System and Device Programming
Prof. Laface Pietro
Lectures notes

Enrico Franco

April 28, 2018

Contents

Chapter 1

Processes and concurrency

A *process* is a sequence of operations performed by a program in execution on a given set of input data. A process is a dynamic entity. It is characterized by a **Process Control Block**.

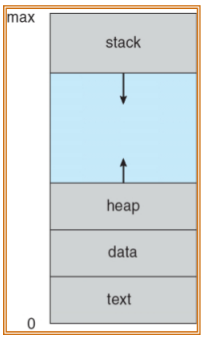

Figure 1.1: Process Control Block

Text contains the code. This part of memory cannot be modified;

Data contains global variables;

Heap contains dynamic variables. It grows "upward";

Stack contains local variables and return addresses of functions. The *Stack Pointer* point to the top of the stack. It grows "downward", so in the opposite direction to the Heap.

It is possible to conceptually image the **trace** of a process as a table containing values of variables and registers (stack pointer and program counter, too) at any time. In this way it is possible to know the state of the process at any instruction. In this way it is possible to perform **context switching**, the

operation performed by the kernel which move the attention of the CPU from a process to a different one, restoring its state as if it was never interrupted.

Every process has a unique identifier which is called **PID**, a unique non negative integer. Although a PID is unique, UNIX reuses the numbers of terminated processes.

Library `unistd.h` contains system calls to retrieve PID:

```
pid_t getpid();    // Process ID
pid_t getppid();   // Parent process ID
uid_t getuid();    // Get the real user ID
gid_t geteuid();   // Get the effective ID
```

1.1 Concurrency

Sequential processes are deterministic. *Cooperating processes* have precedence constraints in order to guarantee a specific order in the process execution. Hence, processes are not more independent and some typical problem arise such as deadlock and synchronization.

1.1.1 Process creation

System call `fork` creates a new **child** process which is a perfect clone of the parent process. Those processes have the same initial value of variables and they share code and file pointers of open files (because they point to the kernel memory) but they do not share anything else (different stacks, different heaps, different data sections).

`fork()` returns two different values:

- The parent process receives the child PID: in this way the parent process can recognize every single child;
- The child process receives the value 0: it does not have to receive the parent PID because it can obtain it through the system call `getppid`;
- -1 in case of error.

Library `unistd.h` contains the system call `fork()`:

```
pid_t fork(void);
```

1.1.2 Process termination

When a process terminates, the kernel sends a `SIGCHLD` to its parent. Receiving a signal is an asynchronous event and the parent process may

- Manage the child termination:
  - System call `wait`;

  - System call `waitpid`;
  - Using a signal handler for signal `SIGCHLD`.

- Ignore the event (default behavior).

System call `wait` blocks the calling process if all its children are running (none is already terminated). `wait` will return as soon as one of its children terminates. It returns an error if the calling process has not children.

If a parent needs to wait a specific child it is better to use `waitpid`, which suspends execution of the calling process until the child, specified by `pid` argument, has changed state. By default `waitpid` waits only for terminated children.

Library `sys/wait.h` contains the system call `wait` and `waitpid`

```
pid_t wait(int *statLoc);
pid_t waitpid(pid_t pid, int *statLoc, int options);
```

Parameter `statLoc` is used a return parameter which stores information about the termination of the process. There are macros to interpret its value.

1.2 Move the execution

1.2.1 System call exec

System call `exec` substitutes the process code with the executable code of another program. The new program begins its execution as usual from the main. In particular, the system call `exec` *does not create a new process*, in fact it substitutes the calling process image with the image of another program and the PID does not change.

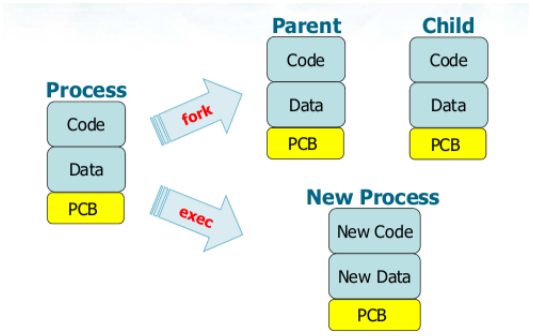

Figure 1.2: Different behaviors of `fork` and `exec`

There are 6 different versions of `exec` system call: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve` distinguishable by the letters:

- l (list): arguments are a list of strings. The last element must be a `NULL` pointer. Or equivalently, `(char *) 0`.;

- v (vector): arguments is a vector of strings. The last element must be a `NULL` pointer. Or equivalently, `(char *) 0`.;
- p (path): the executable filename is looked for in the directories listed in the environment variable `PATH`;
- e (environment): the last argument is an environment vector `envp[]` which defines a set of new association strings `name = value`.

Library `unistd.h` contains all the implementations of system call `exec`. They all return -1 in case of error:

```
int execl(char *path, char *arg0, ..., (char *)0);
int execlp(char *name, char *arg0, ..., (char *)0);
int execle(char *path,const char *arg0,...,char *envp[]);
int execv(char *path, char *argv[]);
int execvp(char *name, char *arg[]);
int execve(char *path, char *arg[], char *envp[]);
```

Arguments

- Pathname of the executable file: in the 'p' versions the complete path is not necessary. The file must be in one of the directories listed in the environment variable `PATH`. Remember that, for security reasons, current directory is not in `PATH`;
- Its argument list: the first argument is the *name* (alias) of the process (its `argv[0]`). The other arguments of the list are argument for the executable;
- Possibly the environment vector.

UNIX shell skeleton

Command in foreground

```
while(TRUE) {
    write_prompt;
    read_command(command, parameters);
    if(fork() == 0)
        /* Child: Execute command */
        execve(command, parameters);
    else
        /* Parent: Wait child */
        wait(&status);
}
```

Command in background (option &)

```
while(TRUE) {
    write_prompt;
    read_command(command, parameters);
    if(fork() == 0)
        /* Child: Execute command */
        execve(command, parameters);
    /* Parent: Continue the execution */
}
```

1.2.2 System call system

System call `system` forks a shell, which executes the string command, while the parent process waits the termination of the shell command.

Library `stdlib.h` contains the system call `system`:

```
int system(const char *string);
```

It returns:

- -1 or 127 in case of error;
- The exit value of the shell that executed the command with the format of `waitpid`.

1.3 Signals

A *signal* is a **software interrupt** (i.e. asynchronous event which may occur at any time) sent to a process to notify it of an event that occurred. A signal can be used as a limited form of inter-process communication or for synchronization. The use of signals is prone to race conditions and they are difficult to manage.

1.3.1 Generating a signal

The UNIX command `KILL -SIGUSR1 4481` is used to send the `SIGUSR1` signal to process with PID 4481. Possibly signals are, `SIGUSR1`, `SIGUSR2`, `SIGALRM`, `SIGCHLD` (sent from a child process to its parent when it terminates) and `SIG_IGN` (used to ignore a specific signal).

System call `kill` is used to send a signal `signo` to a process (or group of processes) identified by the parameter `pid`.

Library `signal.h` contains the system call `kill`:

```
int kill(pid_t pid, int signo);
```

1.3.2 Reacting to a signal

A process can:

- ignore/discard the signal. Not possible for `SIGKILL` and `SIGSTOP` which are unmaskable;
- execute a signal handler function.

To properly react to the asynchronous arrival of a given type of signal, a process must inform the kernel about the action that it will perform when it will receive a signal of that type. Precondition to properly handle a received signal for a process is to declare to the kernel if a signal of a given type will be ignored or caught.

System call `signal` is used to inform the kernel about the reaction, specified as an address of a function `func` to execute when a specific signal `sig` is received.

Library `signal.h` contains the system call `signal`:

```
void (*signal (int sig, void (*func)(int)))(int);
```

The function `func` has this prototype: `void func(int signo)`. In this way, the same function can be used to manage different signals which can be recognized inside the signal handler.

Skeleton of a shell

Command in background (option &)

```
while(TRUE) {
    write_prompt;
    read_command(command, parameters);
    if(fork() == 0)
        /* Child: Execute command */
        execve(command, parameters);
    /* Parent: Continue the execution */
}
```

Actually, this naive solution will not work because every child process becomes a zombie, given that there is no process waiting for it.

