CSC4508 – Operating Systems

François Trahay & Gaël thomas

2022

Contents

CSC4508 – Operating Systems

CSC4508 – Operating Systems

1

Licence

Ce document est préparé avec des logiciels libres :

- LaTeX : les textes sources sont écrits en LaTeX (`http://www.latex-project.org/`, le site du Groupe francophone des Utilisateurs de TeX/LaTeX est `http://www.gutenberg.eu.org`). Une nouvelle classe et une nouvelle feuille de style basées sur la classe `seminar` ont été tout spécialement dévéloppées: `newslide` et `slideint` (projet fusionforge `slideint`, `https://fusionforge.int-evry.fr/www/slideint/`);
- `emacs`: tous les textes sont édités avec l'éditeur `GNU emacs` (`http://www.gnu.org/software/emacs`);
- `dvips`: les versions PostScript (PostScript est une marque déposée de la société Adobe Systems Incorporated) des transparents et des polycopiés à destination des étudiants ou des enseignants sont obtenues à partir des fichiers DVI (« DeVice Independent ») générés à partir de LaTeX par l'utilitaire `dvips` (`http://www.ctan.org/tex-archive/dviware/dvips`);
- `ps2pdf` et `dvipdfmx`: les versions PDF (PDF est une marque déposée de la société Adobe Systems Incorporated) sont obtenues à partir des fichiers Postscript par l'utilitaire `ps2pdf` (`ps2pdf` étant un shell-script lançant `Ghostscript`, voyez le site de `GNU Ghostscript` `http://www.gnu.org/-software/ghostscript/`) ou à partir des fichiers DVI par l'utilitaire `dvipfmx`;
- `makeindex`: les index et glossaire sont générés à l'aide de l'utilitaire Unix `makeindex` (`http://www.ctan.org/tex-archive/indexing/makeindex`);
- `TeX4ht`: les pages HTML sont générées à partir de LaTeX par `TeX4ht` (`http://www.cis.ohio--state.edu/~gurari/TeX4ht/mn.html`);
- `Xfig`: les figures sont dessinées dans l'utilitaire X11 de Fig `xfig` (`http://www.xfig.org`);
- `fig2dev`: les figures sont exportées dans les formats EPS (« Encapsulated PostScript ») et PNG (« Portable Network Graphics ») grâce à l'utilitaire `fig2dev` (`http://www.xfig.org/userman/-installation.html`);
- `convert`: certaines figures sont converties d'un format vers un autre par l'utilitaire `convert` (`http://www.imagemagick.org/www/utilities.html`) de ImageMagick Studio;
- `HTML TIDY`: les sources HTML générés par TeX4ht sont « beautifiés » à l'aide de `HTML TIDY` (`http://tidy.sourceforge.net`) ; vous pouvez donc les lire dans le source.

Nous espérons que vous regardez cette page avec un navigateur libre: Firefox par exemple. Comme l'indique le choix de la licence GNU/FDL, tous les éléments permettant d'obtenir ces supports sont libres.

Presentation of the class

François Trahay

CSC4508 – Operating systems

2022–2023

2

1 Presentation of the class

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 **[U]** Concurrent programming
   - CI3 **[G]** Synchronization
   - CI4 **[K]** System calls
   - CI5 **[K]** Interruption and scheduling
   - CI6 **[K]** Sprint: finalization of the scheduler
2. Memory
   - CI7 **[U]** Virtual memory
   - CI8 **[K]** Memory Management Unit
   - CI9 **[G]** Architecture
   - CI10 **[K]** Sprint
3. Input/Output
   - CI11 **[U]** Input/Output
   - CI12 **[U]** Synthesis: mini-project
   - CI13 **[K]** File systems
   - CI14 **[K]** Sprint

CI15 Exam (lab)

1.2 Kernel sessions: XV6

\# 4

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

\# 5

Evaluation:

- 20% - Continuous assessment during sprints:
  - "how did you implement this mechanism of the OS?"
  - "what happens if X?"
- 80% - graded lab exam with several parts :
  - course question(s)
  - explain how you implemented an OS mechanism
  - develop an application

1.4 Evaluation of the class

\# 6

- At the end of the class, students evaluate the class.
- Objective: improve the class

Threads

François Trahay

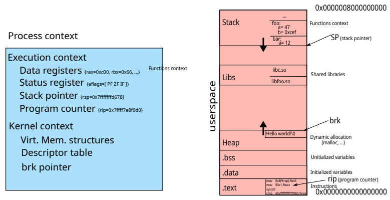
CSC4508 – Operating Systems

2022–2023

1 Execution context of a process

\# 2

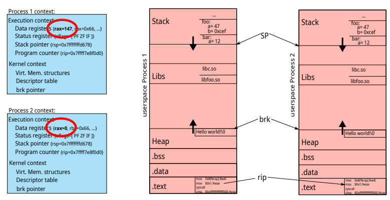
- Context: execution context + kernel context
- Address space: code, data and stack



1.1 Duplicating a process

\# 3

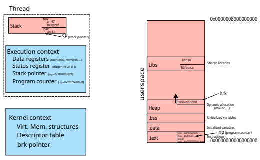
- Fork creates a new process and duplicates
  - Context: execution context + kernel context
  - except for the eax register
  - Address space: code, data and stack



1.2 Execution flows

4

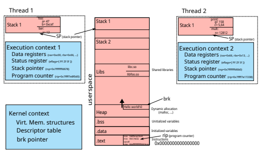
- Execution flow ! = Resources
  - Execution flow (or thread) : execution context + stack
  - Resources: code, data, kernel context



1.3 Multithreaded process

5

- Several execution flows
- Shared resources



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).

1.4 Creating a Pthread

6

Creating a pthread

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
  - `attr` (in): attributes of the thread to be created
  - `start_routine` (in): function to be executed once the thread is created
  - `arg` (in): parameter to pass to the function
  - `thread` (out): identifier of the created thread

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 (*ie.* each process has a parent process), there is no hierarchy between threads.

1.5 Other Pthread functions

7

- `int pthread_exit(void* retval);`
  - Terminates the current thread with the return value `retval`
- `int pthread_join(pthread_t tid, void **retval);`
  - Wait for the `tid` thread to terminate and get its return value

2 Sharing data

\# 8

The memory space is shared between the threads, in particular

- global variables
- static local variables
- the kernel context (file descriptors, streams, signals, etc.)

Some other resources are not shared:

- local variables

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

\# 9

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

\# 10

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:

strtok_example.c

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

void extract_path() {
  char* string = getenv("PATH");
  printf("Parsing '%s'\n", string);


  for(char* token = strtok(string, ":") ;
      token ;
      token = strtok(NULL, ":") ){
    printf("\t %s\n", 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:

strtok_example_bug.c

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

void extract_path() {
  char* string = getenv("PATH");
```

```c
  printf("Parsing '%s'\n", string);
  // string should contain a list of directories separated with :
  // eg. /usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

  // Extract the directories
  // eg. /usr/local/bin, /usr/bin, /bin, /usr/local/games, /usr/games
  for(char* token = strtok(string, ":") ;
      token ;
      token = strtok(NULL, ":") ){
    // token contains a directory (eg. /usr/local/bin)
    printf("\t %s contains: ", token);

    // Extract the subdirectories
    // eg. usr, local, bin
    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 `token = strtok (NULL, ":")` to iterate over the `string` 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, const char *delim);` is `char* strtok_r(char *str, const char *delim, char **saveptr );`

Thus, the previous program can be corrected:

strtok_example_fixed.c

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

void extract_path() {
  char* string = getenv("PATH");
  char* saveptr = NULL;
  printf("Parsing '%s'\n", string);


  for(char* token = strtok_r(string, ":", &saveptr) ;
      token ;
      token = strtok_r(NULL, ":", &saveptr) ){
    printf("\t %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 local bin
```



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

2.3 TLS – Thread-Local Storage

\# 11

- Global variable (or static local) specific to each thread
- Example: `errno`
- Declaring a TLS variable
  - in C11: `_Thread_local int variable = 0;`

TLS variables in C99

Before the C11 standard, using thread-local storage was supported by some compilers using compiler-dependant keywords:

- in C99 with gcc: `__thread int variable = 0;`
- in C99 with Visual studio: `__declspec(thread) int variable = 0;`

`pthread_key`

Another way (more portable, but much more painful to write) to declare a TLS variable is to use a `pthread_key`:

- creation:
  - `int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));`
- terminaison:
  - `int pthread_key_delete(pthread_key_t *key););`
- usage:
  - `void *pthread_getspecific(pthread_key_t key);`
  - `int pthread_setspecific(pthread_key_t key, const void *value);`
- initialisation:
  - `int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));`

3 Synchronization

\# 12

- Guarantee data consistency
  - Simultaneous access to a shared read / write variable
    - `x++` is not atomic (consisting of `load`, `update`, `store`)
  - Simultaneous access to a set of shared variables
    - example: a function `swap(a, b){ tmp=a; a=b; b=tmp; }`
- Several synchronization mechanisms exist
  - Mutex
  - Atomic Instructions
  - Conditions, semaphores, etc. (see Lecture #3)

The following program illustrates the problem of simultaneous access to shared variables. Here, two threads each increment 1 000 000 000 times the same variable:

compteurBOOM.c

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

#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)
```



```c
    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 * 1 000 000 000 = 2 000 000 000`, running this program gives another result, for example:

