Managing Concurrency with POSIX

Real Time Operating Systems and Middleware

Luca Abeni

`luca.abeni@unitn.it`

Processes

- A process implements the notion of *protection*
  - Each process has its own address space
    - And other private resources...
  - A process can write/read in its address space
  - But is not allowed to touch other processes' resources
    - Two processes can share some resources for communication, but this has to be **explicitly allowed** by them!
- Processes usually communicate through *message passing*
  - pipes, sockets, signals, ...

Processes as Active Entities

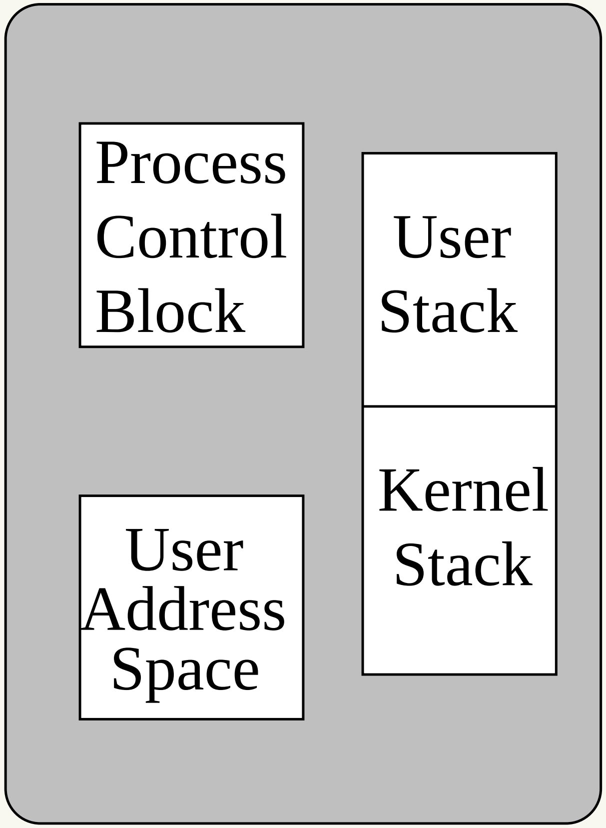
- A process is more than a set of private resources...
- ...It is an active entity!
- Two aspects:
  - Protection / Resource Ownership
  - Execution
    - A process contains at least a schedulable entity, which can access the process's resources
    - Scheduling parameters
    - This schedulable entity is also characterized by (at least) a CPU state and a stack

Single-Threaded Process

Each process has only one thread

- One address space per process
- One stack per process
- One PCB per process
- Other private resources...
- One single execution flow per process

Single-threaded process model

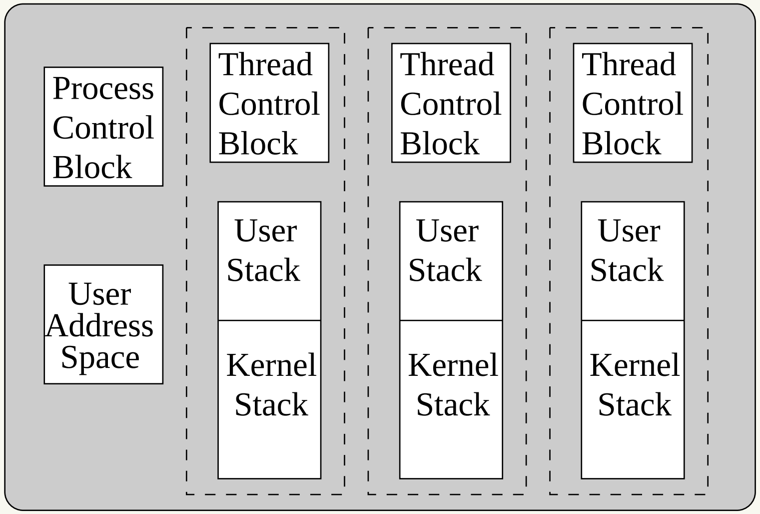


Multi-Threaded Process

A process can have multiple threads running in it

- One address space
- One PCB
- Multiple execution flows in a single process
- Multiple stacks (one per thread)
- A TCB (Thread Control Block) per thread