Hence, it is needed to allocate a signal handler which ignores the `SIGCHLD` generated on child termination. `signal(SIGCHLD, SIG_IGN);`

1.3.3 Issues

If many signals of the *same* type are waiting to be handled, then most UNIXs will only deliver one of them. So multiple signals are lost.

If many signals of *different* types are waiting to be handled, they are not delivered in any fixed order.

It is good practice to re-allocate the handler at the end of the handler function. In fact, in some UNIX systems, the signal reaction is reset to its default action immediately after the signal has been sent.

If a signal is sent during the execution of a signal handler, the signal is lost. So, the signal handler must be short, in order to receive "all" signals.

1.3.4 Synchronization

The basic idea is to allocate a signal handler and to wait the signal using the system call `pause`. When a signal is received, the process will exit from the pause and will continue its execution.

Another scenario can be obtained by using the system call `alarm`. In this way, the process informs the kernel that it wants to receive a signal (`SIGALRM`) after a given amount of time `secs`. If `secs` is set to 0, the previous alarm is canceled. It is similar to a `sleep` but it does not block the process.

Library `unistd.h` contains both the system calls `pause` and `alarm`:

```
int pause(void);
long alarm(long secs);
```

Chapter 2

Threads

Processes do not share memory and they can only communicate through a file, i.e. a pipe. The child can communicate exiting status to the parent (just a single byte). The creation of a child process is a costly operation in terms of space, which has to be duplicated, and time, because it is a slow operation. The management of multiple processes requires scheduling and an expensive context switching.

A thread is a *lightweight process*. The thread model allows a program to control multiple different flows of operations that overlap in time. Threads creation and management require a reduced costs with respect a process implementation, they share a single address space (i.e. variables) and the context switch among threads is faster.

A thread is statically represented by a function, it has its own stack and therefore its own variables declared inside it.

2.1 POSIX threads

POSIX threads or Pthreads is the standard UNIX library for threads, defined for C language. Using this library, a thread is a function that is executed in concurrency with the main thread.

In order to exploit thread functionalities, library `pthreads.h` must be included and the C program must be compiled with the option `lpthread`.

Pthreads library defines more than 60 functions. The most important and used are:

`pthread_t pthread_self()` returns the thread identifier of the calling thread;

`int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*startRoutine)(void *), void *arg)` allows creating a new thread executing the C function `startRoutine`;

`void pthread_exit(void *valuePtr)` allows a thread to terminate returning

a termination status. This value is kept by the kernel until a thread calls `pthread_join`;

`int pthread_join(pthread_t tid, void **valuePtr)` is used by a thread to wait the termination of another thread, if the thread was declared as *joinable*;

`int pthread_detach(pthread_t tid)` declares the thread `tid` as detached. Therefore, its status information will not be kept by the kernel at the termination and no thread can join with that thread;

`int pthread_cancel(pthread_t tid)` terminates the target thread. It is a dangerous operation because the calling thread does not wait for the termination of target thread, so it is not possible to know the status of the terminated thread.

Thread termination A process, with all its threads, terminates if

- Its thread calls `exit`;
- The main thread execute `return` which is compiled as an `exit`;
- The main thread receives a signal whose action is to terminate.

A single thread can terminate without affecting the other process threads

- Executing `return`;
- Executing `pthread_exit`;
- Receiving a cancellation request performed by another thread.

The kernel does not distinguish which function executed the `exit` statement, so whichever thread calls `exit` will cause the entire process termination. If the main wants to stop and keep alive other threads, it must perform a `pthread_exit` rather than a `return`.

Chapter 3

Synchronization

In concurrent programming where processes or threads cooperate and share data, race conditions may arise and there may be sections of not-reentrant code, it is needed to properly synchronize the cooperating actors to make their results not dependent on their relative speed.

A *Critical Section* or *Critical Region* is a section of code, common to multiple threads, in which they can access shared objects or in which they are competing for the use of shared resources. It is needed to ensure that when one thread is executing in its critical section, no other thread is allowed to execute in its critical section, therefore an access protocol must be established to enter the critical section in **mutual exclusion**:

- A thread executes a "reservation" code before entering a CS;
- The reservation code blocks the thread if another is using its CS;
- Leaving its CS, a thread executes a code to release the CS;
- The release possibly unlocks another thread which was waiting in the "reservation" code of its CS.

In general, the **software solutions** to the problem of CS are complex and inefficient. In fact, test and set operations are "invisible" to the other threads and they are not atomic, thus a thread can react to the presumed value of a variable rather than to its current value. Furthermore, the solutions is not easily extensible to an arbitrary number of threads.

The **hardware solutions** can be easily extensible to a larger number of threads but they introduce busy form of waiting with spin lock leading to possible starvation.

3.1 Semaphores

The hardware solution can be used to implement system calls that can be used for solving every kind of synchronization problem (not only the Mutual Ex-

clusion) avoiding the busy form of waiting. These system calls rely on a data structure introduce by Dijkstra in 1965 called semaphore.

A *semaphore* is a shared structure including a counter, a waiting queue managed by the kernel both protected by a lock. The kernel offers a set of **atomic primitives** that allows a thread to be blocked on the semaphore or to wake up if it was locked.

`init(S, k)` defines and initializes semaphore `S` counter to value `k`. `k` is a positive value because the system calls manage the counter so that, if negative, its absolute value is the number of threads waiting on the semaphore queue.

`wait(S)` decrements the counter and blocks the calling thread if the counter value of `S` is negative or zero.

`signal(S)` increases the counter and made some blocked threads ready to run if the counter value of `S` is negative or zero.

`destroy(S)` releases semaphore `S` memory.

Semaphores can be used to solve any synchronization problem using an appropriate protocol, possibly adding more than one semaphore and additional share variables.

3.1.1 POSIX semaphores

POSIX semaphores define a set of kernel independent system calls, included in header file `semaphore.h` which return -1 on error.

`int sem_init(sem_t *sem, int pshared, usigned int value)` initializes the semaphore counter at value `value`. The `pshared` value identifies the type of semaphore. If it is equal to 0, the semaphore is local to the threads of current process, otherwise the semaphore can be shared between different processes.

`int sem_wait(sem_t *sem)` implements the standard wait, i.e. if the counter is negative or zero, the calling thread is blocked.

`int sem_post(sem_t *sem)` implements the standard signal, i.e. increments the counter and wakes up a blocking thread if the counter is negative or zero.

`int sem_getvalue(sem_t *sem, int *valP)` allows obtaining the value of the semaphore counter. The value is assigned to `*valP`.

`int sem_destroy(sem_t *sem)` destroys the semaphore at the address pointed by `sem`.

`int sem_trywait(sem_t *sem)` implements a non-blocking wait, i.e. if the semaphore counter has a value greater than 0, it performs the decrement and returns 0 while, if the semaphore counter is negative or zero, it returns -1 instead of blocking the caller as `sem_wait` does.

3.1.2 Pthread mutex

Pthread mutex defines binary semaphores of type `pthread_mutex_t` by means of system calls

`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)` initializes the mutex referenced by `mutex` with attributes specified by `attr` (default = `NULL`).

`int pthread_mutex_lock(pthread_mutex_t *mutex)` blocks the caller if the `mutex` is locked or acquires the mutex lock if the `mutex` is unlocked.

`int pthread_mutex_trylock(pthread_mutex_t *mutex)` is similar to a `lock`, but returns without blocking the caller if `mutex` is locked. It returns 0 if the lock has been successfully acquired or a `EBUSY` error if the `mutex` was already locked.

`int pthread_mutex_unlock(pthread_mutex_t *mutex)` releases the `mutex` lock.

`int pthread_mutex_destroy(pthread_mutex_t *mutex)` frees `mutex` memory so that it cannot be used anymore.

3.2 Synchronization protocols with semaphores

3.2.1 Producer & Consumer with limited memory buffer

This problem uses a circular buffer implementing a FIFO queue of size `MAX` for storing the produced elements to be consumed. If the buffer is empty, the consumer has to wait. On the other hand, if the buffer is full, the producer has to wait. In an intermediate situation, producer and consumer can produce and consume in a concurrent fashion.