```
$ ./compteurBOOM
BOOM! compteur = 1076588402
```

3.1 Mutex

\# 13

- Type: `pthread_mutex_t`
- Initialisation:
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `int pthread_mutex_init(ptread_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);`
- Terminaison:
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

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

compteur_mutex.c

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

#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).

14

3.2 Atomic operations

- Operation executed atomically
- C11 defines a set of functions that perform atomic operations
  - `C atomic_fetch_add(volatile A *object, M operand);`
  - `_Bool atomic_flag_test_and_set(volatile atomic_flag *object);`
- C11 defines atomic types
  - operations on these types are atomic
  - declaration: `_Atomic int var;` or `_Atomic(int) var;`

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`.

compteur_atomic.c

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

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

/* INT_MAX / 2 */
#define NBITER 1000000000

_Atomic int counter = 0;

void *start_routine(void *arg) {
  int i;

  for (i = 0; i < NBITER; i++) {
    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);
}
```

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

Concurrent programming

François Trahay

CSC4508 – Operating Systems

2022–2023

1 Introduction

\# 2

Content of this lecture:

- discovering existing synchronization mechanisms
  - inter-process synchronization
  - intra-process synchronization
- studying classic synchronization patterns

2 Inter-process synchronisation

\# 3

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

2.1 Pipes

\# 4

- Special files managed in FIFO
- Anonymous pipes
  - `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 input and output of processes to the different tubes.

2.2 Shared memory

\# 5

- 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

\# 6

- 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`, ca 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 synchronisation

\# 7

- Based on shared objects in memory
- Possible use of IPC

3.1 Mutex

\# 8

- Ensures mutual exclusion
- Type: `pthread_mutex_t`
- Initialisation:
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `int pthread_mutex_init(ptread_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

\# 9

- 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);`
  - waits for a condition to occur.

- `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - unblocks a thread waiting for the condition
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - unblocks all threads waiting for the condition

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

\# 10

- 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 *restrict attr, unsigned count);`
- Waiting:
  - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
    - block until count threads reach `pthread_barrier_wait`
    - unblock all count threads

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);`

11

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

12

4 Classic synchronization patterns

Goals

- Being able to identify classic patterns
- Implement these patterns with proven methods

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.

13

4.1 Mutual exclusion synchronization pattern

- Allows concurrent access to a shared resource
- Principle:
  - Mutex `m` 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)           mutex_lock(m)
 x=read (account)        x=read (account)
 x = x + 10              x = x - 100
 write (account=x)       write(account=x)
mutex_unlock(m)         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.

14

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

```
    Prog Vehicule
...
mutex_lock(m);
  while(cpt == 0){ cond_wait(m);  }
  cpt--;
mutex_unlock(m);
|...
mutex_lock(m);
  cpt++;
  cond_signal(m);
mutex_unlock(m);
```

4.3 Producer / Consumer synchronization pattern

15

- One or more threads produce data
- One or more threads consume the data produced
- Communication via a N blocks *buffer*
  1. Executing `Produc` : produces `info0`

info0				

  2. Executing `Produc` : produces `info1`

info0	info1			

  3. Executing `Conso` : consumes `info0`

	info1			

  4. Executing `Produc` : produces `info2`

	info1	info2		

4.3.1 Implementation of a Producer / Consumer pattern

16

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

```
 Producer:                            Consumer:
repeat                               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
endRepeat
```

Inter-process Producer / Consumer

It is of course possible to implement a producer / consumer scheme between processes using conditions and mutexes. Another simpler 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

17

- 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

18

- 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 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

- a mutex `mutex` to protect concurrent access

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

rw_thread_condition.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 (ie. 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;
}
```

Synchronization

François Trahay

CSC4508 – Operating Systems

2022–2023

2

Outlines

3

1 Introduction

Objectives of this lecture:

- How are synchronization primitives implemented?
- How to do without locks?

If you want to study further synchronization primitives, and to understand memory models, the blog post *"We Make a std::shared_mutex 10 Times Faster"* [1] discusses in details atomic operations, instruction reordering, C++ memory model and various synchronization primitives.

1. https://www.codeproject.com/Articles/1183423/We-Make-a-std-shared-mutex-10-Times-Faster

4

2 Atomic operations

5

2.1 Motivation

- By default, an instruction modifying a variable is non-atomic
- example : `x++` gives :
  - `register = load(x)`
  - `register ++`
  - `x = store (register)`

→ Problem if the variable is modified by a other thread simultaneously

6

2.2 Can't we just use `volatile` ?

- Tells the compiler that the variable can change from one access to another:
  - modification by another thread
  - modification by a signal handler
- But `volatile` does not ensure atomicity

Here is an example of a program that may suffer from overly aggressive optimization by the compiler:

volatile.

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


#if 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
[waits indefinitely]
^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
End of assembler dump.
$
```

We see here that at the address `0x11ca`, the program jumps to the address `0x11ca`. So it jumps in place indefinitely.

This is explained by the fact that the variable `a` 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, see the instruction `0x11c0`), then consults the registry. 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:

```
$ gcc -o volatile volatile.c -Wall -pthread -O2 -DUSE_VOLATILE=1
$ gdb volatile
(gdb) disassemble thread1
Dump of assembler code for function thread1:
   0x00000000000011c0 <+0>:     sub    $0x8,%rsp
   0x00000000000011c4 <+4>:     nopl   0x0(%rax)
   0x00000000000011c8 <+8>:     mov    0x2e76(%rip),%eax        # 0x4044 <a>
   0x00000000000011ce <+14>:    test   %eax,%eax
   0x00000000000011d0 <+16>:    je     0x11c8 <thread1+8>
   0x00000000000011d2 <+18>:    lea    0xe2b(%rip),%rdi        # 0x2004
   0x00000000000011d9 <+25>:    callq  0x1040 <puts@plt>
   0x00000000000011de <+30>:    xor    %eax,%eax
   0x00000000000011e0 <+32>:    add    $0x8,%rsp
   0x00000000000011e4 <+36>:    retq
End of assembler dump.
```

Here, the loop `while (a == 0)` is translated to the lines from `0x11c8` to `0x11d0`. At each loop iteration, the value of `a` is loaded, then tested.

2.3 Atomic operations

\# 7

C11 provides a set of atomic operations, including

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

2.4 Test and set

\# 8

- `_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
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:

test_and_set.c

```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;
```

```
#ifdef NOT_THREAD_SAFE

/* thread-unsafe version */
void do_lock() {
  while(lock) ;
  lock = 1;
}

void do_unlock() {
  lock = 0;
}

#else

/* thread-safe version */
void do_lock() {
  while(atomic_flag_test_and_set(&lock)) ;
}

void do_unlock() {
  lock = 0;
}

#endif  /* NOT_THREAD_SAFE */

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

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

2.5 Compare And Swap (CAS)

- `_Bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);`
  - compares `*obj` and `*expected`
  - if equal, copy `desired` into `*obj` and return `true`
  - else, copy the value of `*obj` into `*expected` and return `false`

\# 9 Performs atomically:

```
bool CAS(int* obj, int* expected, int desired) {
  if(*obj != *expected) {
    *expected = *obj;
    return false;
  } else {
    *obj = desired;
    return true;
  }
}
```


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

cas.c

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

#define NITER 1000000
#define NTHREADS 4

struct node {
  int value;
  struct node* next;
};

struct node *stack = NULL;

#ifdef NOT_THREAD_SAFE

/* thread-unsafe version */
void push(int value) {
  struct node* n = malloc(sizeof(struct node));
  n->value = value;
  n->next = stack;
  stack = n;
}

int pop() {
  struct node* n = stack;
  int value = 0;
  if(n) {
    value = n->value;
    stack = n->next;
    free(n);
  }
  return value;
}

#else

/* thread-safe version */
void push(int value) {
  struct node* n = malloc(sizeof(struct node));
  n->value = value;
  n->next = stack;

  int done = 0;
  do {
    done = atomic_compare_exchange_strong(&stack, &n->next, n);
  } while(!done);
}

int pop() {
  int value = 0;
  struct node* old_head = NULL;
  struct node* new_head = NULL;
  int done = 0;

  do {
    /* Warning: this function still suffers a race condition (search for
     * "ABA problem" for more information).
     * Fixing this would be too complicated for this simple example.
     */
    old_head = stack;
    if(old_head)
      new_head = old_head->next;
    done = atomic_compare_exchange_strong(&stack, &old_head, new_head);
  } while (!done);

  if(old_head) {
    value = old_head->value;
    free(old_head);
  }
  return value;
}

#endif  /* NOT_THREAD_SAFE */


_Atomic int sum = 0;
void* thread_function(void* arg) {
  for(int i=0; i<NITER; i++) {
```

```
    push(1);
  }

  int value;
  while((value=pop()) != 0) {
    sum+=value;
  }

  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

10

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

Performs atomically:

```
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:

fetch_and_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*)

11

- 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 [a])

a. https://en.cppreference.com/w/c/atomic/memory_order

3 Synchronization primitives

12

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 acquire it? Are some threads likely to wait indefinitely?

3.1 Busy-waiting synchronization

13

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

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

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
```



```
   * if flag=1, the lock is taken
   */
  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

\# 14

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

\# 15

- 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:

mutex.c

```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
```



```c
#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, uaddr2, 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++) {
    //    printf("%d\n", 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

16

- condition: a counter

```
struct cond {
  int cpt;
};

void cond_wait(cond_t *c, pthread_mutex_t *m) {
  int value = atomic_load(&c->value);
  pthread_mutex_unlock(m);
  futex(&c->value, FUTEX_WAIT, value);
  pthread_mutex_lock(m);
}

void cond_signal(cond_t *c) {
  atomic_fetch_add(&c->value, 1);
  futex(&c->value, FUTEX_WAKE, 0);
}
```

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

cond.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <assert.h>

#define N 10

int n_loops = 20;

struct cond {
  int cpt;
};
typedef struct cond cond_t;

static int futex(int *uaddr, int futex_op, int val) {
  return syscall(SYS_futex, uaddr, futex_op, val, NULL, uaddr, 0);
}

void cond_init(cond_t *c) {
  c->cpt = 0;
}

void cond_wait(cond_t *c, pthread_mutex_t *m) {
  int cpt = atomic_load(&c->cpt);
  pthread_mutex_unlock(m);
  futex(&c->cpt, FUTEX_WAIT, cpt);
  pthread_mutex_lock(m);
}

void cond_signal(cond_t *c) {
  atomic_fetch_add(&c->cpt, 1);
  futex(&c->cpt, FUTEX_WAKE, 0);
}



struct monitor{
  int value;
  pthread_mutex_t mutex;
  cond_t cond;
};

int infos[N];
```



```
int i_depot, i_extrait;
int nb_produits = 0;
struct monitor places_dispo;
struct monitor info_prete;


void* function_prod(void*arg) {
  static _Atomic int nb_threads=0;
  int my_rank = nb_threads++;

  for(int i=0; i<n_loops; i++) {
    int cur_indice;
    int product_id;
    usleep(100);
    pthread_mutex_lock(&places_dispo.mutex);
    while(places_dispo.value == 0) {
      cond_wait(&places_dispo.cond, &places_dispo.mutex);
    }
    places_dispo.value--;
    cur_indice = i_depot++;
    i_depot = i_depot % N;

    product_id = nb_produits++;
    pthread_mutex_unlock(&places_dispo.mutex);

    usleep(500000);
    printf("P%d produit %d dans %d\n", my_rank, product_id, cur_indice);

    pthread_mutex_lock(&info_prete.mutex);
    infos[cur_indice] = product_id;
    info_prete.value ++;
    cond_signal(&info_prete.cond);
    pthread_mutex_unlock(&info_prete.mutex);
  }
  return NULL;
}


void* function_cons(void*arg) {
  static _Atomic int nb_threads=0;
  int my_rank = nb_threads++;

  for(int i=0; i<n_loops; i++) {
    int cur_indice;
    int product_id;
    usleep(100);
    pthread_mutex_lock(&info_prete.mutex);
    while(info_prete.value == 0) {
      cond_wait(&info_prete.cond, &info_prete.mutex);
    }
    info_prete.value--;
    product_id = infos[i_extrait];
    cur_indice = i_extrait;
    i_extrait = (i_extrait+1) % N;
    pthread_mutex_unlock(&info_prete.mutex);

    usleep(100000);
    printf("C%d consomme %d depuis %d\n", my_rank, product_id, cur_indice);

    pthread_mutex_lock(&places_dispo.mutex);
    places_dispo.value ++;
    cond_signal(&places_dispo.cond);
    pthread_mutex_unlock(&places_dispo.mutex);
  }
  return NULL;
}

void init_monitor(struct monitor *m, int value) {
  m->value = value;
  pthread_mutex_init(&m->mutex, NULL);
  cond_init(&m->cond);
}

int main(int argc, char**argv) {
  init_monitor(&places_dispo, N);
  init_monitor(&info_prete, 0);
  i_depot = 0;
  i_extrait = 0;


  int nthreads_prod=2;
  int nthreads_cons=2;
  pthread_t tid_prod[nthreads_prod];
  pthread_t tid_cons[nthreads_cons];
  int ret;
```

```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;
}
```

17

4 Using synchronization

Classic problems:

- *deadlocks*
- lock granularity
- scalability

18

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 [?])
  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 (eg. with `pthread_mutex_timedlock`)

19

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 (eg. deadlocks, memory corruption)
    - Overhead (locking comes at a cost)

4.3 Scalability of a parallel system

20

- Scalability = ability to reduce execution time when adding processing units
- Sequential parts of a program reduce the scalability of a program (Amdhal's law [?])
- In a parallel program, waiting for a lock introduced sequentiality

→ Locks can interfere with scalability

The notion of scalability is discussed in more detail in the module CSC5001 *High Performance Systems*.

Bibliography

System calls

Gaël Thomas

CSC4508 – Operating Systems

2022–2023

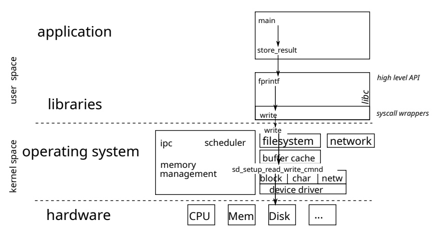
1 Operating systems

2

- Features
  - Offers a unified **programming interface** to the developer
  - Hides hardware implementation details
  - Allows you to run multiple **processes** on a **processor**
- Composition
  - A **library** called **kernel** (*noyau* in French)
    - Unified programming interface (open, fork, etc.)
    - Defined by specifications (System V, POSIX, Win32. . .)
  - + A **set of programs** allowing to interact with the core
    - `ls`, `cp`, `X`, `gnome`, etc.

1.1 Operating systems (2/2)

3



The operating system is responsible for operating various hardware. It, therefore, includes drivers capable of interacting with a particular material. The different drivers for the 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

4

- 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 place much later)
    - see the *fail-fast* approach presented in CSC4102
- `errno`: external variable indicating the cause of the last error
  - The `ERRORS` section in a function manual describes the possible causes of error.

Testimony of a former ASR student: "Without insistence from [the CSC4508 teachers], it would not have jumped out to us so quickly that the problems (in the robotics championship) came from a lack of errors handling on a code that had not been carefully proofread".

How to check the return value of a function and handle errors?

The macro `void assert (scalar expression)` tests the expression passed in parameter and, if false, displays a message error and terminates the program (with the `abort ()` function):

```
  struct stat buf;
  int rc = stat(file, &buf);
  assert(rc>=0);
  // -> in case of an error, prints:
  //   appli: appli.c:12: main: Assertion ‘rc>=0’ failed.
  //   Abandon
```

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

So it is better to test the return code, display a message describing the error, and possibly terminate the process.

```
struct stat buf;
int rc = stat(file, &buf);
if(rc < 0) {
  fprintf(stderr, "Error\n");
  exit(EXIT_FAILURE); // or abort();
}
```

Displaying the cause of an error.