Multi-threaded process model



A Small Summary about Processes

- Let's recall some quick ideas about processes
- As usual, focus on POSIX (sometimes, Unix / Linux)
  - Not intended to be a complete description about multiprogramming in Unix
  - Refer to manpages (`man <function name>` for more info)
- We will see
  - Process creation / termination
  - Synchronization (IPC, signals)

Process Memory Layout

- Private Address Space
  - User Memory
  - Stack
  - Heap
- User Memory is divided in:
  - Initialized Data Segment
  - BSS
    - Uninitialised global variables
  - Text Segment (program code)
- The heap:
  - Usable through `malloc()` & friends
  - Can grow (`brk()` and `sbrk()`)

Process Identification

- Each process is identified by a Process ID (PID)
- A PID is unique in the system
  - When a new process is created, its PID is returned
  - Each process can obtain its pid by calling `getpid()`

```
pid_t getpid(void)
```

- Note that `getpid()` never fails
  - It never returns values ≤ 0

Process Creation

- A new process can be created by calling `fork()`

```
pid_t fork(void)
```

  - The new process (*child* process) contains a copy of the parent's address space
  - The call has one entry point, and two exit points
    - In the child, 0 is returned
    - In the parent, the PID of the child is returned
  - As usual, a negative value is returned in case of error
- See `TaskCreation/fork.c`

Using fork()

- Typical usage:

```
child_pid = fork();
if (child_pid < 0) {
    perror("Fork");
    return -1;
}
if (child_pid == 0) {
  /* Child body */
} else {
  /* Father body */
}
```

- Simpler version:

```
...
if (child_pid == 0) {
  /* Child body */
  exit(0);
}
/* Father body */
```

Problem: the child address space is a copy of the parent's one, so the child's text segment is the same as the father's one $\Rightarrow$ both the parent's body and the child body must be in the same executable file.

Solution: `exec()`

Changing the Process Text and Data

- Exec: *family* of functions allowing to replace the process address space (text, data, and heap)
  - `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`
  - They differer in the arguments; see the manpage
- Loads a new program, and jump to it
  - Does not create a new process!!! (same PID, same PCB, ...)
  - Returns only on error!
- See `TaskCreation/exec.c`

Typical Exec Usage

```
child_pid = fork();
if (child_pid < 0) {
    perror("Fork");
    return -1;
}
if (child_pid == 0) {
    char *args[3] = {"arg1", "arg2", "arg3"};

    execve("child_body", args, NULL);
    perror("Exec"); /* Why don’t we check the return value? *
    return -1;
}
...
```

- Note: some (non POSIX compliant) systems do not make a distinction between program and process, and only provide a “fork + exec” combo
- POSIX also provides a `system()` function, which does fork + exec (+ wait)

Terminating a Process

- A process terminates:
  1. When it invokes the library call `exit()` or the system call `_exit()`
  2. When it *returns* from its main function
  3. When it is *killed* by some external event (a *signal*)
- When it terminates explicitly, a process can return a result to the parent
- Every process can register a hook to be called on *regular* process termination

```
int atexit(void (*function)(void))
```

  - Handlers are not called if exiting with `_exit()`... Why?

Waiting for a Process

- First form of synchronization between processes:
  - A parent waits for its child's termination
  - `wait()`, `waitpid()`, `wait4()`

    ```
    pid_t wait(int *status)
    ```

    - No children $\Rightarrow$ `wait()` fails (return < 0)
    - At least one terminated child $\Rightarrow$ `wait()` returns the child's exit value, and child's private resources are freed
    - No terminated children $\Rightarrow$ `wait()` blocks
- Extended versions of `wait()`: `waitpid()` (POSIX), `wait3()`, `wait4()` (BSD)
  - Permit to select the child to wait for

Wait, Again

- After a process terminates, its private resources are not freed until its parent performs a `wait()`
- Until the `wait()`, a terminated process is in *zombie* state
  - A good parent has to wait for its children!
  - When the parent of a process dies, the process is reparented to `init` (a system process, with PID 1)
  - $\Rightarrow$ when a process dies, all its zombies are eliminated
- A process can be notified about the termination of a child process through an asynchronous event (signal: `SIGCLD`)

Sinchronization through Signals

- Concurrent processes interact in different ways
  - Competition
  - Cooperation
- Cooperation can be implemented through *signals*
  - Sometimes, a process has to wait until cooperating processes have completed some operation
  - $\Rightarrow$ process τ_i waits for an asynchronous event generated by another process τ_j, or by the system