Access functions Functions without considering synchronization and concurrency.

```
void enqueue(int val) {
  queue[in] = val;
  in = (in + 1) % MAX;
  return;
}
```

```
void dequeue(int *val) {
  queue[out] = val;
  out = (out + 1) % MAX;
  return;
}
```

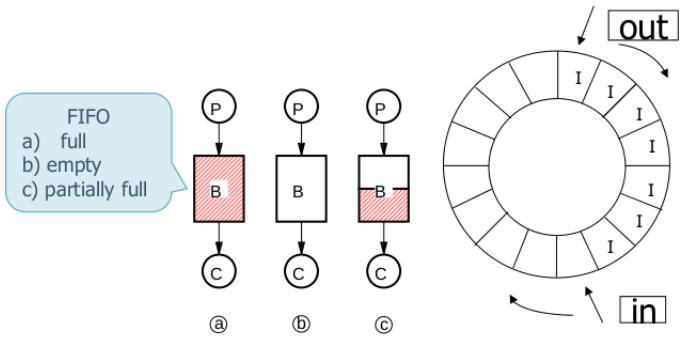

Figure 3.1: Circular buffer

Concurrent access Initially, a consumer has to wait and a producer has to produce, i.e. it has not to be blocked. Therefore, a consumer has to wait on full, initialized to 0 and a producer has to produce at most `MAX` elements. Both signal when an elements is enqueued or dequeued.

```
init(full, 0);
init(empty, MAX);
```

```
Producer() {
  Message m;
  while(TRUE) {
    produce(m);
    wait(empty);
    enqueue(m);
    signal(full);
  }
}
```

```
Consumer() {
  Message m;
  while(TRUE) {
    wait(full);
    m = dequeue();
    signal(empty);
    consume(m);
  }
}
```

Producer and consumer operate on different indexes of the buffer, thus they can operate in concurrency as long as the queue is not full or empty, otherwise either a producer or a consumer is blocked.

The solution is symmetric and it can be easily extended to more than one producer and consumer processes, remembering that operations on the queue are not atomic and multiple consumers or multiple producers must act in mutual exclusion.

Producers & Consumers

```
init(full, 0);
init(empty, MAX);
init(meP, 1);
init(meC, 1);
```

```
Producer() {
  Message m;
  while(TRUE) {
    produce(m);
    wait(empty);
    wait(meP);
    enqueue(m);
    signal(meP);
    signal(full);
  }
}
```

```
Consumer() {
  Message m;
  while(TRUE) {
    wait(full);
    wait(meC);
    m = dequeue();
    signal(meC);
    signal(empty);
    consume(m);
  }
}
```

3.2.2 Readers & Writers

This represents the typical problem of sharing a database between two sets of concurrent threads:

- Readers are allowed to access the database in concurrency;
- Writers must access the database in Mutual Exclusion with other Writers and Readers.

When a writer is writing in the database, several Readers and Writers processes can be blocked waiting the end of the write operation.

Readers precedence: at the end of a writing operation, favour the access of the waiting Readers rather than of the waiting Writers.

Writers precedence: at the end of a writing operation, favour the access of the waiting Writers rather than of the waiting Readers.

Readers priority Giving priority to the Readers means that a Reader does not wait unless a Writer is writing.

While Readers are reading, new Readers are allowed to read and Writers are blocked. The first Reader blocks any Writer and, dually, when the last Reader terminates, a waiting Writer can access the database.

The solution uses:

- A shared variable `nR` that counts the number of Readers inside the critical section;
- A Mutual Exclusion semaphore `meR` to protect variable `nR`;
- A Mutual Exclusion semaphore `w` among Writers or among Readers and Writers;
- A Mutual Exclusion semaphore `meW` among Writers.

```
nR = 0;
init(meR, 1);
init(meW, 1);
init(w, 1);
```

```
Reader() {
  wait(meR);
    nR++;
    if(nR == 1)
      wait(w);
  signal(meR);
  // read
  wait(meR);
    nR--;
```

```
Writer() {
  wait(meW);
  wait(w);
  // write
  signal(w);
  signal(meW);
}
```

```
    if(nR == 0)
      signal(w);
  signal(meR);
}
```

Writers priority Giving priority to the Writers means that a Writer has priority over all Readers.

A Writer trying to enter its critical section blocks *new* Readers, but the Readers that are inside their critical section are allowed to complete their reading task.

The solution uses:

- Two shared variables `nR` and `nW` to count the Readers inside the critical section and the Writers that need to write (one of them possibly writing);
- Two Mutual Exclusion semaphores `meR` and `meW` to protect the variables `nR` and `nW`;
- Two Mutual Exclusion semaphores `r` and `w` to enforce Readers and Writers to wait on different queues.

```
nR = nW = 0;
init(meR, 1);
init(meW, 1);
init(w, 1);
init(r, 1);
```

```
Reader() {
  wait(r);
    wait(meR);
      nR++;
      if(nR == 1)
        wait(w);
    signal(meR);
  signal(r);
  // read
  wait(meR);
    nR--;
    if(nR == 0)
      signal(w);
  signal(meR);
}
```

```
Writer() {
  wait(meW);
    nW++;
    if(nW == 1)
      wait(r);
  signal(meW);
  wait(w);
  // write
  signal(w);
  wait(meW);
    nW--;
    if(nW == 0)
      signal(r);
  signal(meW);
}
```

3.2.3 Single lane tunnel

A tunnel has a single lane, and cars can proceed only in alternate directions. Therefore it is needed to enable any number of cars (threads) to proceed in the

same direction and block traffic in one direction if there is traffic in the opposite one.

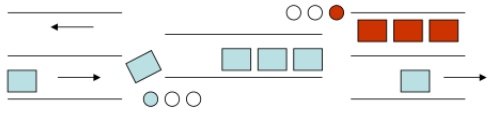

Figure 3.2: Single lane tunnel

This problem is similar the Readers & Writers one, but for two sets of Readers. In its basic implementation can result in starvation of cars in one direction.

The solution uses:

- Two shared count variables `n1` and `n2`, one for each travel direction;
- Two semaphores `s1` and `s2`, one for each travel direction;
- A global semaphore wait `busy`.

```
n1 = n2 = 0;
init(s1, 1);
init(s2, 1);
init(busy, 1);
```

```
left2right() {
  wait(s1);
    n1++;
    if(n1 == 1)
      wait(busy);
  signal(s1);
  // run left to right
  wait(s1);
    n1--;
    if(n1 == 0)
      signal(busy);
  signal(s1);
}
```

```
right2left() {
  wait(s2);
    n2++;
    if(n2 == 1)
      wait(busy);
  signal(s2);
  // run right to left
  wait(s2);
    n2--;
    if(n" == 0)
      signal(busy);
  signal(s2);
}
```

3.2.4 Conditions

A *condition* is a data structure containing a mutual exclusion lock, a condition variable and a data structure to protect because it require synchronization.

```
typedef struct Cond {
  pthread_mutex_t lock;
  pthread_cond_t mycond;
```

```
  int i;
} Cond;
```

`lock` protects the structure;

`mycond` is the condition variable;

`i` is the variable to set or check.

Main thread has to define such a structure, allocate a global structure of that type, initialize its variables and create two threads A and B.

Thread A does work until it needs to check a given condition by calling `pthread_cond_wait` to wait for a signal from thread B, unlocking the lock (otherwise, no one can change the variable content). When signaled, thread A wakes up with lock locked and, after explicitly unlocking the lock, can continues its work.

Thread B does work and locks the condition lock. It changes the value of the global variable associated to the waiting condition and, if thread A wait condition is true, it performs a `pthread_cond_signal`. After unlocking the condition lock, it can continue its work.

Waiting on Condition Variables

`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` blocks the calling thread until the specified condition is signaled. This system call should be called while *lock* is locked, and it will automatically release the *lock* while it waits.

After signal is received and thread is awakened, lock be automatically locked for use by the thread.

The programmer is then responsible for unlocking lock when the thread does not more need it.

Signaling on Condition Variables

`int pthread_cond_signal(pthread_cond_t *cond)` signals another thread which is waiting on the condition variable.

It should be called with the *lock* locked, and must unlock the *lock* to allow `pthread_cond_wait` to complete.

`int pthread_cond_broadcast(pthread_cond_t *cond)` should be used if more than one thread is blocked on the same condition variable, rather than `pthread_cond_signal`. It is a logical error to call `pthread_cond_signal` before calling `pthread_cond_wait`.