The `errno.h` file lists standard errors. The manual of each system call (see `man 2 function`), and of each function (`man 3 function`) indicates, in the `ERRORS` section, the different error codes that may be returned.

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

```
struct stat buf;
int rc = stat(file, &buf);
if(rc < 0) {
  fprintf(stderr, "Error while accessing file '%s': %s\n", file, strerror());
```



```
  // -> message "Error while accessing file 'plop': No such file or directory"
  exit(EXIT_FAILURE);
}
```

or

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

Generic error handling

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

```
#define FATAL(errnum, ...) do { \
    fprintf(stderr, "Error in %s:%d:\n", __FILE__, __LINE__); \
    fprintf(stderr, __VA_ARGS__); \
    fprintf(stderr, ": %s\n", strerror(errnum)); \
    abort(); \
  } while(0)

int main(int argc, char**argv) {
  char *file = argv[1];
  struct stat buf;
  int rc = stat(file, &buf);
  if(rc < 0) {
    FATAL(errno, "Cannot access file '%s'", file);
  }
  return EXIT_SUCCESS;
}
// affiche:
//  Error in fatal.c:21:
//  Cannot access file 'plop': No such file or directory
//  Abandon
```

Debugger

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

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`:

```
$ ./fatal  plop
Error in fatal.c:21:
Cannot access file 'plop': No such file or directory
Abandon (core dumped)

$ gdb ./fatal core
GNU gdb (Debian 8.1-4+b1) 8.1
[...]
Reading symbols from ./fatal...(no debugging symbols found)...done.
[New LWP 11589]
Core was generated by `./fatal plop'.
```

```
Program terminated with signal SIGABRT, Aborted.
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: Aucun fichier ou dossier de ce type.
(gdb) bt
#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
```

On Linux distribution running systemd, the core dumps are managed by `coredumpctl`:

- **`coredumpctl list`** prints the list of all the available core dumps
- **`coredumpctl info`** display basic information (such as the command line that invoked the program, or the backtrace of the threads when the crash occured) about the last core dump.
- **`coredumpctl debug`** invokes a debugger (eg. `gdb`) on the last core dump

2 Stack frames

 # 5

- Each function call creates an *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)

2.1 Content of a *stack frame*

 # 6

- A *stack frame* is defined by
  - a base address that indicates where the *frame* begins (the `rbp` register on x86)
  - the address of the top of the stack (the `rsp` register on x86)
- Function entry:
  - Save `rbp` (using `push rbp`)
  - Reset `rbp` (using `mov rbp, rsp`)
- Function exit:
  - Restore of the old `rbp` (`pop rbp`)
  - Jump to the return address (`ret`)

Function call convention

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

`x86` 32 bits

On 32-bit `x86` architectures, parameters are placed on the stack so that the first argument is located at address `ebp + 8`, the second at 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`.

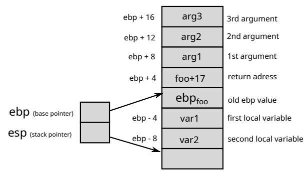


Figure 1: Stack frame on 32-bit `x86` architectures

`x86` 64 bits

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.

`Arm`.

On `Arm` architectures, parameters are passed via registers (`x0` to `x7` on `Arm` 64 bits). The return address is also stored in a register.

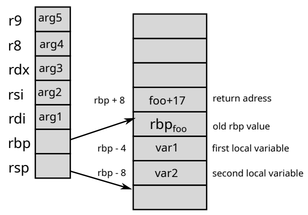


Figure 2: Stack frame on 64-bit `x86` architectures

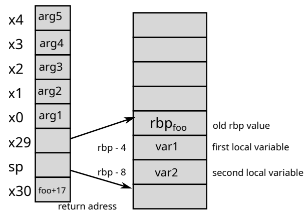


Figure 3: Stack frame on 64-bit `Arm` architectures

2.2 Buffer overflow

\# 7

- (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

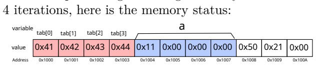
Here is an example of *buffer overflow*:

buffer_overflow.c

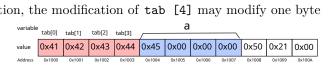
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char**argv) {

  int N = 4;
  char tab[N];
  int a = 17;

  for(int i=0; i<=N ; i++) {
    tab[i] = 'a'+i;
  }

  printf("tab = {%c, %c, %c, %c}\n", tab[0], tab[1], tab[2], tab[3]);
  printf("a = %d\n", a);
  return 0;
}
```

Example

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



During the fifth iteration, the modification of `tab [4]` may modify one byte of the variable `a`:



The variable `a` is therefore no longer equal to 17, but 69 (or 0x45).

Security vulnerabilities

Buffer overflow bugs 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 `a` matches the username, 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 for 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

\# 8

- Using a *buffer overflow* to change the program execution flow
- The return address of a function is on the stack

⟹ possibility of *choosing* the code to be executed afterwards

Example

Here is an example of *stack overflow*:

stack_overflow.c

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

void foo(char* str) {
  char new_str[16];
  strcpy(new_str, str);
  printf("new_str = %s\n", new_str);
}

int main(int argc, char**argv) {

  foo(argv[1]);
  printf("Back in main()\n");
  return 0;
}
```

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`.

Here is an example of execution leading to an *stack overflow*:

```
  $ gdb ./stack_overflow
  (gdb) r coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: stack_overflow coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
new_str = coucouAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x000055555555518e in foo (str=0x7fffffffe03e "coucou", 'A' <repeats 83 times>) at stack_overflow.c:
9       }
(gdb) bt
#0  0x000055555555518e in foo (str=0x7fffffffe03e "coucou", 'A' <repeats 83 times>) at stack_overflo
#1  0x4141414141414141 in ?? ()
#2  0x4141414141414141 in ?? ()
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x0000555555550041 in ?? ()
#10 0x0000000000000000 in ?? ()
(gdb)
```

Here, we observe that when exiting the `foo` function, the program tries to execute the instruction located at the address `0x4141414141414141` (`0x41` is the hexadecimal value of `'A'`), which generates an error.

We could exploit the bug by inserting in `argv [1]` the address of the function `void bar (int a, int b)` as well as its parameters [?].

9

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

10

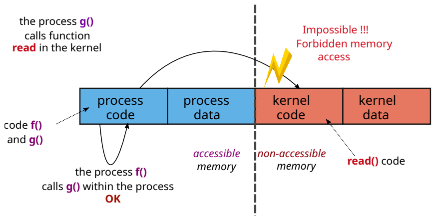
3 User/system interface

- The kernel must *protect* from processes
  - To avoid bugs
  - To avoid attacks
- 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, no direct access to peripherals and instructions that manage the permissions associated with the memory

11

3.1 User/system interface

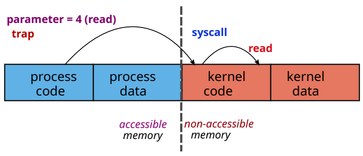
- Problem: how do you call a kernel function when you can't access its memory?



12

3.2 User/system interface

- Solution: special processor instruction called `trap`
  - The kernel associates the address of a `syscall` function to `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`, and `r9` registers;

- The system call number is loaded into 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.

Bibliography

Interruptions and communication

Gaël Thomas

CSC4508 – Operating Systems

2022–2023

2

Outlines

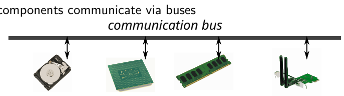
3

1 Communication buses

4

1.1 Communication buses

- Hardware components communicate via buses

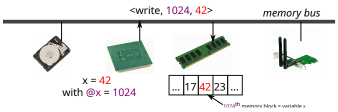


- From a software point of view, 3 main buses
  - Memory bus: mainly to access memory
  - Input / output bus: messages from CPUs to devices
  - Interrupt bus: messages from peripherals to CPUs
- From the hardware point of view: a set of hardware buses with different protocols that can multiplex the software buses

5

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 IO*)

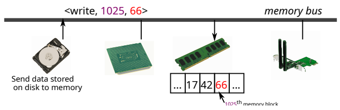


6

1.2.1 DMA: Direct Memory Access

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

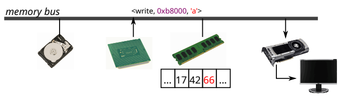
$\Longrightarrow$ The processor can execute instructions during an I/O



7

1.2.2 MMIO: Memory-Mapped IO

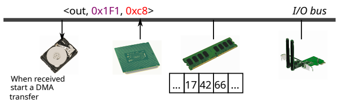
- Processors use memory bus to access devices
  - Sender: a processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)
- Device memory is *mapped* in memory
  - When the processor accesses this memory area, the data is transferred from / to the device



1.3 The input / output bus

8

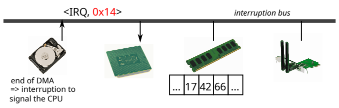
- Request / response protocol, special instructions `in`/`out`
  - Sender: aprocessor
  - Receiver: a peripheral
  - Examples: activate the caps-lock LED, start a DMA transfer, read the key pressed on a keyboard ...



