Processes and Threads

What are processes?
How does the operating system manage them?

Nick Urbanik
nicku@nicku.org

A computing department

Copyright Conditions: Open Publication License

(http://www.opencontent.org/openpub/)

OSIS — ver. 1.5
Processes — p. 164

What is a process?

A process is a program in execution
Each process has a process ID
In Linux, 
$ ps ax$
promts one line for each process.
A program can be executed a number of times simultaneously.
Each is a separate process.

What is a thread?

A thread is a lightweight process
Takes less CPU power to start, stop
Part of a single process
Shares address space with other threads in the same process
Threads can share data more easily than processes
Sharing data requires synchronization, i.e., locking — see slide 51.

This shared memory space can lead to complications in programming:
"Threads often prevent abstraction. In order to prevent deadlock, you often need to know how and if the library you are using uses threads in order to avoid deadlock problems. Similarly, the use of threads in a library could be affected by the use of threads at the application layer." — David Korn

Program counter

The code of a process occupies memory
The Program counter (PC) is a CPU register
PC holds a memory address...
... of the next instruction to be fetched and executed
Environment of a process

- The environment is a set of names and values
- Examples:
  PATH=/usr/bin:/bin:/usr/X11R6/bin
  HOME=/home/nicku
  SHELL=/bin/bash
- In Linux shell, can see environment by typing:
  ```
  $ set
  ```

Permissions of a Process

- A process executes with the permissions of its owner
  - The owner is the user that starts the process
- A Linux process can execute with permissions of another user or group
  - If it executes as the owner of the program instead of the owner of the process, it is called set user ID
  - Similarly for set group ID programs

Multitasking

- Our lab PCs have one main CPU
  - But multiprocessor machines are becoming increasingly common
  - Linux 2.6.x kernel scales to 16 CPUs
- How execute many processes "at the same time"?

Multitasking — 2

- CPU rapidly switches between processes that are “ready to run”
  - Really: only one process runs at a time
  - Change of process called a context switch
    - See slide §36
  - With Linux: see how many context switches/second using `vmstat` under "system" in column "cs"

Multitasking — 3

- This diagram shows how the scheduler gives a "turn" on the CPU to each of four processes that are ready to run
  - Processes may have parents and children
    - Gives a family tree
    - In Linux, see this with commands:
      ```
      $ pstree
      or
      $ ps axf
      ```

Birth of a Process

- In Linux, a process is born from a `fork()` system call
  - A system call is a function call to an operating system service provided by the kernel
  - Each process has a parent
    - The parent process calls `fork()`
    - The child inherits (but cannot change) the parent environment, open files
  - Child is identical to parent, except for return value of `fork()
    - Parent gets child's process ID (PID)
    - Child gets 0

Scheduler

- OS decides when to run each process that is ready to run ("runnable")
  - The part of OS that decides this is the scheduler
  - Scheduler aims to:
    - Maximise CPU usage
    - Maximise process completion
    - Minimise process execution time
    - Minimise waiting time for ready processes
    - Minimise response time
### When to Switch Processes?

The scheduler may change a process between executing (or running) and ready to run when any of these events happen:
- clock interrupt
- I/O interrupt
- Memory fault
- trap caused by error or exception
- system call

See slide §17 showing the running and ready to run process states.

### Scheduling statistics: `vmstat`

- The "system" columns give statistics about scheduling:
  - "cs" — number of context switches per second
  - "in" — number of interrupts per second

See slide §25, `man vmstat`

### Interrupts

Will discuss interrupts in more detail when we cover I/O

An interrupt is an event (usually) caused by hardware that causes:
- Saving some CPU registers
- Execution of interrupt handler
- Restoration of CPU registers
- An opportunity for scheduling

### Process States

**Running**
- actually contains two states:
  - executing, or
  - ready to execute

**Interruptable** — a blocked state
- waiting for event, such as:
  - end of an I/O operation,
  - availability of a resource, or
  - a signal from another process

**Uninterruptable** — another blocked state
- waiting directly on hardware conditions
- will not accept any signals (even `SIGKILL`)

### What is Most Common State?

Now, my computer has 160 processes.

- How many are running, how many are ready to run, how many are blocked?
- What do you expect is most common state?

### Most Processes are Blocked

Here you see that most are sleeping, waiting for input!

