by the kernel to handle reception
- Routing table: associates an IRQ with an IDT number
- IDT table (Interrupt Descriptor Table): associates an IDT number to a function called interrupt handler
- Two tables allow more flexibility than a single table which associates an IRQ number directly with a manager
- Useful in particular with multicore (see the rest of the lecture)
2.2 Receiving an interrupt: example

- A device sends an IRQ (for example 0x14)
- The routing table associates 0x14 with IRQ14
- The IDT table indicates that IRQ14 is managed by the function handle_disk_interrupt

Routing table:
- IRQ14 => IDT 47
IDT table:
- IDT 47 => handler
  handle_disk_interrupt

2.3 Receiving an interrupt (continued)

- In the processor, after executing each instruction:
  - Check if an interrupt has been received
  - If so, find the address of the associated handler
  - Switch to kernel mode and run the interrupt handler
  - Then switch back to the previous mode and continue the execution

- Note: A handler can be run anytime
  - Problem of concurrent access between handlers and the rest of the kernel code
  - Solution: masking interruptions (cli / sti)

2.4 Interruptions and multicore processors

- XAPIC protocol on pentium (x2APIC since Intel Core processors)
  - Each core has a number called APIC number (Advanced Programmable Interrupt Controller)
  - Each core handles interrupts via its LAPIC (local APIC)
  - An IOAPIC routes an interrupt to a given LAPIC
- Routing table configured by the system kernel

Routing table configured in the system kernel:
- Core 0 : LAPIC 0
- Core 1 : LAPIC 1
- Core 2 : LAPIC 2
- Core 3 : LAPIC 3

Routing table:
- IRQ 14 => APIC 1
- APIC 1 : IDT 47
- IDT table:
- IDT 47 => handler
  handle_disk_interrupt

2.5 MSI: Message Signaling Interrupt

- MSI: direct interrupt from a device to a LAPIC without passing through the IOAPIC
  - The kernel must configure the device so that it knows which LAPIC / IDT pair should be generated
  - Used when the need for performance is important
### 2.6 Inter-core communication

- One core can send an interrupt to another core
  - Called Inter-Processor Interrupt (IPI)
  - LAPIC x sends an IPI to LAPIC y
  - In LAPIC y, receiving an IPI is associated with an IDT number

#### Example

```
Core 1
LAPIC
Core 0
LAPIC
Core 3
LAPIC
Core 2
LAPIC
IPI => IDT34
IDT Table: IDT34 => handler handle_ipi
```

### 2.7 IDT table

- Table that associates a handler with each IDT number
  - Used by interrupts as seen previously
  - But also for a system call: `int 0x64` simply generates the interrupt IDT 0x64
  - But also to catch faults when executing instructions:
    - e.g., division by zero generates the interrupt IDT 0x60, an access illegal memory (SIGSEGV) the interrupt IDT 0xe etc.
- The IDT table is therefore the table that contains all of the entry points to the kernel
  - From the software via the system call
  - From material for other IDTs

### 2.8 Time management: two sources

- **Jiffies**: global time source to update the date
  - A device (e.g., HPET) regularly sends IRQ
  - Received by a single core which updates the date
- **Tick**: core-local time source used for scheduling
  - LAPIC regularly generates an interrupt to its core
  - The system associates an IDT number and a handler with this interruption
  - Less precise than the jiffies
**1 Introduction**

- A process needs to be present in main memory to run.
- Central memory divided into two parts:
  - The space reserved for the operating system
  - The space allocated to processes

**Memory management concerns the process space.**

- Memory capacities are increasing, but so are the requirements.

**Need for multiple memory levels**
- Fast memory (cache)
- Central memory (RAM)
- Auxiliary memory (disk)

**Principle of inclusion to limit updates between different levels.**

Regarding the principle of inclusion, in an Intel architecture, the L1 cache (Level 1) is included in L2 cache (Level 2), which is itself included in RAM, which is included in the "main memory" (disk).

Here are the typical access times to data located in the different types of memory on a "classic" machine (Intel Core i7 / Alder Lake processor) in 2017:

- data in the L1 cache: 4 cycles or 1 ns
- data in the L2 cache: 12 cycles or 3 ns (3 times slower than L1)
- data in the L3 cache: 44 cycles or 10 ns (10 times slower than L1)
- data in RAM: 80 - 100 ns (100 times slower than L1)
- data in SSD disk: 130 ms or 1.30 ms (100,000 times slower than L1)
- data on a hard disk: 10 ms (10 million times slower than L1)

The following table shows the cost difference between the types of memory (and the evolution over the years):

<table>
<thead>
<tr>
<th>Year</th>
<th>2008</th>
<th>2009</th>
<th>2010</th>
<th>2014</th>
<th>2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard disk drive, 7200 tr/min (in €/GiB)</td>
<td>0.50</td>
<td>0.32</td>
<td>0.10</td>
<td>0.04</td>
<td>0.027</td>
</tr>
<tr>
<td>SSD disk (in €/GiB)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.50</td>
<td>0.17</td>
</tr>
<tr>
<td>USB key (in €/GiB)</td>
<td>–</td>
<td>–</td>
<td>1.64</td>
<td>0.62</td>
<td>0.27</td>
</tr>
<tr>
<td>RAM (in €/GiB)</td>
<td>–</td>
<td>37.00</td>
<td>21.85</td>
<td>8.75</td>
<td>7.23</td>
</tr>
<tr>
<td>NVMe (in €/GiB)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.21</td>
</tr>
</tbody>
</table>

- Data available in the manual "Intel 64 and IA-32 Architectures Optimization Reference Manual".
2 Paging

2.1 Overview

The address space of each program is split into pages.

Physical memory divided into page frames.

Matching between some pages and page frames.

2.2 Status of memory pages

- The memory pages of a process can be:
  - In main memory / in RAM (active pages)
  - Non-existent in memory (inactive pages never written)
  - In secondary memory / in the Swap (inactive pages that have already been written)

Each process has a contiguous memory space to store its data.

2.3 Logical (or virtual) address

Address space divided using the most significant bits:

- $2^p$ pages and each page contains $2^d$ bytes
- Page size
  - Originally: 512 bytes or 1 KB
  - Today: 4 KB ($d = 12$ bits, so $p = 52$ bits) or more
- Last page is half wasted
- Small capacity memory: small pages
- Scalability of the page management system

In Linux, page frames are 4KB in size (defined size by the constants PAGE_SIZE and PAGE_SHIFT in the file page.h).