Signals

- Signal: asynchronous event directed to process τ
- Process τ can:
  - Wait for a signal
  - Perform some other work in the meanwhile, and the signal will interrupt it

Handling Signals

- Signals $\rightarrow$ software equivalent of interrupts
- A process receiving a signal can:
  - Ignore it
  - Interrupt its execution, and jump to a *signal handler*
  - Abort
- A signal that has not generated one of the previous actions yet is a *pending signal*
- We will see how to:
  - Specify how a process handles a signal
  - Mask (block) a signal
  - Check if there are pending signal for a process
  - Generate (or ask the kernel to generate) signals

Signal Handlers

- Signal Table
  - Per process, private, resource
  - Specifies how the process handle each signal
  - At process creation, default values
- The table entries can be modified by using `signal()`, or `sigaction()` (POSIX, more portable)
- Signal handler: `void sighand(int n)`

```
int sigaction(int signum, const struct sigaction *
                          struct sigaction *oldact)
```

- `signum` is the number of the signal we want to modify
- If `oldact` is not null, returns the old handler

Setting a Signal Handler

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

- `sa_handler` is the signal handler, or `SIG_DFL` (default action), or `SIG_IGN` (ignore the signal)
- `sa_mask` is a mask of signals to disable when the handler runs
  - Can be modified using `sigemptyset()`, `sigfillset()`, `sigaddset()`, and `sigdelset()`
- `sa_flags` defines the signal handling behaviour through a set of flags (see manpage)

Sending a Signal

- A process can send a signal to other processes by using the `kill()` system call
  - Note that it must have the proper permissions (user root can send signals to everyone, regular users can send signals only to their own processes)

```
int kill(pid_t pid, int sig)
```

- This is what the `kill` command uses, too...
- Do not be fooled by the name: it is not only used to kill a process (example: `kill -HUP`)

Signal Numbers

- Signals are identified by numbers, and by some *macros*
- `SIGUSR1` and `SIGUSR2`: user defined
- `SIGALRM`, `SIGVTALRM`, and `SIGPROF` are used by process timers (remember?...)
- `SIGKILL` is used to kill a program (used by ”kill -9”)
- `SIGCLD` is raised every time that a child dies
  - Useful for avoiding zombies (the `SIGCLD` handler can perform a `wait()`)
  - If `SIGCLD` is ignored, strange behaviour: zombies are not created
- See `www.disi.unitn.it/~abeni/RTOS/oscillator.c` (try to compile with `-DNOZOMBIE` or `-DHANDLER1`)

Problems with Signals

- Almost all of the signals are reserved for the system
  - Only `SIGUSR{1,2}` are free for user programs
- Signals can be lost
  - If a signal arrives more than 1 time while it is blocked, it is not queued (it will fire only one time)
  - This makes signals quite unreliable for RT IPC...
- Signals do not transport information
  - only the signal number is available to the handler
- Solution: POSIX Real-Time signals

Real-Time Signals

- Multiple instances of real-time signals can be queued
- Real-time signals can transport information
  - Either an integer or a pointer
  - An extended signal handler has to be used

```
void sig_action(int signum, siginfo_t *info, void *ignored)
```

  - Use `sigaction()`, set the `SA_SIGINFO` flag, and set `sa_sigaction()` instead of sa_handler
- There are at least `SIGRTMAX - SIGRTMIN` available signals for user applications
  - They **must** be referred as `SIGRTMIN + n`
- Use `sigqueue()` to send the signal
- `www.disi.unitn.it/~abeni/RTOS/rtsig.c`

RT Signal Information

- Real-time signals carry information, in `siginfo_t`

```
typedef struct {
    int si_signo;
    int si_code;
    union sigval si_value;
} siginfo_t

union sigval {
    int sival_int;
    void *sival_ptr;
}
```

  - `si_signo`: signal number (same as `signo`)
  - `si_value`: information carried by the signal
  - `si_code` identifies the cause of the signal
    - `SI_USER`: sent by a user process (`kill()`)
    - `SI_QUEUE`: sent by a user process (`sigqueue()`)
    - `SI_TIMER`: a POSIX timer expired
    - ... (see documentation)

Sending RT Signals

```
int sigqueue(pid_t p, int n, const union sigval value)
```