1.4 The interrupt bus - principle

9

- 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 Interruptions

10

2.1 Receiving an interrupt

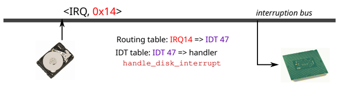
11

- Two tables configured by the kernel to handle reception
  - **Routing table**: associate an **IRQ** with an **IDT** number
  - **IDT table** (*interrupt descriptor table*): associate 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)

12

2.2 Receiving an interrupt: example

- A device sends an **IRQ** (for example `0x14`)
- The **routing table** associates `IRQ14` with `IDT47`
- The **IDT table** indicates that `IDT47` is managed by the function `handle_disk_interrupt`



13

2.3 Receiving an interrupt (continued)

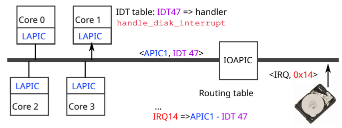
- In the processor, after executing **each instruction**
  - Check if an interrupt has been received
  - If so, find the address of the associated handler
  - Switch to kernel mode and run the interrupt handler
  - Then swicth back to the previous mode and continue the execution
- Note: a handler can be run **anytime**
  - Problem of concurrent access between handlers and the rest of the kernel code
  - Solution: masking interruptions (`cli` / `sti`)

Interruptions and communication 2 Interruptions

14

2.4 Interruptions and multicore processors

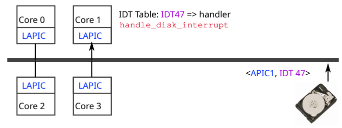
- XAPIC protocol on pentium (x2APIC since Intel Core processors)
  - Each core has a number called APIC number (*Advanced Programmable Interrupt Controller*)
  - Each core handles interrupts via its LAPIC (*local APIC*)
  - An IOAPIC routes an interrupt to a given LAPIC
    - Routing table configured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



15

2.5 MSI: Message Signaling Interrupt

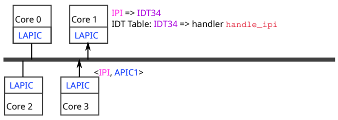
- MSI: direct interrupt from a device to a LAPIC without passing through the IOAPIC
  - The kernel must configure the device so that it knows which LAPIC / IDT pair should be generated
  - Used when the need for performance is important



16

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



17

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
    - a division by zero generates the interrupt `IDT 0x00`, an access illicit memory (`SIGSEGV`) the interrupt `IDT 0x0e` 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

18

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**

Virtual memory

François Trahay

CSC4508 – Operating Systems

2022–2023

1 Introduction

2

- 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

  $\Longrightarrow$ 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 emph swap (disk).
Here are the typical access times to data located in the different types of memory on a "classic"machine (Intel Core i5 *Skylake* processor) in 2017 [1]:

- 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: 60 - 100 ns (100 times slower than L1)
- data on a NVMe disk: 20 μs (20,000 times slower than L1)
- data on an SSD disk: 150 μs (150,000 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):

year	2008	2009	2010	2014	2019	2023
Hard disk drive, 7200 tr/mn (in €/GiB)	0,50	0,32	0,10	0,04	0.027	0.023
SSD disk (in €/GiB)	–	–	–	0,50	0.17	0.09
USB key (in €/GiB)	–	–	1,64	0.62	0.27	0.06
NVMe (in €/GiB)	–	–	–	–	0.21	0.10
RAM (in €/GiB)	–	37,00	21,85	8.75	7.23	2.75

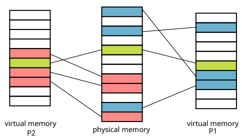
1. Data available in the manual "*Intel 64 and IA-32 Architectures Optimization Reference Manual* "

3

2 Paging

4

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**

5

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)

$\Longrightarrow$ each process has a contiguous memory space to store its data

- The paging mecanism
  - Translates virtual addresses to/from physical addresses
  - Loads the necessary pages (in case of page faults)
  - (Optionally) move active pages to secondary memory

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

6

2.3 Logical (or virtual) address

- Address space is divided using the most significant bits

Logical address on k bits	
Page number	Offset in the page
p bits	($d = (k-p)$ bits)

$\Longrightarrow$ 2^p pages and each page contains 2^{k-p} bytes

- Page size
  - Usually 4 KiB (k-p = 12 bits, so p = 52 bits)
  - *Huge pages*: 2 MiB, or 1 GiB pages
- Choice = compromise between various opposing criteria
  - Last page is half wasted
  - Small capacity memory : small pages
  - Scalability of the page management system

On 64-bit Intel (`x86_64`) or ARM 64 bits architectures (ARMv8), the addresses are stored on 64 bits (i.e. `size (void *)` is 8 bytes), but only 48 bits are usable for virtual addresses.

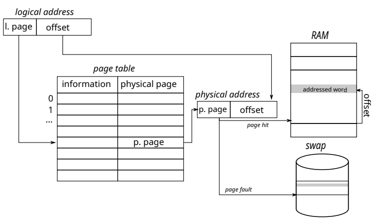
Huge pages

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*.

7

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 ...)

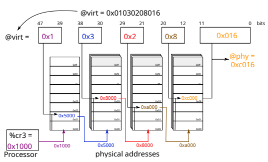


8

2.5 Implementation on a 64-bit pentium

- Page table = 4-levels tree:
  - The physical address of a 512-entry root table is stored in the CR3 register
  - Each entry in a table gives the address of the following table
- @virt decomposed into 4 indexes (n[0..3]) + 1 *offset*, then translated using:

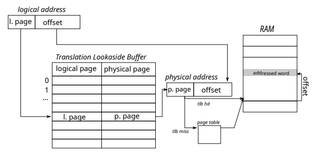
```
uint64_t cur = %cr3;             // cur = root table physical address
for(int i=0; i<3; i++)
  cur = ((uint64_t*)cur)[n[i]]; // physical memory access, next entry
return cur + offset;             // add the offset
```



9

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 N_p compared to the content of each register
  - if found → gives the corresponding frame number N_c
  - Otherwise use the page table



Intel architectures have *Translation Look-aside Buffers* (TLB) with 32, 64, or even 256 entries. TLB are sometimes called *address translation cache*.

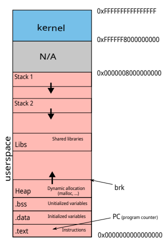
10

3 User point of view

3.1 Memory space of a process

Composed of:

- kernel space
- the different sections of the executed ELF file (`.text`, `.data`, etc.)
- the heap
- the stack (one per thread)
- shared libraries



11

3.2 Memory mapping

How to populate the memory space of a process?

- For each ELF file to be loaded:
  - open the file with `open`
  - each ELF section is *mapped* in memory (with `mmap`) with the appropriate permissions
- Results are visible in `/proc/<pid>/maps`

12

```
$ cat /proc/self/maps
5572f3023000-5572f3025000 r--p 00000000 08:01 21495815   /bin/cat
5572f3025000-5572f302a000 r-xp 00002000 08:01 21495815   /bin/cat
5572f302e000-5572f302f000 rw-p 0000a000 08:01 21495815   /bin/cat
5572f4266000-5572f4287000 rw-p 00000000 00:00 0          [heap]
7f33305b4000-7f3330899000 r--p 00000000 08:01 22283564   /usr/lib/locale/locale-archive
7f3330899000-7f33308bb000 r--p 00000000 08:01 29885233   /lib/x86_64-linux-gnu/libc-2.28.so
7f33308bb000-7f3330a03000 r-xp 00022000 08:01 29885233   /lib/x86_64-linux-gnu/libc-2.28.so
[...]
7f3330ab9000-7f3330aba000 rw-p 00000000 00:00 0
7ffe4190f000-7ffe41930000 rw-p 00000000 00:00 0          [stack]
7ffe419ca000-7ffe419cd000 r--p 00000000 00:00 0          [vvar]
7ffe419cd000-7ffe419cf000 r-xp 00000000 00:00 0          [vdso]
```


3.3 Memory allocation

- `void* malloc(size_t size)`
  - Returns a pointer to an 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 nmemb, 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

13

- 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, `mallopt` 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 reentrancy problems.
- When freeing a buffer with `free`, it is strongly advised to set the pointer to `NULL`. This allows the program to crash immediatly if, by mistake, we access this (now inexistant) buffer again using this pointer.