2.4 Overview

2.6 Implementation on a 64-bit pentium

2.6 Translation Lookaside Buffer (TLB)
Huge pages
Some applications use large amounts of data (sometimes several GiB) that must be placed in a large number of 4 KiB memory pages. In order to limit the number of pages to handle, some architectures (especially arm64 and x86_64) allow the use of larger memory pages (typically 2 MiB and 1 GiB) which are called huge pages.

2.4 Page table

- The correspondence between logical address and address physical is done with a page table that contains:
- Page frame number
- Information bits (presence, permissions, upload timestamp …)

2.5 Implementation on a 64-bit pentium

- Page table = 4 levels tree:
  - Each entry in a table gives the address of the following table
  - Page table that contains
  - Information physical page
  - Logical page number
  - Page frames number
  - Physical address

2.6 Translation Lookaside Buffer (TLB)

- Problem: any access to information requires several memory accesses
- Solution: use associative memories (fast access registers)
- Principle:
  - A number of registers are available
  - Logical page number L is compared to the content of each register
  - If found, registers give the corresponding frame number
- Observation: caching the page table

Intel architectures have Translation Lookaside Buffers (TLB) with 32, 64, or even 256 entries. TLB are sometimes called address translation cache.
Virtual memory 3 User point of view

3.1 Memory space of a process

Composed of:
- kernel space
- the heap
- the stack (one per thread)
- shared libraries

including <stdlib.h>
#include <assert.h>

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 – Operating Systems

3.1 Memory space of a process

Composed of:
- kernel space
- the heap
- the stack (one per thread)
- shared libraries

#include <stdlib.h>
#include <assert.h>

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 – Operating Systems

3.1 Memory space of a process

Composed of:
- kernel space
- the heap
- the stack (one per thread)
- shared libraries

#include <stdlib.h>
#include <assert.h>

Télécom SudParis — François Trahay — 2020–2021 — CSC4508 – Operating Systems

3.2 Memory mapping

How to populate the memory space of a process?
- For each ELF file to be loaded:
  - open the file with
    - each ELF section is mapped in memory (with mmap) with the appropriate permissions

3.3 Memory allocation

void* malloc(size_t size)

Returns a pointer to an allocated buffer of size bytes

void realloc(void* ptr, size_t size)

Changes the size of a buffer previously allocated by malloc

void calloc(size_t num, size_t size)

Same as malloc, but memory is initialized to 0

void* aligned_alloc(size_t alignment, size_t size)

Same as malloc. The returned address is a multiple of alignment

void free(void* ptr)

Free an allocated buffer

- All these functions are implemented in the standard C library (which in some cases make system calls)
- The malloc(3) algorithm is very efficient. It is not therefore generally not necessary to try to optimize it

However:
- When allocating a memory area that must be initialized at 0, we shall privilege calloc(3) (it is more efficient than a malloc(3) followed by memset(3))
- If necessary, malloc allows to fine tune the behavior of malloc(3)
  - Additionally, it is possible to customize the behavior of standard allocation / release routines by setting __malloc_hook, __realloc_hook and __free_hook. Be careful, these mechanisms can lead to run-time problems

When freeing a buffer with free, it is strongly advised to set the pointer to NULL. This allows the program to crash immediately if, by mistake, we access this (now inconsistent) buffer again using this pointer.

The following program illustrates how setting a pointer to NULL allows to crash immediately and how using a debugger allows to quickly find the origin of the error.

```c
void* resetToNULL(void*)
{
  return NULL;
}
```

resetToNULL:

0x000000000000000.text
.data
.bss
Heap
Libs
Shared libraries
Dynamic allocation
(malloc, ...)
Virtual memory

```c
void h(char *p){
    *p = 'a';
}
void g(char *p){
h(p);
}
void f(char *p){
g(p);
}
int main(){
    char ... but allocates an area filled
    with 0s
    return EXIT_SUCCESS;
}
```

4 Memory allocation strategies

4.1 Non-Uniform Memory Access

- Several interconnected memory controllers
- Memory consistency between processors
- Privileged access to the local memory bank
- Possible access (with an additional cost) to dis-
tant memory banks
- Non-Uniform Memory Access
- On which memory bank to allocate data?

4.2 First touch allocation strategy

4.4 Interleaved allocation strategy

How to request memory from the OS

- `void *sbrk(intptr_t increment)`
  - Increase the heap size by `increment` bytes
- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - Map a file in memory
  - If `flags` contains `MAP_ANON`, does not map any file, but allocates an area filled
    with 0s
## 4.2 First touch allocation strategy

- **Linux default lazy allocation strategy**
- Allocation of a memory page on the local node when first accessed
- Assumption: the first thread to use a page will probably use it in the future

```c
double *array = malloc(sizeof(double) * N);
for (int i = 0; i < N; i++) {
    array[i] = something(i);
}
```

## 4.3 Interleaved allocation strategy

- Pages are allocated on the different nodes in a round-robin fashion
- Allows load balancing between NUMA nodes

```c
double *array = numa_alloc_interleaved(sizeof(double) * N);
```

## 4.4 mbind

- Place a set of memory pages on a (set of) NUMA node
  - Allows manual placement of memory pages

```c
double *array = malloc(sizeof(double) * N);
mbind(&array[0], N/4 * sizeof(double), MPOL_BIND, &nodemask, maxnode, MPOL_MF_MOVE);
```

It is also possible to use `set_mempolicy` in order to choose an allocation strategy for future memory allocations.
Why this lecture?

- To understand what is happening in the “hardware” part of the execution stack
- To write programs that are efficient on modern machines

In fact, the compiler generally manages to generate a binary which exploits all the capacities of the processor. But the compiler sometimes fails and generates non-code-optimized. We must therefore be able to detect the problem, and be able to write code that the compiler can optimize.
1.1 Moore’s Law

1965 - 2005
- Moore’s Law (1965): the number of transistors in microprocessors doubles every two years
- The fineness of the processor engraving decreases
- The clock frequency increases
  ⇒ Increased processor performance

Since 2005
- The fineness of engraving continues to decrease (but less quickly)
- The clock frequency no longer increases
- Heat dissipation depends on frequency, number of transistors, size of transistors
  ♦ If we reduce the size of the transistors, we have to reduce the frequency

2 Sequential processor

- An instruction requires N steps
  - Fetch: load instruction from memory
  - Decode: identify the instruction
  - Execute: execution of the instruction
  - Writeback: storage of the result
- Each step is processed by a processor circuit
- Most circuits are not used at every stage
  → One instruction is executed every N cycles

3 Pipeline

- Instruction pipeline
  - At each stage, several circuits are used
  → One instruction is executed at each cycle

3.1 Micro architecture of a pipeline

- Each stage of the pipeline is implemented by a set of logic gates
- Execute step: one subcircuit per type of operation (functional unit)
3.2 Superscalar processors