3.2.5 Precedence graphs

Goal is to implement a graph using the minimum number of semaphores remembering that signals are sent to a semaphore, not to a process and they are caught by the first process able to receive them.

The first thing to do is to delete arcs which are not necessary and can, therefore, lead to use a non minimum number of semaphores.

Acyclic threads: outgoing arcs require a single semaphore and incoming arcs require a single semaphore.

Cyclic threads: outgoing arcs require different semaphores, while incoming arcs use a single semaphore. In fact, using the same semaphore, because of the loop statement, a single process can get two signals if it is very fast.

Chapter 4

Memory management

Memory is an important module in the operating system because without it, it is impossible to run a program. In a multi-user or multi-programming environment, CPU needs to share memory and kernel must manage it offering the concept of *virtual memory* to the processes, by using external devices and differentiating between real and virtual memory (swapping partition).

A pair of **base** and **limit** registers define the logical address space and via hardware it is checked that the process remains in its space.

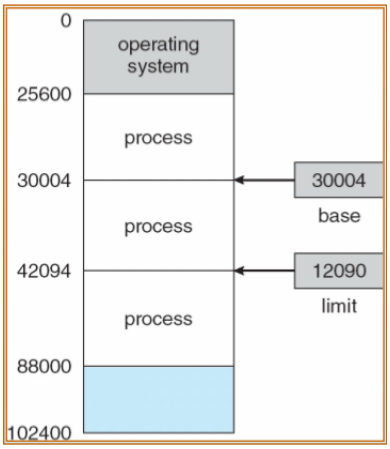

Figure 4.1: Base and limit registers

Address biding of instructions and data to memory addresses can happen at three different stages:

- **Compile time**: If memory location is known a priori, *absolute code* can be generated which must be recompiled if starting location changes;
- **Load time**: If memory location is not known at compile time, it must

generate *relocatable code*;

- **Execution time**: If the process can be moved during its execution from one memory to another, biding is delayed until run time and hardware support is needed for address maps.

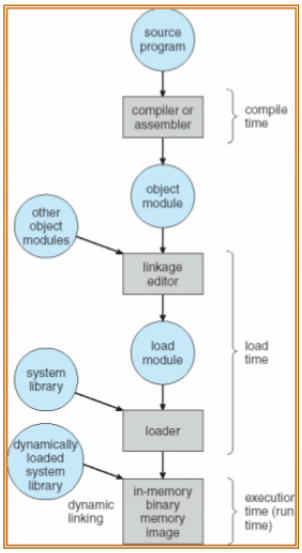

Figure 4.2: Multistep processing

Old CPUs generate a "static" program with pre-defined space and starting address, while modern CPUs generate logical address, starting from 0, mapped by the loader and the kernel into a physical one. In fact, loader interacts with the kernel, knowing which part of the memory are free and where to locate the program. Addresses are always logical contiguous but pages are located around the memory, therefore the design is more complicated but more flexible, too.

- **Logical addresses** or **virtual addresses** are generated by the CPU;
- **Physical addresses** are the addresses seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical and physical addresses differ in execution-time address-binding scheme.

Memory management unit

Memory-Management Unit is the hardware device responsible of mapping a virtual address into a physical one. The value in the *relocation register* is added to every address generated by a user process at the time it is sent to memory.

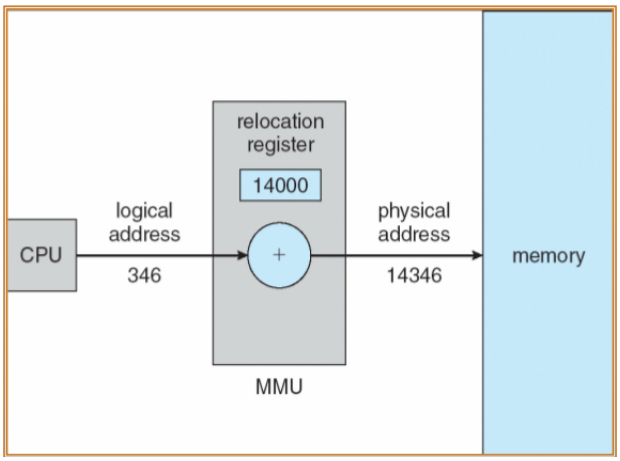

Figure 4.3: Relocation register. From logical to physical address

The user program deals with logical address and it never sees the real physical addresses.

Dynamic loading permits to load a routine on need. It is useful when large amounts of code are needed to handle infrequently occurring cases. In fact, an unused routine is never loaded, leading to a better memory-space utilization. No special support from the operating system is required, because tit is implemented through program design.

Dynamic linking permits to postpone linking until execution time. Small piece of code, *stub*, are used to locate the appropriate memory-resident library routine. Stub replaces itself with the address of the routine, and executes the routine. Operating system needs to check if routine is in processes' memory address. It is particularly useful for libraries.

Swapping A process can be temporarily swapped out of memory to a backing store and then brought back into memory for continued execution. *Backing store* is a fast disk large enough to accommodate copies of all memory images for all users which must provide direct access to these memory images. The system maintains a **ready queue** of ready-to-run processes which have memory images on disk. *Roll out, roll in* is a swapping variant used for priority-based scheduling algorithms: lower-priority process is swapped out so higher-priority process can be loaded and executed.

Major part of swap time is transfer time and the total transfer time is directly proportional to the amount of memory swapped.

4.1 Contiguous allocation

Usually, main memory is divided into two partitions:

- Resident operating system, usually held in low memory with interrupt vector;
- User processes held in high memory.

Relocation registers are used to protect user processes from each other, and from changing operating-system code and data. MMU dynamically maps logical addresses.

When a process arrives, it is allocated memory from a hole[1] large enough to accommodate it. Operating system maintains information about allocated partitions and free partitions (holes). Different algorithms are available to satisfy a request from a list of free holes.

- First-fit allocates the first hole that is big enough;
- Best-fit allocates the smallest hole that is big enough; it must scan the entire list, unless it is ordered by size and produces the smallest leftover hole;
- Worst-fit allocates the largest hole; it must scan the entire list too and produces the largest leftover hole.

First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

Fragmentation

- **External Fragmentation**: total memory space exists to satisfy a request, but it is not contiguous;
- **Internal Fragmentation**: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition but not being used.

External fragmentation can be reduced by *compaction* which is possible only if relocation is dynamic, it is done at execution time and implies to shuffle memory contents to place all free memory together in one large block.

4.2 Paging

Physical memory is divided into fixed-size blocks called **frames** and logical memory is divided into blocks of same size called **pages**. In this way, logical address space of a process can be *non-contiguous* and a process can be allocated physical memory whenever the latter is available.

Address generated by the CPU is divided into:

- **Page number**, used as an index into a *page table* which contains base address of each page in physical memory;

[1] Block of available memory. Holes of various size are scattered throughout memory

- **Page offset**, combined with base address to define the physical memory address that is sent to the memory unit.

Page table Page table is kept in main memory, identified by **Page-table base register (PTBR)** pointing to the page table and **Page-table lenght register (PRLR)** indicating size of the page table.

In this scheme, every access requires two memory accesses, the first one for the page table and the second one for the data/instruction. The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or **translation look-aside buffers (TLBs)**. An associative memory provides an address given a content, which the opposite behavior with respect to a canonical memory.

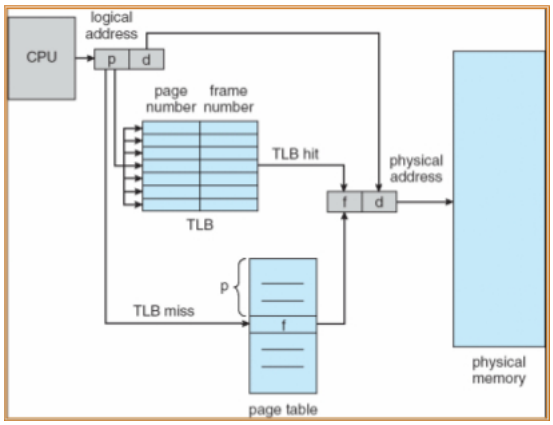

Figure 4.4: Address generation with TLB

Being ϵ the time for associative lookup, t the cycle time and α the hit ratio, the **Effective Access Time** is calculated as $EAT = (t + \epsilon)\alpha + (2t + \epsilon)(1 - \alpha)$

Memory protection Memory protection is implemented by associating a protection bit with each frame. A **valid-invalid** bit is attached to each entry in the page table:

- *Valid* indicates that the associated page is in the process' logical address space, and is thus a legal page;
- *Invalid* indicates that the page is not in the process' logical address space.