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

resetToNULL.c

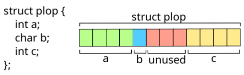
```c
/**********************/
/* resetToNULL.c */
/**********************/

/* This program illustrates the value of assigning a variable to NULL
   which contains a pointer to an unallocated buffer.
   Indeed, it makes a segmentation fault, which makes it possible to
   identify that we have executed an illegal operation. Using a
   debugger allows to understand the problem.
*/
/*                                                                  */
/* 1) Compile the program with the option -g                        */
/*    cc -g -o resetToNULL resetToNULL.c                            */
/* 2) ./resetToNULL                                                 */
/*    ==> Segmentation fault                                        */
/* 3) ulimit -c unlimited                                           */
/* 4) ./resetToNULL                                                 */
/*    ==> Segmentation fault (core dumped)                          */
/* 5) ddd ./resetToNULL core                                        */

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

```
void h(char *p){
  *p = 'a';
}

void g(char *p){
  h(p);
}

void f(char *p){
  g(p);
}

int main(){
  char *p = NULL;

  f(p);

  p = malloc(1);
  assert(p != NULL);

  f(p);

  free(p);
  p = NULL;

  f(p);

  return EXIT_SUCCESS;
}
```

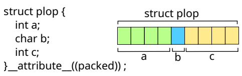
14

3.4 Memory alignment

- Memory alignment depends on the type of data
  - `char` (1-byte), `short` (2-bytes), `int` (4-bytes), ...
- A data structure may be larger than its content



- A data structure can be packed with `__attribute__((packed))`



Memory alignment applies to variables as well as to members of data structures.
The following program illustrates how alignement affects the size of a data structure:

memory_alignment.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

struct plop {
  int a;
  char b;
  int c;
};

struct plop_packed {
  int a;
  char b;
```



```
  int c;
} __attribute__((packed));


int main(void) {

  struct plop p1;
  struct plop_packed p2;
  printf("struct plop -- size: %lu bytes, address: %p\n",
         sizeof(struct plop), &p1);
  printf("\t.a -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.a), &p1.a, offsetof(struct plop, a));
  printf("\t.b -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.b), &p1.b, offsetof(struct plop, b));
  printf("\t.c -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p1.c), &p1.c, offsetof(struct plop, c));
  printf("\n");

  printf("struct plop_packed -- size: %lu bytes, address: %p\n",
         sizeof(struct plop_packed), &p2);
  printf("\t.a -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.a), &p2.a, offsetof(struct plop_packed, a));
  printf("\t.b -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.b), &p2.b, offsetof(struct plop_packed, b));
  printf("\t.c -- size: %lu bytes, address: %p, offset: %lu\n",
         sizeof(p2.c), &p2.c, offsetof(struct plop_packed, c));
  printf("\n");
  return 0;
}
```

```
$ ./memory_alignment
struct plop -- size: 12 bytes, address: 0x7ffc05ad8184
        .a -- size: 4 bytes, address: 0x7ffc05ad8184, offset: 0
        .b -- size: 1 bytes, address: 0x7ffc05ad8188, offset: 4
        .c -- size: 4 bytes, address: 0x7ffc05ad818c, offset: 8

struct plop_packed -- size: 9 bytes, address: 0x7ffc05ad817b
        .a -- size: 4 bytes, address: 0x7ffc05ad817b, offset: 0
        .b -- size: 1 bytes, address: 0x7ffc05ad817f, offset: 4
        .c -- size: 4 bytes, address: 0x7ffc05ad8180, offset: 5
```

15

3.5 The libc point of view

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

16

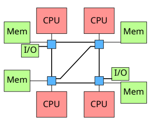
4 Memory allocation strategies

17

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 distant *memory banks*

⟹ *Non-Uniform Memory Access*
⟹ On which memory bank to allocate data?



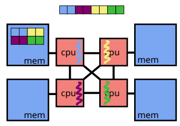
18

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 will use it in the future

first_touch.c

```
double *array = malloc(sizeof(double)*N);

for(int i=0; i<N; i++) {
  array[i] = something(i);
}

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```



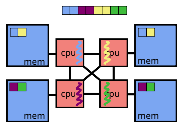
19

4.3 *Interleaved* allocation strategy

- Pages are allocated on the different nodes in a *round-robin* fashion
- Allows load balancing between NUMA nodes
- `void *numa_alloc_interleaved(size_t size)`

interleaved.c

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

for(int i=0; i<N; i++) {
  array[i] = something(i);
}

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```



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

Virtual memory

20

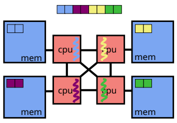
4.4 mbind

- `long mbind(void *addr, unsigned long len, int mode, const unsigned long *nodemask, unsigned long maxnode, unsigned flags)`
- Place a set of memory pages on a (set of) NUMA node

⟹ allows manual placement of memory pages

manual.c

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

#pragma omp parallel for
for(int i=0; i<N; i++) {
  double value = array[i];
  /* ... */
}
```



Architecture

François Trahay

CSC4508 – Operating Systems

2022–2023

2

Outlines

3

1 Introduction

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-optimized code. 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

\# 4

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

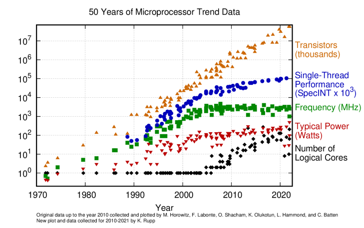
⟹ Increased processor performance

Since 2005

- The fineness of engraving continues to decrease (but less quickly)
- The clock frequency no longer increases due to heat dissipation
  - Heat dissipation depends on the frequency, and the number of transistors
- Multiple computing units per processor

1.2 Evolution of processors performance

\# 5



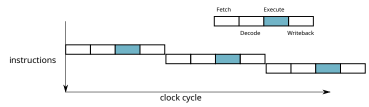
Source: `https://github.com/karlrupp/microprocessor-trend-data`

2 Sequential processor

\# 6

- 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



The number of steps required to execute an instruction depends on the processor type (Pentium 4: 31 steps, Intel Haswell: 14-19 steps, ARM9: 5 steps, etc.)

3 Pipeline

\# 7

- Instruction pipeline
- At each stage, several circuits are used

→ One instruction is executed at each cycle

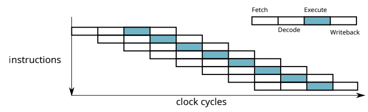


Figure 1: Execution of instructions on a processor with *pipeline*

3.1 Micro architecture of a pipeline

\# 8

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

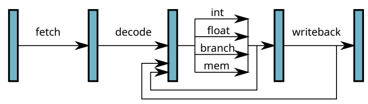


Figure 2: Micro-architecture of a *pipeline*

3.2 Superscalar processors

\# 9

- Use of different functional units simultaneously

$\Longrightarrow$ several instructions executed simultaneously!