- Use of different functional units simultaneously
- Several instructions executed simultaneously
- Requires to load and decode several instructions simultaneously

Figure 3: Micro-architecture of a superscalar processor

3.2.1 Superscalar processors

- How to fill the pipeline when the instructions contain conditional jumps?

```assembly
cmp a, 7; a > 7?
ble L1
mov c, b; b = c
br L2
L1: mov d, b; b = d
L2: ...
```

- In case of a bad choice: the pipeline must be "emptied"
  \[ \Rightarrow \text{waste of time} \]

The cost of a wrong choice when loading a branch depends on pipeline depth: the longer the pipeline, the longer it takes to empty it (and therefore wait before executing an instruction). For this reason (among others), the depth of the pipeline in a processor is limited.
### 3.4 Branch prediction

- The processor implements a prediction algorithm
- General idea:
  - For each conditional jump, store the previous results
  
  | addr branch history | 0x23 0011 | 0x42 1000 | 0x5A 1111 | 0x7E 0000 |

- The branch prediction algorithms implemented in modern processors are very advanced and reach an efficiency greater than 98% (on the SPEC89 benchmark suite).

- To know the number of good / bad predictions, we can analyze the hardware counters of the processor.

  - With the PAPI library, the `PAPI_BR_PRC` and `PAPI_BR_MSP` counters give the number of conditional jumps correctly and incorrectly predicted.

  - Linux `perf` also allows collecting this information (among others). For example:

    ```bash
    $ perf stat -e branches,branch-misses ./branch_prediction 0
    is random is not set
    100000000 iterations in 1178.597000 ms
    result=199999996
    Performance counter stats for './branch_prediction 0':
    2447232697 branches
    6826229 branch-misses # 0,28% of all branches
    1.179914189 seconds time elapsed
    1.179784000 seconds user
    0.000000000 seconds sys
    ``

  - Vector instructions were democratized at the end of the years 1990 with the MMX (Intel) and 3DNow! (AMD) instruction sets that allow to work on 64 bits (for example to process 2 32-bit operations at once).

  - Since then, each generation of x86 processors brings new extensions to the instruction set: SSE2, SSE3, SSE4, AVX, AVX2, AVX512 (256 bit), AVX32 (512 bit). The other types of processors also provide vector instructions sets (eg NEON [128 bit], or Scalable Vector Extension [SVE] on ARM).

  - Vector instructions sets are specific to certain processors. The `/proc/cpuinfo` file contains (among others) the instructions sets that are available on the processor of a machine. For example, on an Intel Core i7:

    ```bash
    $ cat /proc/cpuinfo
    processor : 0
    vendor_id : GenuineIntel
cpu family : 6
    model : 69
    model name : Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
    stepping : 1
    microcode : 0x1d
cpu MHz : 1484.683
cache size : 4096 KB
    physical id : 0
siblings : 4
    core id : 0
    cpu cores : 2
    apicid : 0
    initial apicid : 0
    fpu : yes
    fpu_exception : yes
cpuid level : 13
    apic
    flags : fpu ves dvd pse tsc ksrs vtcesd ds longmode seq scrn bit decis begin
    ``

### 3.5 Vector instructions

- Many applications run in Data Parallelism mode
- Single Instruction Multiple Data (SIMD): the same operation applied to a set of data

  ```
  for(i=0; i<size; i++) {
    C[i] = A[i] * B[i];
  }
  ```

- Example: image processing, scientific computing

- Using vector instructions (MMX, SSE, AVX, ...)
  - Instructions specific to a processor type
  - Process the same operation on multiple data at once

- Instructions specific to a processor type

  ```
  for(i=0; i<size; i+= 8) {
    *pC = _mm_mul_ps(*pA, *pB);
    pA++; pB++; pC++;
  }
  ```

- With the PAPI library, the `PAPI_BR_PRC` and `PAPI_BR_MSP` counters give the number of conditional jumps correctly and incorrectly predicted.
The `flags` field contains the list of all the capabilities of the processor, especially the available instruction sets: MMX, SSE, SSE2, SSSE3, SSE4_1, SSE4_2, AVX2.

Vector instructions can be used directly in assembler or by exploiting the intrinsics provided by compilers. However, because of the number of available instruction sets and since each new processor generation provides new instruction sets, it is recommended to leave the compiler optimize the code, for example using the `-O3` option.

**4.1 Hyperthreading / SMT**

- Problem with superscalar / vector processors:
  - The application must have enough parallelism to exploit
  - Other applications may be waiting for the CPU

- **Simultaneous Multi-Threading (SMT, or Hyperthreading):**
  - Modify a superscalar processor to run multiple threads
  - Duplicate some circuits
  - Share certain circuits (e.g. FPU) between processing units

SMT is an inexpensive way to increase the performance of a processor: by duplicating the "small" circuits (ALU, registers, etc.) and by pooling the "big" circuits (FPU, prediction of branches, caches), we can execute several threads simultaneously. The additional cost in terms of manufacturing is light and the gain in performance can be significant.

Since the dispatcher schedules the instructions of several threads, a branch mis-prediction becomes less serious since while the pipeline of the thread is emptied, another thread can be scheduled.

The performance gain when multiple threads are running is not systematic since some circuits remain shared (for example, the FPU).

**4.2 Multi-core processors**

- Limited scalability of SMT
  - Dispatcher is shared
  - FPU is shared
  - Duplicate all the circuits

It is of course possible to combine multi-core with SMT. Most semiconductor foundries produce multi-core SMT processors: Intel Core i7 (4 cores x 2 threads), SPARC T5 Niagara-3 (16 cores x 8 threads), IBM POWER 7 (8 cores x 4 threads).
4.3 SMP architectures

- Symmetric Multi-Processing
- Multiple processors sockets on a motherboard
- The processors share the system bus
- Processors share memory
- Scalability problem: contention when accessing the bus

4.4 NUMA architectures

- NUMA nodes connected by a fast network
- Memory consistency between processors
- Privileged access to the local memory bank
- Access possible (with an additional cost) to memory banks located on other nodes → Non-Uniform Memory Architecture

5 Memory hierarchy

5.1 Motivation

- Until 2005: increase in CPU performance: 55 % / year
- Since 2005: increase in the number of cores per processor
- Increased memory performance: 10 % / year
- The memory accesses which are now expensive: Memory Wall
- Mechanisms are needed to improve memory performance

Until the 1990s, the bottleneck was the processor. From the software point of view, developers had to minimize the number of instructions to be executed.

