Chapter II
Processes

Jehan-François Pâris
jfparis@uh.edu
Chapter Overview

- Processes
- States of a process
- Operations on processes
  - fork(), exec(), kill(), signal()
- Cooperating Processes
  - pipes
- Threads and lightweight processes
A process is a **program** executing a given **sequential computation**.

- An **active entity** unlike a program
- Think of the difference between a recipe in a cookbook and the activity of a cook preparing a dish according to the recipe!
Processes and programs (I)

- Can have one program and many processes
  - When several users execute the same program (text editor, compiler, and so forth) at the same time, each execution of the program constitutes a separate process.
  - A program that forks another sequential computation gives birth to a new process.
Examples

• Several executions of same program

• A program forking a child
Processes and programs (II)

- Can have one process and two—or more—programs
  - A process that performs an exec() call replaces the program it was executing
Examples

- One process executing two programs
The UNIX shell

• Program that
  – Reads input from the keyboard
  – Creates the process that will execute the command.
  – Wait for the completion of the process it has created unless it was specified otherwise

• User-level program that you and I could write
A very basic UNIX shell

- for (;;) {
  parse_input_line(arg_vector);
  if built_in_command(arg_vector[0]) {
    do_it(arg_vector);
    continue;
  } // built-in command
  pathname = find_path(arg_vector[0]);
  create_process(pathname, arg_vector);
  if (interactive()) wait_for_this_child();
- } // for loop
• All functions in italics are templates yet to be written

• Real shells do more:
  – I/O redirection
  – Pipes (as in `ls -alg | more`)
  – Command aliasing,
  – Wildcard characters (as “*”)
  – …
Importance of processes

• Processes are the **basic entities** managed by the operating system
  – OS provides to each process the illusion it has the whole machine for itself
  – Each process has a dedicated **address space**
The process address space

- Set of main memory locations allocated to the process
  - Other processes cannot access them
  - Process cannot access address spaces of other processes
- A process address space is the **playpen** or the **sandbox** of its owner
A last word

- There are many quasi-synonyms for process:
  - Job (very old programmers still use it)
  - Task
  - Program (strongly deprecated)
PROCESS STATES

- Processes go repeatedly through several stages during their execution
  - Waiting to get into main memory
  - Waiting for the CPU
  - Running
  - Waiting for the completion of a system call
The big diagram

- **New** → **Ready**
  - Admit process

- **Ready** → **Running**
  - Get CPU

- **Running** → **Blocked**
  - Completion

- **Blocked** → **System request**

- **Running** → **Exit**
  - **Terminated**

- **Running** → **Interrupt**

This is fundamental material
Process arrival

- New process
  - Starts in NEW state
  - Gets allocated a Process Control Block (PCB) and main memory
  - Is put in the READY state waiting for CPU time
The ready state

- AKA the *ready queue*
- Contains all processes waiting for the CPU
- Organized as a *priority queue*
- Processes leave the priority queue when they get some CPU time
  - Move then to the RUNNING state
The running state (I)

- A process in the running state has exclusive use of the CPU until
  - It *terminates* and goes to the *TERMINATED* state
  - It does a *system call* and goes to the *BLOCKED* state
  - It is *interrupted* and returns to the *READY* state
The running state (II)

- Processes are forced to relinquish the CPU and return to the READY state when
  - A higher-priority process arrives in the ready queue and preempts the running process
  - Get out, I’m more urgent than you!
  - A timer interrupt indicates that the process has exceeded its time slice of CPU time
The blocked state (I)

- Contains all processes waiting for the completion of a system request:
  - I/O operation
  - Any other system call

- Process is said to be
  - \textit{blocked} (Arpaci-Dusseau & Arpaci-Dusseau)
  - \textit{waiting}
  - \textit{sleeping} (UNIX)
The blocked state (II)

- A system call that does not require callers to wait until its completion is said to be *non-blocking*
  - Calling processes are immediately returned to the *READY* state

- The blocked state is organized as a *set of queues*
  - One queue per device, OS resource
The process control block (I)

- Contains all the information associated with a specific process:
  - *Process identification* (pid), *argument vector*, ...
    - UNIX pids are unique integers
  - *Process state* (new, ready, running, ...),
  - *CPU scheduling information*
    - Process priority, processors on which the process can run, ...,
The process control block (II)