TLB is an hardware structure and therefore shared. Hence, it has to be rebuilt on context switch.

One copy of read-only code can be shared among processes (i.e. text editors, compilers, window systems). Shared code must appear in the same location in the logical address space of all processes.

On the other hand, each process keeps a separate copy of the code and data. The pages for the private code and data can appear anywhere in the logical address space. Page table is inherited by a child process, in fact parent and child processes execute the same code and share the initial status. A new frame will be allocated by the process which change some variable content.

4.2.1 Structure of the page table

Having n bits to store the page number, in kernel memory it is needed to have 2^n entries to store the page table.

Hierarchical Page Tables Hierarchical Page Tables break up the logical address space into multiple page tables. A simple technique is a two-level page table.

For example, a logical address on 32-bit machine with 1K page size is divided into:

- a page number consisting of 22 bits;
- a page offset consisting of 10 bits.

Since the page table is paged, the page number is further divided into

- a 12-bit page number;
- a 10-bit page offset.

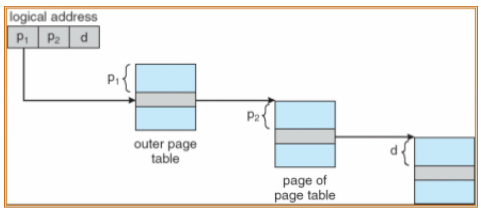

Figure 4.5: Two levels page table

Hashed Page Tables Hashed Page Tables are common in address space larger than 32 bits. The virtual page number is hashed into a page table containing a chain of elements hashing to the same location. Virtual page numbers are compared in this chain searching for a match and, if a match is found, the corresponding physical frame is extracted.

Inverted Page Tables Inverted Page Tables contain one entry for each real page memory. Each entry consists of the virtual address of the page stored in the real memory location, with information about the process which owns that page. This solution decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

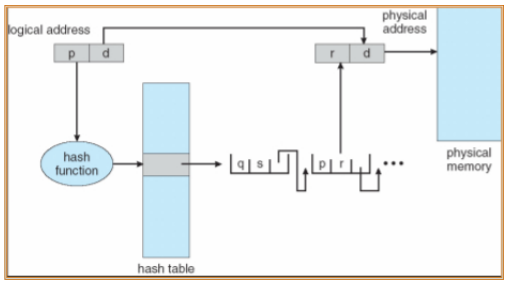

Figure 4.6: Hashed page table

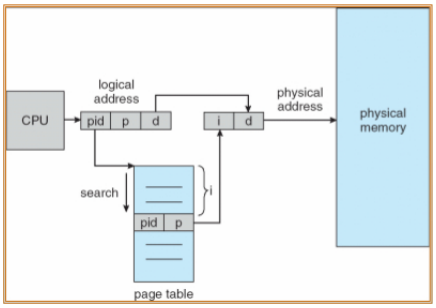

Figure 4.7: Inverted page table

4.3 Segmentation

Segmentation is a memory management scheme which supports user view of memory. In fact, a program is a collection of segments [2]. Logically, an address consists of a two tuple composed by segment number and offset.

Segment table is identified by a **Segment-table base register (STBR)** pointing to the segment table's location in memory and **Segment-table length register (STLR)** indicating the number of segments used by a program. Segment table maps two-dimensional physical addresses; each table entry has

- **Base**, containing the starting physical address where the segments reside in memory;
- **Limit**, specifying the length of the segment.

Protection is ensured associating a validation bit and read/write/execute privileges to each entry in segment table. Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

[2] A segment is a logical unit containing homogeneous data with a variable size

Memory mapping

Instruction `mmap` allocates some memory for a file. It is useful if the file has to be read in a non-sequential way. In this way, file is seen as a memory array and the programmer does not care about reading or moving on the file because it transparently done by the kernel. Library `sys/mman.h` contains `mmap` function:
`void *mmap(void *ADDR, size_t LEN, int PROT, int FLAGS,`
`int FILDES, off_t OFF)`

It maps `LEN` bytes starting at offset `OFF` from the file specified by the file descriptor `FILDES` into the caller's address space and it returns the actual place where the object is mapped. `ADDR` is a hint only where to allocate the file, and is usually specified as 0 (`NULL`).

`PROT` describes the desired memory protection as the bitwise or of `PROT_READ`, `PROT_WRITE` and `PROT_EXEC`.

`FLAGS` is the bitwise or of

- `MAP_SHARED`: any update made to the mapped region will be global, hence it will be seen by any other process (common option);
- `MAP_PRIVATE`: the updates will be kept private to each process, copy on write;
- Many other options on Linux.

Chapter 5

Virtual memory

Virtual memory provides a separation of user logical memory from physical memory. Only part of the program needs to be in memory for execution, therefore logical address space can be much larger than physical address space. Virtual memory allows address spaces to be shared by several processes and it allows for ore efficient process creation. It can be implemented via demand paging or demand segmentation.

5.1 Demand paging

Demand paging consists of bringing a page into memory only when it is needed, leading to less input/output operations, less memory needed and faster response. An invalid reference causes an abort, while a not-in-memory fault causes a page fault exception.

Demand paging works because of the locality model. In fact, process migrates from one locality to another. Those localities may overlap.

Whit each entry in the page table, a valid-invalid bit is associated. Initially this bit is set to invalid (i.e. not-in memory) on all entries. During address translation, if valid-invalid bit in page table entry is set to invalid a page fault occurs.

A swapper that deals with pages is a **pager**.

Page fault If there is a reference to a page, first reference to that page will trap the operating system: *page fault*.

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort;
   - Just not in memory;
2. Get empty frame;
3. Swap page into frame;

4. Reset tables;
5. Set validation bit;
6. Restart the instruction that caused the page fault.

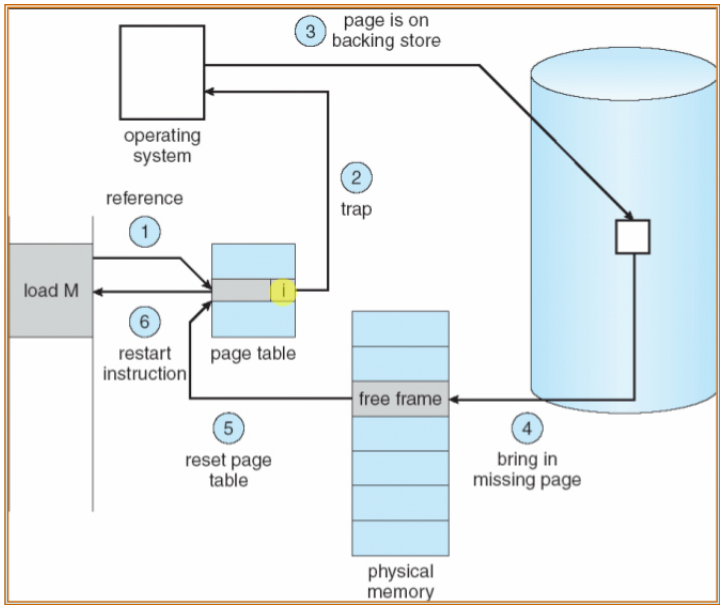

Figure 5.1: Page fault handling

$EAT = (1-p)t_{memory} + p(t_{page\ fault} + t_{swap\ out} + t_{swap\ in} + t_{restart})$ being $0 \leq p \leq 1$, page fault ratio.

5.1.1 Page replacement

If there is no free frame, it is needed to find some page in memory not really in use and swap it out.

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement;
- Use modify (dirty) bit to reduce overhead of page transfers: only modified pages are written to disk;
- Page replacement completes separation between logical and physical memory. Larger virtual memory can be provided on a smaller physical memory.

Basic algorithm

1. Find the location of the desired page on disk;
2. Find a free frame:
   - If there is a free frame, use it;
   - If there is no free frame, use a page replacement algorithm to select a victim frame;
3. Bring the desired page into the newly free frame and update page and frame tables;
4. Restart the process.