- Require to load and decode several instructions simultaneously

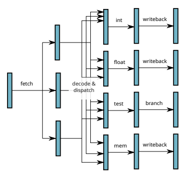
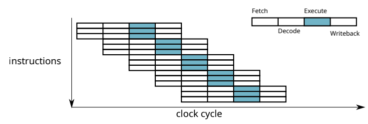


Figure 3: Micro-architecture of a *superscalar* processor

3.2.1 Superscalar processors

\# 10



3.2.2 Dependence between instructions

\# 11

Limitations of the superscalar:

- There should be no dependency between statements executed simultaneously.
  - Example of non-parallelizable instructions

```
a = b * c;
d = a + 1;
```

- Degree of parallelism of the instructions: *Instruction Level Parallelism* (ILP)
- Instructions executed in parallel must use different functional units

3.3 Branching

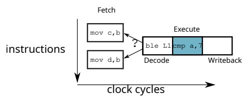
\# 12

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

```
      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 "

⟹ 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

\# 13

- The processor implements a prediction algorithm
- General idea:
  - For each conditional jump, store the previous results

```
0x12 loop:
        ...
0x50    inc eax
0x54    cmpl eax, 10000
0x5A    jl loop
0x5C end_loop:
        ...
```

addr	*branch history*
0x23	0011
0x42	1000
0x5A	1111
0x7E	0000

The branch prediction algorithms implemented in modern processors are very advanced and reach a 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 [1], the `PAPI_BR_PRC` and `PAPI_BR_MSP` counters give the number of conditional jumps correctly and incorrectly predicted.

Linux **perf** also allows collects this information (among others). For example:

1. http://icl.cs.utk.edu/projects/papi/


```
$ 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
```

3.5 Vector instructions

\# 14

- 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

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

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 extension to the instruction set: SSE2, SSSE3 (128 bit), SSE4, AVX, AVX2 (256 bit), AVX512 (512 bit). The other types of processors also provide vector instructions sets (eg NEON [128 bits], or Scalable Vector Extension [SVE] on ARM).

Vector instruction 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:

```
$ 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
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
 pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
 pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl
 xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64
 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
 sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
 f16c rdrand lahf_lm abm ida arat epb pln pts dtherm tpr_shadow vnmi
 flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms
 invpcid xsaveopt
bugs :
bogomips : 5387.82
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
[...]
```

The `flags` field contains the list of all the *capabilities* of the processor, especially the available instructions sets: `mmx, sse, sse2, ssse3, sse4_1, sse4_2, avx2`.

Vector instruction 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 instructions sets, it is recommended to leave the compiler optimize the code, for example using the `-O3` option.

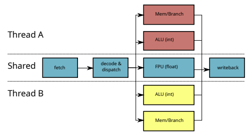
15

4 Parallel Processing

16

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 (eg 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 miss-prediction becomes less serious since while the pipeline of the thread is emptied, another thread can be scheduled.

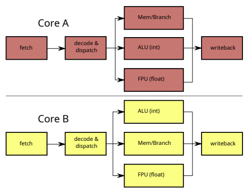
The performance gain when multiple threads are running is not systematic since some circuits remain shared (by example, the FPU).

17

4.2 Multi-core processors

- Limited scalability of SMT
  - *dispatcher* is shared
  - FPU is shared

→ Duplicate all the circuits

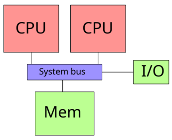


It is of course possible to combine multi-core with SMT. Most semiconductor foundries produce multi-core SMT processors: Intel Core i7 (4 cores x 2 threads), SPARC T3 Niagara-3 (16 cores x 8 threads), IBM POWER 7 (8 cores x 4 threads).

4.3 SMP architectures

\# 18

- *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

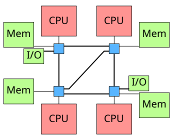


4.4 NUMA architectures

\# 19

- 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*



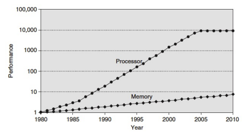
The first NUMA machines (in the 1990s) were simply sets of machines linked by a proprietary network responsible for managing memory transfers. Since 2003, some motherboards allow to plug several Opteron processors (AMD) connected with a HyperTransport link. Intel subsequently developed a similar technology (*Quick Path Interconnect*, QPI) to connect its Nehalem processors (released in 2007).

5 Memory hierarchy

\# 20

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, performance was limited by the performance of the processor. From the software point of view, developers had to minimize the number of instructions to be executed in order to achieve the best performance.

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. A 18-core processor with hyper-threading (ie 36 threads) running at 3.1 Ghz [1] can therefore generate $2 \times 36 \times 3.1 \times 10^9 = 223.2$ billion memory references per second. If we consider access to 64-bit data, this represents 1662 GiB/s (1.623 TiB/s). In addition to these data accesses, the memory access to the instructions (up to 128 bits per instruction) also have to be taken into account. We thus arrive to a 3325 GiB/s (therefore 3.248 TiB/s !) maximum flow.

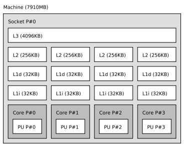
For comparison, in 2023 a DDR5 RAM DIMM has a maximum throughput of around 70 GiB/s. It is therefore necessary to set up mechanisms to prevent the processor from spending all its time waiting for memory.

1. Example: an Intel Xeon Gold 6254 released in 2019

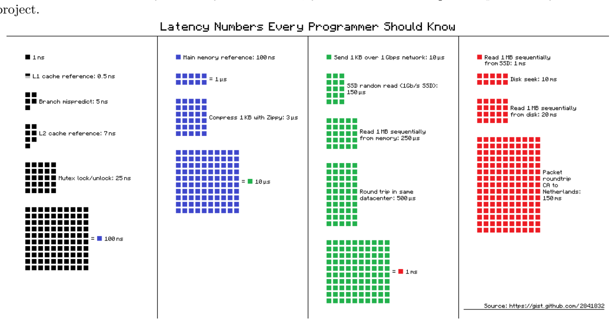
5.1 Cache memory

21

- 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.

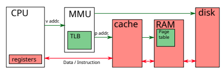


Source: `https://gist.github.com/jboner/2841832`

5.2 Memory Management Unit (MMU)

22

- 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

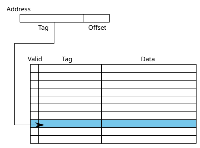


5.2.1 *Fully-associative* caches

23

- 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
```

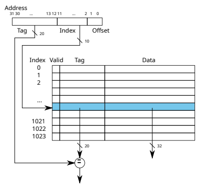
5.2.2 *Direct-mapped* caches

\# 24

- 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: *0x12345***67**8 and *0xbff72***67**8

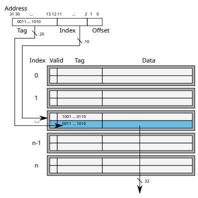


5.2.3 *Set-associative* caches

\# 25

- *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*)



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.

5.2.4 Cache consistency

\# 26

- 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

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

For (many) more details, read the books [**?**] and [**?**] which describe in detail the architecture of computers. If you are looking for specific details, read [**?**].

Bibliography

Input/output

FRANÇOIS TRAHAY

CSC4508 – OPERATING SYSTEMS

2022–2023

Outlines

In this lecture, we mainly talk about files, as this is the easiest example of I/O to manipulable. 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 the files:
  - text: containing bytes that can be displayed on the screen. This type of files 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 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 *offset*.
  - Indexed sequential: the file contains records, each record being identified by a key (unique or no). 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 ...).

3

1 Buffered / non-buffered IO

- Buffered I/O
  - Write operations are grouped in a *buffer* which is written to disc from time to time
  - When reading, a data block is loaded from disk to *buffer*

  $\rightarrow$ a buffered I/O $\neq$ an operation on the disk
  - eg. `fopen`, `fread`, `fscanf`, `fwrite`, `fprintf`, etc.
  - Data stream identified by an *opaque pointer* `FILE*`
- Unbuffered I/O
  - an unbuffered I/O = an operation on the disk †
  - eg. `open`, `read`, `write`, etc.
  - 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 no.

4

2 I/O primitives

5

2.1 File open / close

- `int open(const char *path, int flags, mode_t mode)` : retour = f_id

  `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 ($mode\ \&\ \sim umask$)
  - `O_SYNC`: open file in synchronous write mode
  - `O_NONBLOCK` (ot `O_NDELAY`): open and subsequent operations performed on the descriptor will be non-blocking.
- `int close(int desc)`

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 writen 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 to 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 disc:
  - 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 `int fsync(int fd)` primitive

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

`int creat(const char *path, mode_t mode)` : return value = f_id
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

\# 6

- `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 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 data packets smaller than the size that is requested.

2.3 Writing on a file descriptor

\# 7

- `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 specify 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. No space left on device)

Writing to disk is atomic: if two processes P_1 and P_2 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 P_1,

- either the data written by P_2,
- but never a mix of the data written by P_1 and P_2.

Note that when the file is opened with the option `O_APPEND`, if P_1 and P_2 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 P_1 followed by the one written by P_2,
- or the data written by P_2 followed by the one written by P_1.