- *Program counter* and other CPU registers including the *Program Status Word* (PSW),
- *Memory management information*
  - Very system specific,
- *Accounting information*
  - CPU time used, system time used, ...
- *I/O status information* (list of opened files, allocated devices, ...)
The process table

- System-wide table containing
  - *Process identification* (pid), *argument vector*, ...
  - *Process current state*
  - *Process priority and other CPU scheduling information*
  - A *pointer* to the remaining information.
Swapping

- Whenever the system is very loaded, we might want to expel from main memory or **swap out**
  - Low priority processes
  - Processes that have been waiting for a long time for an external event
- These processes are said to be **swapped out** or **suspended**.
How it works

New → Ready

Admit process

Get CPU

Running → Terminated

Exit

System request

Blocked

Completion

Interrupt

Ready

Suspended

Activate

Deactivate

Suspended

Blocked

Completion

Suspended

Ready

Deactivate
Suspended processes

- Suspended processes
  - Do not reside in main memory
  - Continue to be included in the process table

- Can distinguish between two types of suspended processes:
  - Waiting for the completion of some request *(blocked_suspended)*
  - Ready to run *(ready_suspended)*
A system should *not* swap out ready processes unless their priority is very low.

Otherwise swapping out ready processes is a desperate measure.
OPERATIONS ON PROCESSES

• Process creation
  - fork()
  - exec()
    - The argument vector

• Process deletion
  - kill()
  - signal()
Process creation

- Two basic system calls
  - `fork()` creates a carbon-copy of calling process sharing its opened files
  - `execv()` overwrites the contents of the process address space with the contents of an executable file
The first process of a system is created when the system is booted.

All other processes are forked by another process (parent process).

- They are said to be children of the process that created them.
fork() (II)

- When a process forks, OS creates an *identical copy* of forking process with
  - a *new address space*
  - a *new PCB*

- The *only* resources shared by the parent and the child process are the *opened files*
fork() (III)

Parent:
fork()
returns PID of child

Child:
fork()
returns 0

opened files
First example

- #include <iostream>
  using namespace std;
  main()
  {
    fork();
    cout << "Hello" << endl;
  } // main
  
  will print two lines as cout will be executed by the parent and the child
How it works

... fork();
cout ...;
...

cout ...;
...
Second example

```cpp
main() {
    fork();
    fork();
    fork();
    cout << "Hello" << endl;
} // main
```

will print four lines as `cout` will be executed by the parent, its two children and its grandchild.
How it works
```c
int pid;
pid = fork();
if (pid == 0) {
    // child process
    ...
}
else {
    // parent process
    ...
}
```
int pid;
pid = fork();
if (pid == 0) {
  // child process
  ...
  _exit(0); // normal exit
}
// if
// parent process continues
...
int pid;
if ((pid = fork())== 0) {
    // child process
    ...
    _exit(0); // normal exit
} // if
// parent process continues
    ...
Waiting for child completion

- **wait(0)**
  - Waits for the completion of any child
  - No wait if any child has already completed

- **while (wait(0) != kidpid)**
  - Waits for the completion of a specific child identified by its pid
An example (I)

- `#include <iostream>
  #include <sys/types.h>
  #include <sys/wait.h>
  using namespace std;`
main() {
    int pid;
    if((pid = fork()) == 0) {
        cout << "Hello!" << endl;
        _exit(0);
    } // child
    wait(0);
    cout << "Goodbye!" << endl;
} // main
Notes

• UNIX keeps in its process table all processes that have terminated but their parents have not yet waited for their termination
  – They are called *zombie processes*

• The statement
  ```
  while (kidpid != wait(0));
  ```
  is a loop with an *empty body*
int kidpid;
if ((kidpid = fork())== 0) {
    // child process
    ...
    _exit(0); // normal exit
} // if
// parent waits for child
while (wait(0) != kidpid)
    ...
exec

- Whole set of exec() system calls
- Most interesting are
  - `execv(pathname, argv)`
  - `execve(pathname, argv, envp)`
  - `execvp(filename, argv)`
- All exec() calls perform the same basic tasks
  - Erase current address space of process
  - Load specified executable
• **execv(pathname, argv)**
  - char pathname[]
    - *full pathname* of file to be loaded: `/bin/ls` instead of *ls*
  - char argv[][][]
    - the *argument vector*:
      passed to the program to be loaded
Argument vector (I)