As the performance of processors increases, the bottleneck is now the memory. On the software side, we therefore seek to minimize the number of costly memory accesses. This pressure on memory is exacerbated by the development of multi-core processors.

For example, an Intel Core i7 processor can generate up to 2 memory access per clock cycle. An 8-core processor with hyper-threading (16 threads) running at 3.0 GHz \(^1\) can therefore generate \(2 \times 16 \times 1 \times 3 \times 10^9\) = 96 billion memory references per second. If we consider access to 64-bit data, this represents 3.072 GB/s (3.072 TB/s). In addition to these data accesses, the memory access to the instructions (up to 128 bits per cycle)...

\(^1\) Example: an Intel Core i7-5960X released in 2014.
Architecture 5 Memory hierarchy

instruction) also have to be taken into account. We thus arrive to a 6144 GiB/s (therefore 6.144 TiB/s !) maximum flow.

For comparison, in 2016 a DDR4 RAM DIMM has a maximum throughput of around 20 GiB/s. It is therefore necessary to set up mechanisms to prevent the processor from spending all its time waiting for memory.

5.2 Cache memory

- Memory access (RAM) are very expensive (approx. 60 ns - approx. 180 cycles)
- To speed up memory access, let’s use a fast cache memory:
  - L1 cache: very small capacity (typically: 64 KiB), very fast (approx. 4 cycles)
  - L2 cache: small capacity (typical: 256 KiB), fast (approx. 10 cycles)
  - L3 cache: large capacity (typically: between 4 MiB and 30 MiB), slow (approx. 40 cycles)
- Very expensive hard disk access (SWAP): approx. 40 ms (150 µs on an SSD disk)

To visualize the memory hierarchy of a machine, you can use the lstopo tool provided by the hwloc project.

# 22

5.3 Memory Management Unit (MMU)

- Translates virtual memory addresses into physical addresses
- Look in the TLB (Translation Lookaside Buffer), then in the page table
- Once the physical address is found, request the data from the cache / memory

5.3.1 Fully-associative caches

- Cache = array with N entries
- For each reference, search for Tag in the array
- If found (cache hit) and Valid = 1: access to the cache line Data
- Otherwise (cache miss): RAM access
- Problem: need to browse the whole table → Mainly used for small caches (ex: TLB)

The size of a cache line depends on the processor (usually between 32 and 128 bytes). You can find this information in /proc/cpuinfo:

```
$ cat /proc/cpuinfo | grep cache_alignment
cache_alignment : 64
```

To realize the memory hierarchy of a machine, you can use the latstop tool provided by the hwloc project.

```
Architecture 5 Memory hierarchy
# 21

5.2 Cache memory

- Memory access (RAM) are very expensive (approx. 60 ns - approx. 180 cycles)
- To speed up memory access, let’s use a fast cache memory:
  - L1 cache: very small capacity (typically: 64 KiB), very fast (approx. 4 cycles)
  - L2 cache: small capacity (typical: 256 KiB), fast (approx. 10 cycles)
  - L3 cache: large capacity (typically: between 4 MiB and 30 MiB), slow (approx. 40 cycles)
- Very expensive hard disk access (SWAP): approx. 40 ms (150 µs on an SSD disk)

To visualize the memory hierarchy of a machine, you can use the lstopo tool provided by the hwloc project.

# 22

5.3 Memory Management Unit (MMU)

- Translates virtual memory addresses into physical addresses
- Look in the TLB (Translation Lookaside Buffer), then in the page table
- Once the physical address is found, request the data from the cache / memory

5.3.1 Fully-associative caches

- Cache = array with N entries
- For each reference, search for Tag in the array
- If found (cache hit) and Valid = 1: access to the cache line Data
- Otherwise (cache miss): RAM access
- Problem: need to browse the whole table → Mainly used for small caches (ex: TLB)

The size of a cache line depends on the processor (usually between 32 and 128 bytes). You can find this information in /proc/cpuinfo:

```
$ cat /proc/cpuinfo | grep cache_alignment
cache_alignment : 64
```

To realize the memory hierarchy of a machine, you can use the latstop tool provided by the hwloc project.
5.3 Memory Management Unit (MMU)

5.3.2 Direct-mapped caches
- Using the least significant bits of the address to find the index of the entry in the cache.
- Comparison of the Tag (most significant bits) of the address and the entry.
- Direct access to the cache line.
- Warning: risk of collision.
- Example: 0x01234567 and 0x01272671

5.3.3 Set-associative caches
- Index to access a set of K cache lines.
- Search for the Tag among the addresses of the set.
- K-way associative cache (in French: Cache associatif K-voies)

5.3.4 Cache consistency
- What if 2 threads access the same cache line?
  - Concurrent read: replication in local caches.
  - Concurrent write: need to invalidate data in other caches.
  - Cache snooping: the cache sends a message that invalidates the others caches.

Nowadays, caches (L1, L2 and L3) are generally associative to 4 (ARM Cortex A9 for example), 8 (Intel Sandy Bridge), or even 16 (AMD Opteron Magny-Cours) ways.

To detail this course a little more, we recommend this page web: Modern microprocessors – A 90 minutes guide! ([http://www.lighterra.com/papers/modernmicroprocessors/](http://www.lighterra.com/papers/modernmicroprocessors/)).

For (more) more details, read the books [?] and [?] which describe in detail the architecture of computers.

If you are looking for specific details, read [?].

Bibliography
In this lecture, we mainly talk about files, as this is the easiest example of I/O to manipulate. However, note that the content of the first 3 sections apply to I/O other than files (eg sockets).

Reminder on files:

- A file is a series of contiguous bytes stored in a medium (for example, a disk) under a name (the "name of the file").
- We distinguish several types of files:
  - text: containing bytes that can be displayed on the screen. This type of file consists of lines identified by the character end of line (on Unix, ASCII code character 10 while on Windows, ASCII code character 10 followed by a character of ASCII code 13);
  - binary: containing bytes that cannot be displayed on the screen.

On Unix, the commands `hexdump -C filename`, `bless filename` or `xxd filename` shows the precise contents of a file. Use them to 1) compare the contents of `helloWorldUnix.c` and `helloWorldWindows.c`, 2) see that the file `default_names_fichierIssuDuTP10DuModuleCSC4103.txt` is not quite a text file (and, see also here how are the accented characters stored in a file).

- When you `open` a file, the operating system provides a notion of current position (sometimes called `offset` in the rest of this course) for reading or writing:
  - This current position determines which byte in the file will be read/written during the next I/O operation.
  - This offset advances each time a read or write operation is performed.
  - The operating system provides the user with primitives to explicitly change this position (without reading or writing bytes).
- The `end of a file` corresponds to the location behind the last byte of the file. When a program reaches the end of file, it cannot read bytes anymore. On the other hand, the program can write bytes (depending on the mode in which the file was opened).
- There are 3 ways to access a file:
  - Sequential: the bytes are read one after the others from the beginning of the file.
  - Direct: you can set the offset without reading bytes before effect.
  - Indexed sequential: the file contains records, each record being identified by a key (unique or not). Using the key, you can position the offset at the start of a recording. You can also read the recordings in the order defined by their key.
The Linux system and the C library provide sequential and direct access modes. For an indexed sequential access mode, other libraries are required (Unix NDBM, GDBM, Oracle Berkeley DB ...).

### 1 Buffered / non-buffered I/O

- **Buffered I/O**
  - Write operations are grouped in a buffer which is written to disk from time to time
  - When reading, a data block is loaded from disk to buffer
  - An operation on the disk
- **Unbuffered I/O**
  - An operation on the disk
  - Open file identified by a file descriptor of type `int`

† To be exact, an “unbuffered” I/O generates a system call. The OS can then decide to cache the data or not.

### 2 I/O primitives

#### 2.1 File open / close

- **int open(const char *path, int flags, mode_t mode):**
  - return = file descriptor
  - Flags can take one of the following values:
    - `O_RDONLY`: read only
    - `O_WRONLY`: write only
    - `O_RDWR`: read and write
  - Additional flags:
    - `O_APPEND`: append data (write at the end of the file)
    - `O_TRUNC`: truncate (empty the file when opening it)
    - `O_CREAT`: creation if the file does not exist. The permissions are `mode & ~umask`
    - `O_SYNC`: open file in synchronous write mode
    - `O_NONBLOCK` (or `O_NDELAY`): open and subsequent operations performed on the descriptor will be non-blocking.

- **int close(int desc):**

#### 2.2 Reading on a file descriptor

- **int read(int fd, void *buf, size_t count):**
  - return = number of bytes read

#### 2.3 Writing on a file descriptor

- **int write(int fd, const void *buf, size_t count):**
  - return = number of bytes written

#### 3. File descriptor duplication

- **int dup2(int old_fd, int new_fd):**
  - return = file descriptor

About the `O_SYNC` option in `open`:

- To improve performance, by default, during a write operation, the operating system does not physically write the bytes on disk (they are stored in a kernel cache, waiting to be written to disk)
- Therefore, in the event of a sudden stop of the machine (example: power outage):
  - Data thought to have been written to disk may be lost because they were in fact in memory;
  - There is also a risk of inconsistency in the data on the disk.
- Solutions to synchronize file data in memory with the disk:
  - Implicit synchronization (i.e. on each write): adding the `O_SYNC` option when opening the file;
  - Explicit synchronization (i.e. the application decides) via the `fsync` primitive

Note that we can also create a file using the `creat` primitive:

- **int creat(const char *path, mode_t mode):**
  - return value = file descriptor

which is equivalent to the following call to `open`:

- **open(path, O_WRONLY|O_CREAT|O_TRUNC, mode):**
2.2 Reading on a file descriptor

- ssize_t read(int fd, void *buf, size_t count): return == number of bytes successfully read
- When read returns, the buf zone contains the read data;
- In the case of a file, the number of bytes read may not be equal to count:
  - We reached the end of the file
  - We did a non-blocking read and the data was exclusively locked

In the case where the read function is used on a descriptor other than a file (e.g. a pipe, or a socket), the fact that the number of bytes read may not equal count may have other meanings:
- for a communication pipe (see the Inter-process communication lecture), the correspondent has closed its end of the pipe.
- for a socket (see course NET4103), the network protocol uses new data packets smaller than the size that is requested.

• either the data written by P1,
• but never a mix of the data written by P1 and P2.
Note that when the file is opened with the option O_APPEND, if P1 and P2 write simultaneously (at the end of the file, because of O_APPEND), when the two processes will have finished their writing, we will find at the end of file:
- either the data written by P1 followed by the one written by P2,
- or the data written by P2 followed by the one written by P1.

No writing is therefore lost! Attention, this concurrent write at the end of file is not equivalent to two processes simultaneously performing the following operations:

\[ \text{lsync(fd, O_APPEND)} / \text{move the cursor to the end of file} / \text{write(fd, data, taille)} \]

In fact, in the latter case, one of the written data may be overwritten by the other.

The copy.c file on the next page illustrates the use of open, read, write and close.

2.3 Writing on a file descriptor

- ssize_t write(int fd, const void *buf, size_t count): return value = number of bytes written
- In the case of a file, the return value (without error) of the write operation means that:
  - Bytes were written to kernel caches unless O_SYNC was specified at file open;
  - Bytes have been written to disk if O_SYNC was specified.
- In the case of a file, a number of bytes written that is different from count means an error (e.g. file space left on device)

Writing to disk is atomic: if two processes P1 and P2 simultaneously write to the same file in the same location, when the two processes have finished their writing, we will find:
- either the data written by P1,
This operation of copying the contents of one file to another descriptor is an operation frequently performed in web servers. Indeed, these servers must in particular send the content of files to clients who have requested them. This is why the Linux system offers the `sendfile` primitive (`ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)`). It reads count bytes of in_fd and writes them to out_fd (which must match an open file descriptor) more efficiently than the combination `read` / `write`.

The `fallocate` function is the Linux specific version of the portable function `posix_fallocate`.

---

**2.4 File descriptor duplication**

- **Mechanism** mainly used to perform redirection of the three standard I/O files.
- `int dup(int old_fd)`: return value = new_fd
  - associates the smallest available file descriptor of the calling process the same entry in the open files table as the descriptor old_fd.
- `int dup2(int old_fd, int new_fd)`: force the file descriptor new_fd to become a synonym of the old_fd descriptor. If the descriptor new_fd is not available, the system first closes `close(new_fd)`.

---

**3 I/O and concurrence**

**3.1 Locking a file**

- **struct flock**
  - `short l_type; short l_whence; off_t l_start; off_t l_len;`
- `int fcntl(int fd, F_SETLK, struct flock*lock);`
  - Locks are attached to an inode. So locking a file affects all file descriptors (and therefore all open files) corresponding to this inode.
  - A lock is the property of a process: this process is the only one authorized to modify or remove it.
  - Locks have a scope of `[integer1:integer2]` or `[integer:]`
  - Locks have a type:
    - `F_RDLCK`: allows concurrent read access
    - `F_WRLCK`: exclusive access
  - The `exclusive_lock.c` file illustrates exclusive file locking:

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(){
    int fd;
    struct flock lock;
    fd = open("/tmp/ficTest",O_RDWR|O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO);
    if(fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    /* Exclusive lock on the 15 th byte */
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 15;
    lock.l_len = 1;
    /* Because of the F_SETLKW parameter, we get stuck on the fcntl if */
    /* the lock cannot be acquired */
    printf("attempt to acquire an exclusive lock by process %d...
", getpid());
    if(fcntl(fd, F_SETLKW, &lock) < 0){
        perror("Acquiring lock");
        exit(EXIT_FAILURE);
    }
    printf("... Exclusive lock acquired by process %d
", getpid());
    /* Here we could do the processing that needed to be protected */
    /* by the lock */
    sleep(10);
    /* Release the lock */
    printf("Releasing the lock by process %d...
", getpid());
    lock.l_type = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 15;
    lock.l_len = 1;
    if(fcntl(fd, F_SETLK, &lock) < 0){
        perror("Releasing lock");
    }
    return 0;
}
```