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:

```
lseek(fd,0,SEEK_END); /* move the cursor to the end of file */
write(fd,data,taille);
```

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

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

copy.c

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

#define USAGE "USAGE: copy src dest\n"
#define WRITE_ERROR "write error (no space left on device ?)\n"

int source, dest;
int buf;
int nb_read, nb_written;

int main(int argc, char *argv[]) {
  if (argc != 3) {
    write(STDERR_FILENO, USAGE, strlen(USAGE));
    return EXIT_FAILURE;
  }
  source = open(argv[1], O_RDONLY);
  if (source < 0) {
    perror(argv[1]);
    return EXIT_FAILURE;
  }
  dest = open(argv[2],
              O_WRONLY|O_CREAT|O_TRUNC,
              S_IRWXU|S_IRWXG|S_IRWXO);
  if (dest < 0) {
    perror(argv[2]);
    return EXIT_FAILURE;
  }
  while ((nb_read = read(source, (void*)&buf, sizeof(buf))) > 0) {
    nb_written = write(dest, (void*)&buf, nb_read);
    if (nb_written <= 0) {
      if (nb_written == 0) {
        write(STDERR_FILENO, WRITE_ERROR, strlen(WRITE_ERROR));
      }
      else {
        perror("write");
      }
      return EXIT_FAILURE;
    }
  }
  if (nb_read < 0) {
    perror("read");
    return EXIT_FAILURE;
  }
  if (close(source) < 0) {
    perror(argv[1]);
    return EXIT_FAILURE;
  }
  if (close(dest) < 0) {
    perror(argv[2]);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

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 client 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 write them to `out_fd` (which must match an *socket*). `sendfile` is more more efficient than the combination `read` / `write`.

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

8

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)`

9

3 I/O and concurrence

10

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 $[integer : \infty]$
- Locks have a type:
  - `F_RDLCK`: allows concurrent read access
  - `F_WRLCK`: exclusive access

The `exclusive_lock.c` file illustrates exclusive file locking:

exclusive_lock.c

```c
/***********/
/* exclusive_lock.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 15th 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...\n",
         getpid());
  if (fcntl(fd, F_SETLKW, &lock) < 0){
    perror("Acquiring lock");
    exit(EXIT_FAILURE);
  }
  printf("... Exclusive lock acquired by process %d\n", 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...\n", 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\n");

  return EXIT_SUCCESS;
}
```

The **shared_lock.c** file illustrates the shared locking:

shared_lock.c

```
/****************/
/* shared_lock.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);
  }

  /* 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...\n",
         getpid());
  if (fcntl(fd, F_SETLKW, &lock) < 0){
    perror("Acquiring lock");
    exit(EXIT_FAILURE);
  }
  printf("... shared lock acquired by process %d\n", 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...\n", 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\n");

  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.
  - we 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.

11

3.2 Offset manipulation

- `off_t lseek(int fd, off_t unOffset, int origine)` : return = new offset
  allows you to handle the *offset* of the file
- Warning ! Race condition if several threads manipulate the file
- Solutions:
  - Handling of the file in mutual exclusion
  - Using `pread` or `pwrite` instead of `lseek + read` or `lseek + write`

12

4 Improving the I / O performance

4.1 Giving advices to the kernel

13

- `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 programm 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

14

- `int aio_read(struct aiocb *aiocbp)`
- `int aio_write(struct aiocb *aiocbp)`
- Starts an asynchronous read / write operation
- Returns immediately
- `int aio_suspend(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

15

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - "map" a file in memory
  - memory accesses 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

Gaël Thomas

CSC4508 – Operating Systems

2022–2023

2

Outlines

3

1 Device and device driver

4

1.1 Device and device driver

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

5

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)

6

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
  - 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
  - Hard disk, DVD player ...

7

1.4 Block devices in xv6

- A single block device driver in xv6
  - Manages IDE hard disks
  - Function `iderw ()` in `ide.c`
- `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

8

1.5 Principle of the `iderw` algorithm

- `iderw` 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

9

2 The I / O cache

10

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

11

2.2 Principle of an I/O cache

- The system manages a set of *buffers* in memory
- To read a block (read operation)
  - 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. Modifies the contents of the *buffer* in memory
  3. Mark *buffer* as modified (written to disk later)

12

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 *rightarrow* 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()`, `! B_VALID` ⇔ read and `B_DIRTY` ⇔ write

13

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 an *buffer* associated with `(dev,blkno)`
    - Increments a reference counter associated with the *buffer*
    - Locks the *buffer*
    - Return 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)

14

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

15

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)

16

3 The log

17

3.1 Operation versus writing to disk

- A **write operation** of a process often requires **several block writes**
  - 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
  - ...

18

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 $\Longrightarrow$ the file does not exist at the end
    - Deleting a file then adding $\Longrightarrow$ the file exists at the end
    - Similarly, adding data then truncating (size should be 0)
    - ...

19

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!

20

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) == 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

21

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)

22

3.6 Third idea: parallel log

- Problems: Multiple processes may perform transactions in parallel
  - Parallel transaction writes are interleaved in the log

$\rightarrow$ 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

23

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

$\rightarrow$ the system can therefore manage up to 3 copies of a block
  - One on disk in the file system called **disk block**
  - One on disk in the log called **disk log block**
  - One in memory in the buffer cache called **cached block**

24

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**

25

3.9 Using the log

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

```
begin_op()
log_write(b1)
log_write(b2)
...
end_op()
```


26

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()`)

27

3.11 Implementation in xv6 (2/3)

- `void log_write(struct buf* b)`
  - Add the block associated with `b` to the log
  - Add block number to **memory log**
  - Mark buffer as `B_DIRTY` $\Longrightarrow$ does not leave the cache (see `bget()`)

28

3.12 Implementation in xv6 (3/3)

- After a crash, call `install_trans()` which propagates the writes from **disk log** to file system
  - In the worst case, writes that had already been performed are replayed
  - But at the end of the replay, the filesystem is in a consistent state

29

4 Partitions and file systems

30

4.1 File system

- File system: defines the structure for storing files (often for a block type device)
  - UFS : Unix Files 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 !

31

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

32

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
  - Blocks 2 to x: kernel loader
    - 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...

33

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`

34

5 UFS/xv6 file system

35

5.1 Overall file system structure

- Five large contiguous zones (in `fs.h`)
  - The **super block** describes the other areas
  - The **journal** contains the disk logs
  - The **dinode table** contains the metadata of the files (size, type like ordinary or directory ...)
  - The **table of free blocks** indicates the free blocks
  - The **data blocks area** contains the data of the files

36

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

37

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 [i]` block contains the 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 `ind [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.

38

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** (function `balloc()` in `fs.h`)
  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

39

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 $\Longrightarrow$ file deleted when `nlink` equals to 0.

40

5.6 From path to inode

- To find a dinode 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`

41

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. Decrement `nlink` from `f` and if `nlink` equals 0
  3. Delete data blocks from file `f`
  4. Remove the inode `f` (setting its type to 0)

42

6 xv6 I/O stack

43

6.1 Inode

- **inode** = memory cache of a **dinode**
  - Enter the cache at `open()`
  - Can be evicted from cache from `close()`
  - Contains the fields of the dinode
  - \+ fields to know which dinode the inode corresponds to
    - Device number and dinode number
  - \+ fields required when the dinode is used
    - A lock to manage concurrent access
    - A counter giving the number of processes using the inode to know when the inode can be evicted from the cache
- **Inode table** = table which contains the inodes

44

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

45

6.3 Main functions of inodes (2/3)

- `void itrunc(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)

46

6.4 Main functions of inodes (3/3)

- `void iput(struct inode* ip)`
  - Corresponds to `close ()`
  - Decreases the inode usage counter
  - If cpt 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 (itrunc)
      - 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.

47

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

48

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

49

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
  - Is in memory, managed by the kernel
  - Allows to speed up I/O
- A file system separates
  - The naming (directory) of the files (dinodes + data blocks)
  - 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

[Aleph, 1996] Aleph, O. (1996). Smashing the stack for fun and profit. *Phrack #49.*

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM.

[Bryant and O'Hallaron, 2003] Bryant, R. E. and O'Hallaron, D. R. (2003). *Computer Systems: A Programmer's Perspective.* Prentice Hall.

[Coffman et al., 1971] Coffman, E. G., Elphick, M., and Shoshani, A. (1971). System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78.

[Patterson, 2011] Patterson, D. A. (2011). *Computer architecture: a quantitative approach.* Elsevier.

[Patterson and Hennessy, 2013] Patterson, D. A. and Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface.* Newnes.

Index