- an array of pointers to the individual argument strings
  - `arg_vector[0]` contains the name of the program as it appears in the command line
  - other entries are parameters
  - end of the array is indicated by a NULL pointer
Argument vector (II)

- `char argv[][][];`
- `char **argv;`
execve and execvp

- **execve(pathname, argv, envp)**
  - Third argument points to a list of environment variables

- **execvp(argv[0], argv)**
  - Lets user specify a filename instead of a full pathname
  - Looks for `argv[0]` in list of directories specified in environment variable `PATH`
Putting everything together

```c
int pid
if ((pid = fork()) == 0) {
    // child process
    ...
    execvp(filename, argv);
    _exit(1); // exec failed
} // if
while (pid != wait(0));
// parent waits
...```
Observations (I)

- Mechanism is quite costly
  - `fork()` makes a *complete copy* of parent address space
    - *very costly* in a virtual memory system
      - `exec()` thrashes that address space
    - Berkeley UNIX introduced cheaper `vfork()`
      - Shares the parent address space until the child does an `exec()`
Copy-on-write

Parent and child share same address space

When either of them modifies a page, other gets its own copy of original page

COW of original page
Observations (II)

- Neither `fork()` nor `exec()` affect opened file descriptors
  - They remain unchanged
- Important for UNIX I/O redirection mechanism
• Fork was not that expensive on a minicomputer with a 16-bit address space
  – Never had to copy more than 64KB

• Using a fork/exec allowed a very easy implementation of I/O redirection
  – After the `fork()` thus in the child
  – Before the `exec()` while parent is still in control
for (;;) {
    parse_input_line(argv);
    if built_in(argv[0]) {
        do_it(arg_vector);
        continue;
    }
    // built_in command
    path = find_path(argv[0]);
A very basic shell (II)

```c
if ((pid = fork()) == 0) {
    // put here I/O
    // redirection code
    execv(path, argv);
    _exit(1); // failure
} // child process

if (interactive())
    while (wait(0) != pid);
} // main for loop
```
Comments

- Shell built-in commands include
  - `exit` terminates the shell
  - `cd` changes current directory
- Commands are assumed to be interactive
  - *Non-interactive* commands end with an “&”
Terminating a process (I)

- Sending a signal:
  - `kill()` has two arguments
    - the `process id` of the receiving process
    - a `signal name` or a `signal number`

- `#include <signal.h> // must` `kill(this_pid, this_signal);`

- Process receiving the signal will `terminate`
Terminating a process (II)

Process P

Process Q

kill(Q_pid, SIGINT);

What should I do? AARGH!
Catching a signal (I)

- The process receiving signal can *catch* it by using `signal()`
  - Will not terminate:

- `signal(a_signal, catch_it);`
  - where `catch_it` is a pointer to a function that will be called whenever signal `a_signal` signal is received.

- The ninth signal, `SIGKIL`, cannot be caught.
Catching a signal (II)

Process Q

kill(Q_pid, SIGINT);

Process is now shielded by signal() call

Process P
When processes are not enough

- Single threaded server:
  - Processes one request at a time

for (;;) {
    receive(&client, request);
    process_request(...);
    send (client, reply);
} // for
A tricky question

- What does a server do when it does not process client requests?
Three good answers

- Nothing
- It waits for client requests
- It sleeps
  - **Blocked state** is sometimes called *the sleep state*
The problem

• Most client requests involve disk accesses
  – File servers
  – Authentications servers

• When this happens, the server remains in the BLOCKED state
  – Cannot handle other customers’ requests
An analogy

• In most fast-food restaurants, counter employees process customer orders one order at a time.

• Not be possible in a traditional restaurant
  – A waitperson that would only be able to wait on one table at a time would be idle most of the time.
A first solution