Optimal Algorithm Replace page that will not be used for longest period of time. Used for measuring every algorithm performance. It is unfeasible because it should predict the future.

FIFO Algorithm

Time	1	2	3	4	5	6	7	8	9	10	11	12
String	4	3	2	1	4	3	5	4	3	2	1	5
Fault	•	•	•	•	•	•	•			•	•	
Frame 1	4	3	2	1	4	3	5	5	5	2	1	1
Frame 2		4	3	2	1	4	3	3	3	5	2	2
Frame 3			**4**	**3**	**2**	**1**	4	4	**4**	**3**	5	5

$$f = 10/12 = 83\%$$

This algorithm is subjected to Belady's Anomaly: adding more frames will cause more page faults.

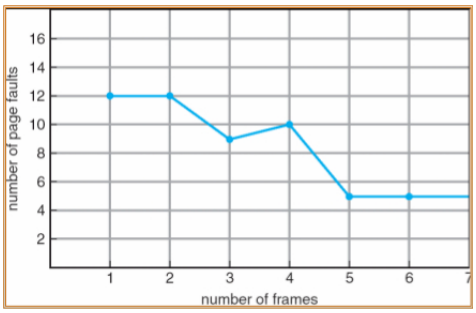

Figure 5.2: Belady's Anomaly

Least Recently Used Algorithm Every page entry has a counter where the clock; every time page is referenced through this entry, the clock is copied into the counter. When a page needs to be changed, look at the counters to determine which are to change.

Time	1	2	3	4	5	6	7	8	9	10	11	12
String	4	3	2	1	4	3	5	4	3	2	1	5
Fault	•	•	•	•			•			•	•	•
Frame 1	4	3	2	1	4	3	5	4	3	2	1	5
Frame 2		4	3	2	1	4	3	5	4	3	2	1
Frame 3			**4**	**3**	**2**	**1**	4	3	**5**	**4**	3	2

$$f = 10/12 = 83\%$$

A possible implementation is the **stack implementation** which keeps a stack of page numbers in a double link form:

- Page referenced
  - Move it to the top;
  - Requires six pointers to be changed;
- No search for replacement.

This implementation causes a large overhead and therefore it is unfeasible and it is not used in reality.

Least Recently Used Stack Adding a new page cannot increase the number of page faults. In fact, the old "stack" is the same and it can keep a new page.

Time	1	2	3	4	5	6	7	8	9	10	11	12
String	4	3	2	1	4	3	5	4	3	2	1	5
Fault	•	•	•	•	•	•	•			•	•	•
Frame 1	4	3	2	1	4	3	5	4	3	2	1	5
Frame 2		4	3	2	1	4	3	5	4	3	2	1
Frame 3			4	3	2	1	4	3	5	4	3	2
Frame 4				4	3	**2**	1	1	**1**	**5**	**4**	3

$$f = 8/12 = 75\%$$

LRU Approximation Algorithms

Reference bit Select a page which has not been referred for a certain amount of time, i.e. since the last page fault.

- With each page associate a bit, initially set to zero;
- When page is referenced bit is set to 1;
- Replace the page where bit is 0, if one exists.

Second chance

- Need reference bit;
- Clock replacement;
- If page to be replaced (in clock order) has reference bit set to 1, then:
  - Set reference bit to 0;
  - Leave page in memory;
  - Replace next page in clock order, subject to same rules.

5.1.2 Allocation of frames

Each process needs *minimum* number of pages. Two major allocation schemes:

Fixed allocation Allocation can be equal (i.e. assign an equal number of frames among the processes) or proportional (i.e. assign a number of frames according to the size of process).

Priority allocation Use a proportional allocation scheme using priority numbers rather than size. If a process generates a page fault, select for replacement one of its frame or select for replacement a frame from a process with lower priority number.

Two strategies are possible from page replacement:

- **Global replacement**: each process selects a replacement frame from the set of all frames; one process can take a frame from another;
- **Local replacement**: each process selects from only its own set of allocated frames.

If a process does not have "enough" pages, the page-fault rate is very high, leading to:

- Low CPU utilization;
- Operating system thinks that it needs to increase the degree of multiprogramming;
- Adding another process.

Thrashing means that a process is busy swapping pages in and out.

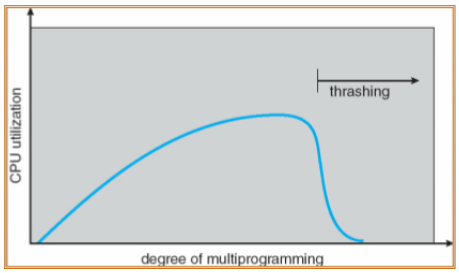

Figure 5.3: Thrashing

5.1.3 Working-Set Model

Defining Δ as the working-set windows (i.e. a fixed number of page references), it is possible to define WSS_i as the working set of process P_i (i.e. hence the total number of pages referenced in the most recent Δ).

- If Δ is too small, it will not encompass entire locality;
- If Δ is too large, it will encompass several localities;
- If $\Delta = \infty$, it will encompass entire program.

The working-set strategy has a variable *resident set*, $\overline{RT} = \frac{1}{T}\sum_{t=1}^{T} RS(t, \Delta)$.

5.1.4 Page-Fault Frequency Scheme

Establish "acceptable" page-fault rate ($1/c$ where c is the time between two page faults).

If actual rate is too low, the process loses frame. On the other hand, if actual rate is too high, process gains frame.

Strategy Run at each page fault, not at every reference. Is controlled by the time interval between page faults.

- If $\tau < c$, i.e. the measured page fault frequency is greater than the one acceptable, a page has to e added to the Resident Set of the process issuing this page fault;
- If $\tau > c$, all pages not referred in that interval by the current process are eliminated from its Resident Set. The reference bit of all pages in the Resident Set are cleared.

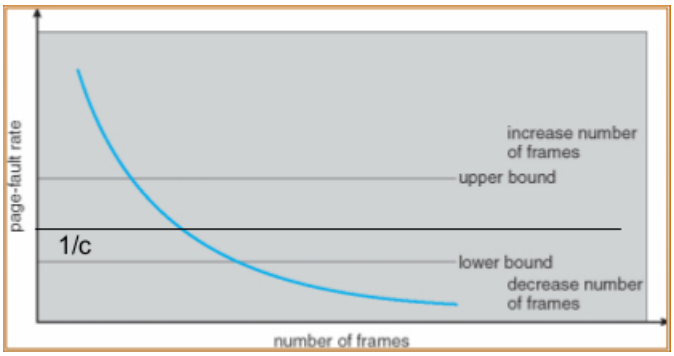

Figure 5.4: Page-Fault frequency scheme

5.1.5 Other issues

Prepaging To reduce the large number of page faults that occurs at process startup, prepage all or some of the pages a process will need, before they are referenced but if prepaged pages are unused, I/O and memory was wasted.

Assuming s pages are prepaged and α of the pages is used, it is needed to compare the cost of $s \cdot \alpha$ save pages faults with the cost of prepaging $s \cdot (1 - \alpha)$ unnecessary pages.

Page size Page size selection must take into consideration fragmentation, table size, I/O overhead and locality.

Chapter 6

Device management

6.1 Booting a PC to run a kernel

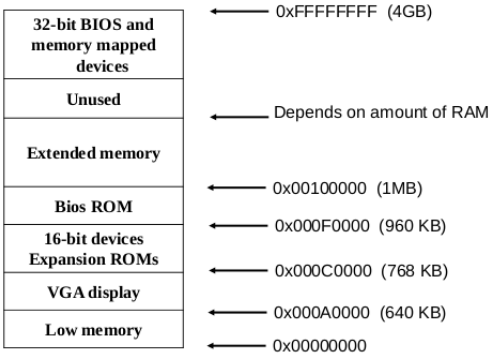

Figure 6.1: UNIX Memory Organization

The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an eary PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF.

- The 640KB area marked "Low Memory" was the *only* random-access memory (RAM) that an early PC could use;
- The 348KB area from 0x000A0000 through 0x000FFFFF was reserved by the hardware for special uses such as video display buffers and firmware held in nonvolatile memory;
- The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from 0x000F0000 through 0x000FFFFF.