- As usual, returns < 0 in case of error
- If no error occurs, queue a signal `n` for process `p`
- Information `value` is transmitted with the signal
- RT Signals can also be generated by the kernel
  - Described by `struct sigevent`

```
struct sigevent {
      int sigev_notify;
      int sigev_signo;
      union sigval;
      void(*)(unsigned sigval) sigev_notify_funct
      (pthread_attr_t*) sigev_notify_attributes;
}
```

  - `sigev_notify`: `SIGEV_NONE`, `SIGEV_SIGNAL`, or `SIGEV_THREAD`

Real-Time Scheduling in POSIX

- POSIX provides support for Real-Time scheduling
- Priority scheduling
  - Multiple priority levels
  - A task queue per priority level
  - The first task from the highest-priority, non empty, queue is scheduled
- POSIX provides multiple scheduling policies
  - A scheduling policy describes how tasks are moved between the priority queues
  - Fixed priority: a task is always in the same priority queue

Real-Time Scheduling in POSIX

- POSIX specifically requires four scheduling policies:
  - `SCHED_FIFO`
  - `SCHED_RR`
  - `SCHED_SPORADIC`
  - `SCHED_OTHER`
- `SCHED_FIFO` and `SCHED_RR` have fixed priorities
- `SCHED_SPORADIC` is a *Sporadic Server* → decreases the response time for aperiodic real-time tasks
- `SCHED_OTHER` is the “traditional” Unix scheduler
  - Dynamic priorities
  - Scheduled in background respect to fixed priorities

Fixed Priorities - 1

- SCHED_FIFO and SCHED_RR use fixed priorities
  - They can be used for real-time tasks, to implement RM and DM
  - Remember: the application developer is in charge of assigning priorities to tasks!
  - Real-time tasks have priority over non real-time (SCHED_OTHER) tasks
- So... What is the difference between these two policies?
  - Only visible when more tasks have the same priority

Fixed Priorities - 2

- `SCHED_FIFO`: priority queues handled in FIFO order
  - When a task start executing, only higher priority tasks can preempt it
- `SCHED_RR`: time is divided in intervals
  - After executing for one interval, a task is removed by the head of the queue, and inserted at the end
- So, there is a difference only if multiple tasks have the same priority
  - Never do this!

SCHED_FIFO vs SCHED_RR

- Only one task per priority level → `SCHED_FIFO` and `SCHED_RR` behave the same way
- More tasks with the same priority
  - With `SCHED_FIFO`, the first task of a priority queue can starve other tasks having the same priority
  - `SCHED_RR` tries serve tasks having the same priority in a more fair way
- The round-robin interval (scheduling quantum) is implementation dependent
- RR and FIFO priorities are comparable. Minimum and maximum priority values can be obtained with `sched_get_priority_min()` and `sched_get_priority_max()`

Setting the Scheduling Policy

```
int sched_get_priority_max(int policy)
int sched_get_priority_min(int policy)

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param)
int sched_setparam(pid_t pid,
                   const struct sched_param *param)
```

- If `pid == 0`, then the parameters of the running task are changed
- The only meaningful field of `struct sched_param` is `sched_priority`

Problems with Real-Time Priorities

- In general, “regular” (`SCHED_OTHER`) tasks are scheduled in background respect to real-time ones
- A real-time task can preempt / starve other applications
- Example: the following task scheduled at high priority can make the system unusable

```
void bad_bad_task()
{
      while(1);
}
```

  - Real-time computation have to be limited (use real-time priorities only when **really needed**!)
  - Running applications with real-time priorities requires root privileges (or part of them!)

Memory Swapping and Real-Time

- The *virtual memory* mechanism can swap part of the process address space to disk
  - Memory swapping can increase execution times unpredictabilities
  - Not good for real-time applications
- A real-time task can lock part of its address space in main memory
  - Locked memory cannot be swapped out of the physical memory
  - This can result in a DoS (physical memory exhausted!!!)
- Memory locking can be performed only by applications having (parts of) the root privileges!

Memory Locking Primitives

- `mlock()`: lock some pages from the process address space into main memory
  - Makes sure this region is always loaded in RAM
- `munlock()`: unlock previously locked pages
- `mlockall()`: lock the whole address space into main memory
  - Can lock the *current* address space only, or all the future allocated memory too
  - Can be used to disable “lazy allocation” techniques
- These functions are defined in `sys/mman.h`
  - Please check the manpages for details