```c
int pid;
for (;;) {
    receive(&client, request);
    if ((pid = fork())== 0) {
        process_request(...);
        send (client, reply);
        _exit(0); // done
    } // if
} // for
```
The good and the bad news

• The good news:
  – Server can now handle several user requests in parallel

• The bad news:
  – `fork()` is a very expensive system call
A better solution

- Provide a faster mechanism for creating cheaper processes:
  - Lightweight processes
  - Threads
How?

- Lightweight processes and threads *share the address space of their parent*
  - No need to create a new address space
- Most expensive step of `fork()` system call
Is it not dangerous?

- To some extent because
  - No memory protection inside an address space
  - Lightweight processes can now interfere with each other
- But
  - All lightweight process code is written by the same team
General Concept (I)

- A thread or lightweight process
  - Does not have its own address space
  - Shares it with its parent and other peer threads in the same address space (task)

- Each thread has a program counter, a set of registers and its own stack.
  - Everything else is shared
• A regular process (single-threaded)
• A process containing several threads
Implementation

- Thread and LWPs can either be
  - **Kernel supported:**
    - Mach, Linux, Windows NT and after
  - **User-level:**
    - Pthread library, ...
Kernel-Supported Threads (I)

- Managed by the kernel through system calls
- One process table entry per thread
- This is the best solution for *multiprocessor architectures*
  - Kernel can allocate *several processors* to a *single multithreaded task*
Kernel-Supported Threads (II)

• Supported by Mach, Linux, Windows NT and more recent systems

• Performance Issue: Switching between two threads in the same task involves a system call
  – Results in two context switches

Here they come again!
Linux Threads (I)

• Created through:

  \texttt{clone (fn, stack, flags)}

where

- \texttt{fn} specifies function to be executed by the new thread or process
- \texttt{stack} points to the stack it will use
- \texttt{flags} is a set of flags specifying various options (\texttt{CLONE_VM} for threads)
If the `CLONE_VM` flag is missing, Linux creates a regular process with its own address space.
User-Level Threads (I)

- User-level threads are managed by procedures within the task address space
  - The *thread library*
- One process table entry per task/address space
  - Kernel is not even aware that process is multithreaded
User-Level Threads (II)

- Can be retrofitted into an OS lacking thread support
  - Portable thread libraries
- *No performance penalty:* Switching between two threads of the same task is done cheaply within the task
  - Same cost as a procedure call
User-Level Threads (III)

- **Programming issue:**
  - Each time a thread does a *blocking system call*, kernel will move the whole process to the *blocked state*.
  - It does not know better.
  - Programmer must use *non-blocking system calls*.
  - *Can be nasty.*
Kernel
Process wants to sleep for 5 seconds:
Let us move it to the blocked state

User-Level Threads (IV)
sleep(5);

Let us move it to the blocked state
POSIX Threads

- POSIX threads, or *pthreads*, started as pure user-level threads managed by the POSIX thread library
  - Gained later *some kernel support*
- Ported to various Unix and Windows systems (*Pthreads-win32*).
- Function names start with *pthread_*
- Calls tend to have a complex syntax
An Example (I)

#include <pthread.h>
static int i;
static int count[2];

Static variables are shared by all threads
Other variables are stored on the private stack of each thread.
void *child(void *arg) {
    int index;
    index = (int) arg;
    for(;;) {
        printf("Child count: %d\n",
            ++count[index]);
        sleep(1); // one sec. delay
    } // for loop
} // child
int main() {
    thread_t tid; // thread id
    int i = 0;
    pthread_create(&tid, NULL, child, (void *) i);
    // pthread will execute
    // "child" function

    NULL stack address specifies
    a new stack "anywhere"
i++; // now i == 1
while (count[i] < 12) {
    printf("Parent count: %d\n", ++count[i]);
    sleep(1); // one sec. delay
} // while loop
return 0;
} // main
<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel threads</th>
<th>User-level threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiprocessing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease of use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Comparing the approaches

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel threads</th>
<th>User-level threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Multiprocessing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease of use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Comparing the approaches

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel threads</th>
<th>User-level threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td>Multiprocessing</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease of use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Comparing the approaches

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel threads</th>
<th>User-level threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Multiprocessing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Ease of use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing the approaches

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel threads</th>
<th>User-level threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Multiprocessing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Ease of use</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• No clear winner between kernel-supported and user-level threads

• Solaris (from Sun, now taken over by Oracle)
  – Supports both user-level threads and kernel threads
  – Lets programmers combine them as they need