Intel 80286 and 80836 processors supported 16MB and 4GB physical address spaces respectively, but the PC architects preserved the original layout for the low 1MB of physical address space for backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or *conventional memory* (the first 640KB) and *extended memory* (everything else). More BIOS is located at the high end of the 32-bit address range for use by 32-bit PCI devices.

6.1.1 The BIOS

The BIOS is responsible for performing *Power-On-Self-Test* and basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

486 and later processors start executing at physical address 0xFFFFFFF0, which is at the very top of memory space, in an area reserved for the ROM BIOS. The first instruction is `jmp far F000:E05B` which jumps to the normal BIOS, located in the 64KB region from 0xF0000 to 0xFFFFF mentioned above. The CPU starts in *real mode*, so the `jmp far` instruction is a real mode jump that restores us to low memory. In real mode, the segmented address `segment:address` translates to the physical address `segment*16 + offset`. Thus, F000:E05B translates to 0x000FE05B.

In real mode, nether Global Descriptor Table (GDT) nor Local Descriptor Table (LDT) (which, in protected mode, contain global and local segment descriptions respectively) are needed by the CPU. The code that initializes these data structures must run in real mode. When the BIOS run, it initializes the PCI bus and all the important devices it knows about (in particular VGA display), then it searches for a bootable device such as a floppy disk, hard drive or CD-ROM and when it find it, it reads the boot loader from the disk and transfers control to it.

6.1.2 The Boot Loader

The *Boot Loader* is the program run by the BIOS to load the image of a kernel into RAM. Floppy and hard disks are by historical convention divided up into 512 byte regions called *sectors*. If the disk is bootable, the first sector is called the *boot sector*, since this is where the Boot Loader code resides. When the BIOS find a bootable floppy or hard disk, it loads the 512-byte boot sector into lowe memory, at physical address 0x7C00 through 0x7DFF and then it uses a `jmp` instruction to set the `CS:IP` to 0000:7C00, passing control to the Boot Loader.

Boot Loader important files are:

- `boot.s` - First, the boot loader switches the processor from real mode to 32-bit protected mode, because it is only in this mode that software

can access all the memory above 1MB in the processor's physical address space;

- `main.c` - Second, the boot loader reads the kernel from the hard disk by directly accessing the IDE disk device registers via the x86's special I/O instructions;
- `boot.asm` - This file is a disassembly of the boot loader. It easy to see exactly where in the physical memory all of the boot loader's code resides.

The boot loader's link and load addresses match perfectly. There is a rather large disparity between the kernel's link and load addresses. Operating system kernels often like to be linked and run at very high virtual address, such as 0xF0100000, in order to leave the lower part of the processor's virtual address space for user programs to use. Actually, it is not possible to load the kernel at *physical address* 0xF0100000, then it is needed to use the processor's memory management hardware to map virtual address 0xF0100000 - link address where the kernel code *expects* to run - to physical address 0x00100000 - where the boot loader *actually* loaded the kernel. The kernel's virtual memory address is high enough to leave plenty of address space for user processes, it will be loaded in physical memory at the 1MB point in the PC's RAM, just above the BIOS ROM.

6.2 I/O subsystem

User application access to a wide variety of different devices is accomplished through layering and through encapsulating all the device-specific code into device drivers, while application layers are presented with a common interface for all (or at least large general categories of) devices. Most devices can be characterized as either block I/O, character I/O, memory mapped file access or network sockets. A few devices are special, such as time-of-day clock and the system timer. Most operating systems also have an escape, or back door, which allows applications to sens command directly to device drivers if needed. In UNIX this is `ioctl()` system call. `ioctl()` takes three arguments: the file descriptor for the device driver being accessed, an integer indicating the desired function to be performed and an address used for communicating or transferring additional information.

6.2.1 Block and Character Devices

Block devices are accessed one block at a time, and are indicated by a `b` as the first character in a long listing on UNIX systems. Operations supported include `read`, `write` and `seek`. Accessing blocks on a hard drive directly, without going through the file system structure, is called *raw I/O* and can speed up certain operation by bypassing the buffering and locking normally introduced by the OS, i.e. it becomes the application's responsibility to manage those issues. A

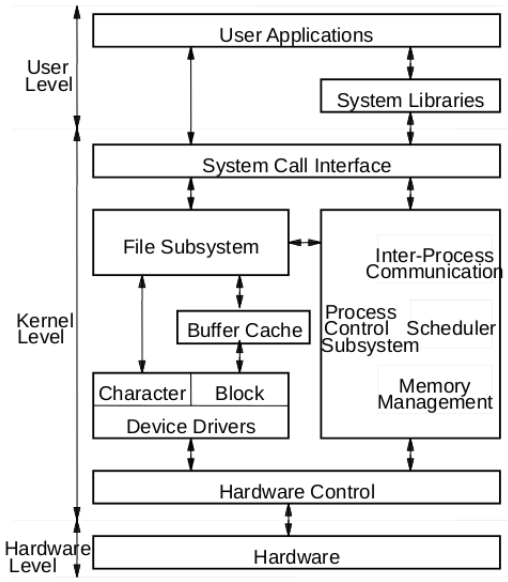

Figure 6.2: UNIX I/O subsystem

new alternative is *direct I/O*, which uses the normal file system access, but which disables buffering and locking operations. Memory mapped file I/O can be layered on top of block-device drivers. Rather than reading the entire file, it is mapped to a range of memory addresses and then paged into memory as needed using the virtual memory mechanism. Access to the file is then accomplished through normal memory access, i.e. pointers, rather than through `read` and `write` system calls. This approach is commonly used for executable program code.

Character devices are accessed one byte at a time, and are indicated by a `c` in UNIX systems. Supported operations include `get` and `put`, with more advanced functionality such as reading an entire line supported by higher-level library routines.

6.2.2 File system data structures

In the original UNIX file system, the physical disks is divided into logical disks called *partitions*. Each partition is a standalone file system. Each device in the system is characterized by its own major device number and each partition has an associated minor device number which the device drivers uses to access the raw file system. The major/minor device number combination is used as an handle into device switch table. In particular, the major number acts as an index, and the minor number is passed as an argument to the driver routines so that they can recognize the specific instance of a device.

Each file system contains:

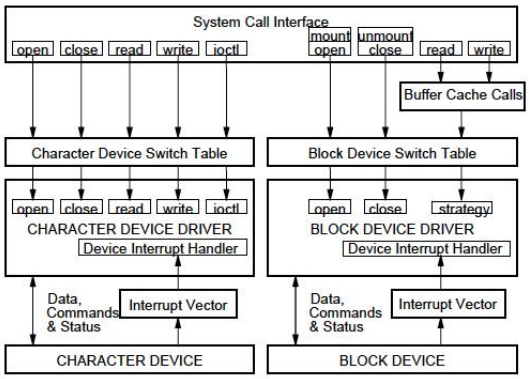

Figure 6.3: UNIX software interface

- a *boot block* located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system. Typically, the first sector contains a bootstrap program that reads in a larger bootstrap program from the next few sector, and so forth;
- a *superblock* describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the i-node number of the root directory, magic number, etc.;
- a linear *array of i-nodes*. There is a one to one mapping of files to i-nodes and viceversa. An i-node is identified by its *i-node number*, which contains the information needed to find the i-node itself on the disk. Thus, while users think of files in terms of file names, UNIX thinks of files in terms of i-nodes.
- a *data blocks* containing the actual content of files.

System calls like `mkfs` can be used to create (format) the file system (partition). Indeed, it creates the superblock, the i-node list, the list of free blocks and the root node for the new file system. Other system calls like `fsck`, instead, are used to check and repair the file system.

Therefore, it is simple to understand that there is a similarity between normal files and device in a UNIX file system: both are characterized by an i-node.

Open system call for I/O devices

1. Convert pathname into i-node;
2. Increase the i-node reference counter;
3. Allocate an element into user file table and system file table;
4. Get major and minor number from i-node;

Block

Major number	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

Character

Major number	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

Figure 6.4: UNIX switch tables

5. Save content using `setjmp`[1] to resolve a possible `longjmp` executed by the driver;
6. If device is a block device:
   (a) Use major number as an index into block device switching table;
   (b) Call open procedure of the driver associated to the index above passing minor number and mode;
7. Otherwise:
   (a) Use major number as an index into character device switching table;
   (b) Call open procedure of the driver associated to the index above passing minor number and mode;
8. If open fails, decrease counters in file and i-node tables.