---
The `shared_lock.c` file illustrates the shared locking:

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
```

```c
    int fd;
    struct flock lock;
    fd = open("/tmp/ficTest", O_RDWR|O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    /* Shared lock on the 15th byte */
    lock.l_type = F_RDLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 15;
    lock.l_len = 1;
    /* Because of the F_SETLKW parameter, we get stuck on the fcntl if */
    /* the lock cannot be acquired */
    printf("attempt to acquire a shared lock by process %d...
", getpid);
    if (fcntl(fd, F_SETLKW, &lock) < 0)
        perror("Acquiring lock");
        exit(EXIT_FAILURE);
    printf("... shared lock acquired by process %d
", getpid);
    /* Here we could do the processing that needed to be protected */
    /* by the lock */
    sleep(10);
    /* Release the lock */
    printf("Releasing the lock by process %d...
", getpid);
    lock.l_type = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 15;
    lock.l_len = 1;
    if (fcntl(fd, F_SETLK, &lock) < 0)
        perror("Releasing lock");
        exit(EXIT_FAILURE);
    printf("...OK
");
    return EXIT_SUCCESS;
}
```

- If we run `exclusive_lock` first, running `exclusive_lock` or `shared_lock` wait before locking.
- If we run `shared_lock` first, another `shared_lock` can set the (shared) lock. On the other hand, a `exclusive_lock` must wait to be able to lock.
- Note that `exclusive_lock` may suffer starvation:
  - start a 1st `shared_lock`
  - start `exclusive_lock` it wait
  - start a 2nd `shared_lock`. The 1st `shared_lock` ends. But as the 2nd `shared_lock` is running, `exclusive_lock` is still waiting.
  - start a 3rd `shared_lock`. The 2nd `shared_lock` ends. But as the 3rd `shared_lock` is running, `exclusive_lock` is still waiting.
  - see that as long as `shared_lock` starts while the previous `shared_lock` has not finished running, `exclusive_lock` must wait. `exclusive_lock` may face starvation.

To prevent this starvation, we must add a mutual exclusion.
4.1 Giving advices to the kernel

- int posix_fadvise(int fd, off_t offset, off_t len, int advice)
- Examples of advice: POSIX_FADV_SEQUENTIAL, POSIX_FADV_RANDOM, POSIX_FADV_WILLNEED
- Return value: 0 if OK, error number otherwise
- Allows you to tell the kernel how the program will access a file, which allows the kernel to optimize accordingly

Since January 2011, we know that this function is used in Firefox to reduce startup time by 40% to 50% by loading more efficiently GUI libraries xul.dll and mozjs.dll (more information here: https://bugzilla.mozilla.org/show_bug.cgi?id=627591).

4.2 Asynchronous I/O

- int aio_read(struct aiocb *aiocbp)
- int aio_write(struct aiocb *aiocbp)
- Starts an asynchronous read/write operation
- Returns immediately
- int aio_cancel(const struct aiocb *const aiocb_list[], int nitems, const struct timespec *timeout)
- Waits for the end of an asynchronous operation
- int aio_error(const struct aiocb *aiocbp)
- Tests the end of an asynchronous operation

For more information on asynchronous I/O, refer to the documentation (man 7 aio).

The current implementation of AIO Posix is provided in user-land by libc and can cause scalability issues.

Another solution is to use the Asynchronous I/O interface provided by the Linux kernel (see the system calls io_submit, io_setup, etc.), or the libaio library which provides an overlay to Linux system calls.

4.3 mmap

- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
- "map" a file in memory
- Memory access to the buffer are transformed into disk operations
- int munmap(void *addr, size_t length)
- "unmap" a buffer

To ensure that the memory accesses have been passed on to the disk, you can use the msync function.
FILE SYSTEMS

1 Device and device driver

1.1 Device and device driver
1.3 Devices in UNIX
1.3.2 Types of peripherals
1.5 Block devices in soft
1.5.1 Principles of the caches algorithm
1.1 Device and device driver

- **Device** = hardware component other than CPU and memory
- **Device driver** = software allowing access to a device
  - A data structure giving the status of the device
  - An input / output function allowing access to the device
  - The driver is usually found in the kernel

1.2 Devices in UNIX

- A device is identified by a number called dev
  - Most significant bits (major): driver number
    - For example: 8 = SSD hard drive driver
  - Least significant bits (minor): device number
    - For example: 0 = disk 1, 1 = disk 1 / part 1, 2 = disk 1 / part 2
  - The kernel contains a table which associates a driver number with the driver (access function + status)

1.3 2 types of peripherals

- **“character”** devices
  - Read / write byte by byte
  - Generally access via MMIO or input / output bus
  - Blocks the CPU during the I/O operation
  - Examples: Keyboard, printer, sound card...
- **“block”** devices
  - Read / write by data blocks (typically 512 bytes)
  - The device is therefore seen as an array of blocks
  - Usually access via DMA
  - Does not block the CPU during the I/O operation
  - Examples: Hard disk, DVD player...

1.4 Block devices in xv6

- A single block device driver in xv6
  - Manages IDE hard disks
  - Function `iderw()` in `/dev/ide`
  - `iderw()` takes a buf (buf.h) structure as a parameter
    - buf.flags:
      - B_VALID: if false, read operation requested
      - B_DIRTY: if true, write operation requested
    - buf.dev/blockno: access to block `blockno` from disk `dev`
    - buf.data: data read or written
      - If read, the output of `iderw`, data = data read
      - If write, the input of `iderw`, data = data to write
1.5 Principle of the iderv algorithm

- iderv mainly performs the following actions:
  - Start the DMA transfer (see lecture #5)
  - From memory to disk if write request
  - From disk to memory if read request
  - Sleep the process with the sleep function (see lecture #4)
    → switch to another ready process

- Once the transfer is complete:
  - The disk generates an interrupt
  - The interrupt is handled by the ideintr function
  - ideintr calls wakeup to wake up the sleeping process

2.1 The I/O cache

- Disk access is very slow compared to memory access:
  - Hard disk drive: several milliseconds
  - SSD disk: x10, hundreds of microseconds
  - NVMe disk: x100, microseconds
  - Memory: x100, dozens of nanoseconds

- I/O cache improves the performance of block type devices:
  - Keeps frequently or recently used blocks in memory
  - Managed by the operating system kernel

2.2 Principle of an I/O cache

- The system manages a set of buffers in memory:
  - If the block is not yet in the cache:
    1. Remove an unused buffer from the cache