6.2.3 Terminal driver

A multiplicity of I/O devices are regarded as "terminals" in Unix systems. A terminal is a character-oriented device, comprising streams of characters received from and sent to the device. Although characters streams are structured, incorporating control characters, escape codes and special characters, the I/O protocol is not structured as would be the I/O protocol of smart terminals. Terminals provide job control facilities: interactively, the user and the terminal can send control characters that suspend the current running job, reverting to the interactive job control shell that spawned the job and can run commands which place jobs in background or which switch a background job into foreground, without suspending it if necessary.

[1] Saves the current context (environment) in the stack. Used to backtrack in case of errors, instead of a chain of `return` statements

In UNIX, a terminal device comprises the underlying *device driver*, responsible for the physical control of the device hardaware via the I/O instructions and handling device interrupt requests, and the *line discipline*, as shown in figure 6.5. A line discipline is independent from the actual device hardware, and the same line discipline can be used for a terminal concentrator device responsible for multiple controlling terminals as for a pseudoterminal. Line disciplines can be setup by functions or system calls. They are the same across all terminal devices and independent from the actual hardware, dealing as they do in the simple abstractions provided by device drivers: transmit and receive a character and set different hardware states. Each terminal device can be switched among multiple line disciplines.

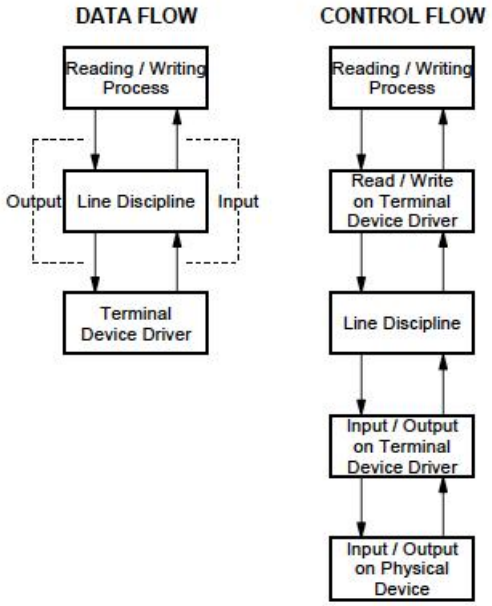

Figure 6.5: UNIX line discipline

Every line discipline may operate in two modes:

- **Raw mode**: the line discipline transfers data between a terminal and a process without any conversion. The `enter` character has no specific meaning in this mode. The behavior of the raw mode can be set using the `ioctl` system call;
- **Canonical mode**: characters are got into kernel buffer and passed to the process only when `ENTER` is pressed. The user input is converted for the receiving process and the process output is converted for the user. It manages `backspace`, formatting characters such as `tab` and `enter` and special characters such as `Ctrl-C`, `Del` and `Ctrl-D`.

The POSIX standard replaces the `ioctl` system call entirely, with a set of library functions with standardized names and parameters. The POSIX `termio` data structure is the data structure used by all of the terminal library calls.

```
struct termios {
  tcflag_t c_iflag;  // Input mdoes
  tcflag_t c_oflag;  // Output modes
  tcflag_t c_cflag;  // Control modes
  tcflag_t c_lflag;  // Local modes
  cc_t c_cc[NCCS];   // Control characters
};
```

Each process in the system has either a single controlling terminal, or no controlling terminal at all. A process inherits its controlling terminal from its parent, and the only operations upon a process are acquiring a controlling terminal, by a process that has no controlling terminal, and relinquishing it, by a process that has a controlling terminal. The standard defines the `O_NOCTTY` flag for the `open` system call. A process with no controlling terminal opens a terminal device file that is not already the controlling terminal for some other process, without specifying the `O_NOCTTY` flag.

`getty`

`getty` (get teletype) is a typical UNIX system program used to manage physical and virtual terminals. When it detects a connection, it prompts for a username password an runs the "login" program to authenticate the user.

At the login phase, the `init` process reads `/etc/inittab` and executes a `getty` process for each known terminal that has been switched on. Then, `getty` calls the system call `open` related to the terminal, which makes the process wait for the hardware connection has been established. The `getty` program, through system call *exec*, becomes the `login` process, which runs, exploiting system call `exec`, the user selected shell reading the last field of the username record in file `/etc/passwd`.

6.2.4 Buffer cache

In order to speedup the read request from a device, it is possible to use buffers and/or caches which allow a faster communication to take place

When the kernel is asked to serve a **read** request from disk, it tries to read the data from the *buffer cache*. If the data are in the cache, the data are returned to the user process without any disk access, otherwise the kernel reads a block of data from disk and stores it in the buffer cache, trying to keep it in memory as long as possible, for a possible future usage.

When the kernel is asked to serve **write** request to disk, it puts the data in the buffer cache. In this way, data stay in memory for possible read operations and any change of the data is performed in memory rather than on disk.

The buffer cache management tries to minimize the number of disk accesses by *reading ahead* the data that have high probability[2] to be referenced in the near future and by delaying as much as possible the transfer of the content of the buffer cache to disk (*delayed write*).

Read system call

```
procedure read
input:  user file descriptor
        user buffer address
        number of bytes to read
output: number of bytes copied in the user buffer
{
  while(! complete number of bytes to read) {
    convert offset to byte in a block (using bmap);
    compute the offset within a block
      and the number of byte to read;
    if(number of byte to read == 0)
      break;  /* tries to read beyond the end of file */
    read a block (using bread or breada);
    copy data from kernel buffer to user space;
    release buffer (using brelse);

    return number of bytes;
}
```

Block read: `bread`

```
procedure bread
input:  file system block number (blockno)
output: buffer containing requested block
{
  buffer = getblock(block blockno);
  if(data in buffer are valid)
    return buffer;
  start disk I/O read for block blockno into buffer;
  sleep(event: disk I/O read completed);
  return buffer;
}
```

Acquiring a block: `getblk`

```
procedure getblk
input:  device number
```

[2] According to the space and temporal locality principle

```
output: locked buffer to be used for storing the block
{
  while(buffer not found) {
    if(block is present in its hash queue) {
      /* Case 5 */
      if(buffer is locked) {
        sleep(event: buffer unlocked);
        continue;
      }
      /* Case 1 */
      mark buffer 'locked';
      remove buffer from free list;
      return buffer;
    } else {
      if(free list is empty) {
        /* Case 4 */
        sleep(event: freed buffer);
        continue;
      }
      remove buffer from free list;
      if(buffer marked 'delayed write') {
        /* Case 3 */
        asynchronous disk write of buffer;
        continue;
      }
      /* Case 2 */
      remove buffer from old hash queue;
      put buffer into new hash queue;
      return buffer;
    }
  }
}
```

Block release: `brelse`

```
procedure brelse
input:  locked buffer
output: none
{
  wake up all processes waiting for any free buffer;
  wake up all processes waiting for this buffer;
  disable interrupts;
  if(buffer contents are valid and buffer not old) {
    put buffer in free list queue;
  else
    put buffer in free list head;
```

```
  enable interrupts;
  mark buffer unlocked;
}
```

Block write: `bwrite`

```
procedure bwrite
input:  buffer to write
output: none
{
  start disk I/O write of buffer;
  if(synchronous I/O) {
    sleep(event: disk I/O write completed);
    brelse(buffer);
  } else if(buffer is marked delayed write)
    mark buffer 'old';
}
```

Block read ahead: `breada`

```
procedure breada
input:  file system block number for immediate read (blkno1)
        file system block number for asynchronous read (blockno2)
output: buffer containing data from immediate read (buffer1)
{
  if(buffer with blkno1 not present in buffer cache) {
    buffer1 = getblk(block blkno1);
    if(data in buffer1 are not valid)
      start disk I/O read for blockno1 into buffer1;
  }
  if(buffer with blkno2 not present in buffer cache) {
    buffer2 = getblk(block blkno2);
    if(data in buffer2 are valid)
      brelse(buffer2);
    else
      start disk I/O read for block blkno2 into buffer2;
  }
  if(buffer with blkno1 present in buffer cache) {
    buffer1 = bread(blkno1);
    return buffer1;
  }
  sleep(event: blkno1 disk I/O read completed);
  return buffer1;
}
```