    2. Copy the contents of the disk block to this buffer
  - Otherwise, simply return the buffer associated with the block

- To modify a block (write operation):
  1. Read the block (call the read operation)
  2. Modify the contents of the buffer in memory
  3. Mark buffer as modified (written to disk later)
2.3 The xv6 buffer cache

- Buffer cache = xv6 I/O cache
- Made up of a finite set of buf structures
- Each buf structure is associated with a block of a disk

- Three possible states:
  - B_VALID: read operation incomplete (requires read)
  - B_VALID and B_DIRTY: data in memory and buffer is unmodified
  - B_VALID and B_DIRTY: data in memory and buffer is modified

  → need to be written to disk before leaving the cache

→ in `iderw()`, if B_VALID ⇔ read and B_DIRTY ⇔ write

2.4 How the buffer cache works (1/3)

- The buf structures form a circular double linked list, the head is the most recently used block

- `struct buf* bget(uint dev, uint blkno)`: return a locked buffer associated to (dev, blkno)
  - If there is already a buffer associated with (dev, blkno)
  - Increments a reference counter associated with the buffer
  - Locks the buffer
  - Returns the buffer
  - Otherwise
  - Search for a buffer with counter == 0 and with the state ! B_DIRTY
  - Associate the buffer with (dev, blkno) (+ cpt = 1 and lock the buffer)

2.5 How the buffer cache works (2/3)

- `struct buf* bread(uint dev, uint blkno)`: return a locked buffer in the B_VALID state
  - Call `bget()`
  - If the buffer is ! B_VALID, call `iderw()`

- `void bwrite(struct buf* b)`: Writes the contents of b to disk
  - Mark the buffer B_DIRTY
  - Call `iderw()` to write the buffer

2.6 How the buffer cache works (3/3)

- `void brelse(struct buf* b)`: Release the lock associated with b
  - Decreases the reference counter
  - Move the buffer to the head of the list (most recently used)
3.2 Operation versus writing to disk

- File creation requires:
  - Allocation of a new file
  - Adding the name to a directory
- Adding data to a file requires:
  - Writing new blocks to disk
  - Updating the file size
- Deleting a file requires:
  - Deleting the data blocks from the file
  - Deleting the name from the directory

3.2 Consistency issues

- The system can crash anytime
  - Inconsistency if it stops in the middle of an operation
    - A name in a directory references a non-existent file
    - Data added to a file but size not updated
- Operations must be propagated in the order in which they were performed
  - Inconsistency if propagation in random order
    - Adding a file then deleting ⇒ the file does not exist at the end
    - Deleting a file then adding ⇒ the file exists at the end
    - Similarly, adding data then truncating (size should be 0)

3.3 Bad solutions

- No cache when writing (directly propagate write operations)
  - Very inefficient because each write becomes very (very!) slow
- Recovery in the case of a crash
  - Recovering a file system is slow
    - Examples: FAT32 on Windows or ext2 on Linux
  - Recovering is not always possible
    - A crash makes the filesystem unusable!
3.4 First idea: transactions

- A transaction is a set of write operation that is
  - Either fully executed
  - Or not executed at all
- Principle of implementation
  - An operation (coherent set of writes) «is a transaction
  - The writes of a transaction are first written to disk in a “pending” area
  - Once the operation is complete, the “pending” area is marked as valid (the transaction is complete)
  - Regularly (or in the event of a crash), validated writes in the pending zone are propagated to the file system.

3.5 Second idea: log

- To ensure that the entries are propagated in order in which they were executed, the pending zone is structured like a log
  - Each entry is added at the end of the log
  - The validated transactions of the pending zone are propagated to the file system in the order of the log (from the start of the log to the end).

3.6 Third idea: parallel log

- Problems: Multiple processes may perform transactions in parallel
  - Parallel transaction writes are interleaved in the log
    - How do you know which ones are validated?
- Classic solution
  - If several transactions in //, all the operations are validated when the last one is completed
  - Advantage: easy to implement (count of the number of operations in //)
  - Disadvantage: risk of never validating if new operations continue to arrive.

3.7 log structure

- The system technically manages two logs
  - One in memory called memory log
    - Contains only the list of modified block numbers
    - The content of the modified blocks is in the buffer cache
  - One on disk called disk log
    - Contains the list of modified block numbers and a copy of the blocks
    - Note: the block is propagated from the log to the filesystem later
      - the system can therefore manage up to 3 copies of a block
  - One on disk in the file system called disk block
  - One in memory in the buffer cache called cached block
3.8 Log algorithm principle

- Steps to modify block number n
  1. Load the disk block in the buffer cache
  2. Modification of the buffer (i.e., cached block)
  3. Add n to the list of modified blocks in the memory log
- At the end of an operation, steps to validate the transaction
  1. Copy modified cached blocks to disk log
  2. Copy the modified block list to disk log
  3. Mark the transaction as validated
- Later, to propagate the transaction
  1. Copy disk log blocks to file system
  2. Reset disk log and memory log

3.9 Using the log

- Three functions in the log management interface (log.c)
  - `begin_op`(): start a transaction
  - `end_op`(): complete a transaction
  - `log_write(struct buf* b)` : add b to the transaction
- To perform a logged operation, instead of calling directly `write()`, so we have to execute:
  1. `begin_op()`
  2. `log_write(buf)`
  3. `log_write(buf)`
  4. `end_op()`

3.10 Implementation in xv6 (1/3)

- `void begin_op()` : start a transaction
  - If log writing to disk in progress, wait
  - If the log is full, wait
  - Increments the number of pending operations (log.outstanding)
- `void end_op()` : complete a transaction
  - Decrements the number of operations in progress, and if equal to 0:
    - Write memory log + cached blocks in disk log (`write_log()`)
    - Mark committed disk log transaction (`write_head()`)
    - Propagate writes from disk log to the filesystem (`install_trans()`)
    - Delete logs in memory and on disk (`write_head()`)

3.11 Implementation in xv6 (2/3)

- `void log_write(struct buf* b)` : add the block associated with b to the transaction
  - Add block number to memory log
  - Mark buffer as B_DIRTY =⇒ does not leave the cache (see `bget()`)
3.12 Implementation in xv6 (3/3)

■ After a crash, call `install_trans()` which propagates the actions from disk log to file system.

■ In the worst case, actions that had already been performed are replayed.

■ But at the end of the replay, the file system is in a consistent state.

4.1 File system

■ File system: defines the structure for storing files (often for a block type device)

■ UFS: Unix File System (xv6, BSD)

■ ext: extended file system (Linux - ext4 nowadays)

■ NTFS: New Technology File System (Windows)

■ APFS: Apple File System (MacOS)

■ FAT: File Allocation Table (Windows)

■ BTRFS: B-TRee File System (Linux)

■ and many others!

4.2 Principle of a file system

■ File: consistent set of data that can be read or written.

■ Filesystem: associate names and files.

■ Example: `/etc/passwd` → root:*:0:0:System Administrator...

■ Usually a special symbol is used as a separator for directories.

■ `/` in UNIX systems, `\` in Windows systems.
4.3 Partitions

- A disk is often made up of several partitions
- Partition = continuous area that contains a file system
- Typical structure of a disk
  - First block: partition table
    - For example: Master Boot Record
    - In charge of loading the kernel of one of the partitions
    - For example: LILO, GRUB
  - Blocks x to y: partition 1
  - Blocks y to z: partition 2
  - etc...

4.4 Disk image

- A file itself can contain the data of a complete disc
- Called a disk image or a virtual disk
- Typically used in virtualization
- For example: xv6.img is the disk image used with the qemu emulator to start xv6
5.2 Dinode

- A file on disk consists of:
  - metadata called a dinode (fixed size, see fs.h)
  - file type (ordinary, directory, device)
  - file size
  - the list of the file data blocks
  - an indirection block (see following slides)
  - device number if device file
  - number of hard links to the file (reminder: a hard link is a name in a directory)
  - data blocks.
  - these are the blocks that contain the content of the file

5.3 Data blocks of a file

- A dinode directly lists the numbers of the first 12 blocks
  - the dinode.addrs[0] block contains bytes 0 to 511 of the file
  - the dinode.addrs[1] block contains bytes 512 to 1023
  - the dinode.addrs[i] block contains bytes i*512 to i*512 + 511
- The indirection block contains the following block numbers
  - the indirection block number ind is given in dinode.addrs[12]
  - the tail [0] block contains bytes 12*512 to 12*512 + 511

Note: since a block is 512 bytes and a block number is coded out of 4 characters, a file has a maximum size of 12 + 512/4 blocks.

5.4 Adding a block to a file

- To add a new block to a dinode dino (function bmap() in fs.h)
  1. Find a free block number in the table of free blocks
  2. Mark the occupied block (put its bit 1 in the table)
  3. Add the block number to the list of data blocks in dino.
- this addition may require to allocate an indirection block

5.5 Directories

- A directory is a file of type T_DIR
- Contains an array associating names and numbers of dinodes
  - inum: inode number
  - name: file name
- Inode 1 is necessarily a directory: it is the root directory of the filesystem

Note: dinode.nlink gives the number of times a dinode is referenced from a directory
- file deleted when nlink equals to 0.
5.6 From path to inode

- To find a inode number from the path /e0/../en (see namex() in fs.c)
  1. cur = 1
  2. For i in [0 .. n] (a) Look for the association [inum, name] in the data blocks of the cur dinode such that name is ei
     (b) cur = inum

5.7 File creation and deletion

- To create the file f in the d directory (function create() in sysfile.c)
  1. Find a free inum dinode by finding an inode whose type is 0 in the dinode array (ialloc () in fs.h)
  2. Add the association [inum, f] to d

- To delete the file f from the d directory (sys_unlink() function in sysfile.c)
  1. Delete the entry corresponding to f in d
  2. Decrease alink from f and if alink equals 0
  3. Delete data blocks from file f
  4. Remove the inode f (setting its type to 0)
6.2 Main functions of inodes (1/3)

- **struct inode** `iget(int dev, int inum)`
  - Corresponds to `open()`: returns an inode associated with `[dev, inum]`
  - Increments the inode usage counter (non-evictable)
  - Do not lock the inode and do not read the inode from disk (optimization to avoid disc playback when creates a file)
  - `inode.valid` indicates whether the inode has been read from disk

- **void ilock(struct inode* ip)**
  - Acquires a lock on the inode
  - Read inode from disk if not already read

- **void iunlock(struct inode* ip)**
  - Release the lock on the inode

6.3 Main functions of inodes (2/3)

- **void itruncate(struct inode* ip)**
  - Free all the blocks in the file (size 0)

- **void iupdate(struct inode* ip)**
  - Copy the inode to the disk dinode (technically, via the I/O cache)

6.4 Main functions of inodes (3/3)

- **void iput(struct inode* ip)**
  - Corresponds to `close()`
  - Decreases the inode usage counter
  - If `ipt` drops to 0, the inode can be evicted from the cache and
  - If `nlink` is 0 (the inode is no longer referenced by a directory)
  - Delete data blocks from inode (flush)
  - Mark the inode as free (type = 0)

Note: if you delete a file from a directory (`unlink()`) while the file is still in use (`open`) by a process, the inode is not deleted: it will be when last `close()` when the reference counter drops to 0.

6.5 Open files

- Multiple processes can open the same file
  - Each process has independent read / write permissions
  - Each process has a read cursor, which is independent of that of the other processes
- A file structure opened by `open()` contains:
  - A pointer to an inode
  - Access permissions
  - A reading cursor
6.6 File descriptors

- Each process has an ofile table of open files
  - A descriptor d is an index in this table
  - proc[i].ofile[d] points to an open file
  - proc[i].ofile[d].ip points to inode

Good to know:
- During a fork(), the parent and the child share the open files
- So proc[parent].ofile[d] == proc[child].ofile[d]
- And so, if the father reads, the child read cursor changes
- Useful for setting up pipes

7 What you must remember

- A device driver is just a function (iderw() for example)
- Reads and writes are logged
  - Ensures file system consistency in the event of a crash
  - The kernel has an I/O cache
    - In memory, managed by the kernel
    - Allows to speed up I/O
  - A file system separates
    - The naming (directory) of the file (dinode + data block)
    - The metadata (dinode) of the data blocks
  - A file descriptor is an index in the ofile table
  - proc->ofile[i] is an open file that references an inode
Références