- Most processes are "I/O bound": they spend most time waiting for input or waiting for output to complete
- With one CPU, only one process can actually be running at one time
- However, surprisingly few processes are ready to run
- The load average is the average number of processes that are in the ready to run state
- In output from the top program above, see over last 60 seconds, there are 2.02 processes on average in RTR state

---

OSSI — ver. 1.5
Processes - p. 14/66

OSSI — ver. 1.5
Processes - p. 15/66

OSSI — ver. 1.5
Processes - p. 16/66

OSSI — ver. 1.5
Processes - p. 17/66

OSSI — ver. 1.5
Processes - p. 18/66

OSSI — ver. 1.5
Processes - p. 19/66

OSSI — ver. 1.5
Processes - p. 20/66

OSSI — ver. 1.5
Processes - p. 21/66
Linux Process States — 3

**Stopped** — process is halted
- can be restarted by another process
- e.g., a debugger can put a process into stopped state

**Zombie** — a process has terminated
- but parent did not wait() for it

Process States: vmstat

- The “procs” columns give info about process states:
  - “r” — number of processes that are in the ready to run state
  - “b” — number of processes that are in the uninterruptable blocked state

Tools for monitoring processes

- **vmstat**
  - Good to monitor over time:
    - `$ vmstat 5`
- **procinfo**
  - Easier to understand than vmstat
  - Monitor over time with
    - `$ procinfo -f`
  - View processes with top — see slides[27] to [30]
  - The system monitor sar shows data collected over time:
    - See man sar; investigate sar -c and sar -q
  - See the utilities in the procps software package. You can list them with
    - `$ rpm -ql procps`

Monitoring processes in Win 2000

Windows 2000 provides a tool:
- Start → Administrative Tools → Performance. Can use this to monitor various statistics

Tools for monitoring processes

- **vmstat**
  - Good to monitor over time:
    - `$ vmstat 5`
- **procinfo**
  - Easier to understand than vmstat
  - Monitor over time with
    - `$ procinfo -f`
  - View processes with top — see slides[27] to [30]
  - The system monitor sar shows data collected over time:
    - See man sar; investigate sar -c and sar -q
  - See the utilities in the procps software package. You can list them with
    - `$ rpm -ql procps`

Monitoring processes in Win 2000

- Tools for monitoring processes
  - **vmstat**
    - Good to monitor over time:
      - `$ vmstat 5`
  - **procinfo**
    - Easier to understand than vmstat
    - Monitor over time with
      - `$ procinfo -f`
    - View processes with top — see slides[27] to [30]
    - The system monitor sar shows data collected over time:
      - See man sar; investigate sar -c and sar -q
    - See the utilities in the procps software package. You can list them with
      - `$ rpm -ql procps`

Monitoring processes in Win 2000

- Windows 2000 provides a tool:
  - Start → Administrative Tools → Performance. Can use this to monitor various statistics

Process Monitoring with top

**top**: load average

- Over that time is the average number of processes that are ready to run, but which are not executing
- A measure of how “busy” a computer is.

**top**: process states

- sleeping Most processes (109/111) are sleeping, waiting for I/O
- running This is the number of processes that are both ready to run and are executing
- zombie There is one process here that has terminated, but its parent did not wait() for it.
  - The wait() system calls are made by a parent process, to get the exit() status of its child(ren).
  - This call removes the process control block from the process table, and the child process does not exist any more. (§34)
- stopped When you press (Control-z) in a shell, you will increase this number by 1
**top: Processes and Memory**

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PRI</th>
<th>NI</th>
<th>SIZE</th>
<th>RSS</th>
<th>SHARE</th>
<th>STAT</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1253</td>
<td>root</td>
<td>15</td>
<td>0</td>
<td>73996</td>
<td>13M</td>
<td>11108 D</td>
<td>S</td>
<td>2.9</td>
<td>5.5</td>
<td>19:09</td>
<td>X</td>
</tr>
</tbody>
</table>

**SIZE** This column is the total size of the process, including the part which is swapped (paged out) out to the swap partition or swap file

Here we see that the process X uses a total of 73,996 KB, i.e., 73,996 × 1024 bytes ≈ 72MB, where here 1MB = 20 bytes.

**RSS** The resident set size is the total amount of RAM that a process uses, including memory shared with other processes. Here X uses a total of 13MB RAM, including RAM shared with other processes.

**SHARE** The amount of shared memory is the amount of RAM that this process shares with other processes. Here X shares 11,108 KB with other processes.

---

**Virtual Memory: suspended processes**

- With memory fully occupied by processes, could have all in blocked state!
- CPU could be completely idle, but other processes waiting for RAM
- Solution: virtual memory
  - will discuss details of VM in memory management lecture
- Part or all of process may be saved to swap partition or swap file

---

**OS Process Control Structures**

- Every OS provides **process tables** to manage processes
- In this table, the entries are called **process control blocks** (PCBs), **process descriptors** or **task descriptors**. We will use the abbreviation PCB.
- There is one PCB for each process
- In Linux, PCB is called **task_struct**, defined in include/linux/sched.h
- In a Fedora Core or Red Hat system, you will find it in the file /usr/src/linux-2.4/include/linux/sched.h
- If you have installed the kernel-source software package

---

**Process Control Blocks**

**The Process Table**

Data structure in OS to hold information about a process

---

**What is in a PCB**

- In slide §3, we saw that a PCB contains:
  - a process ID (PID)
  - process state (i.e., executing, ready to run, sleeping waiting for input, stopped, zombie)
  - program counter, the CPU register that holds the address of the next instruction to be fetched and executed
  - The value of other CPU registers the last time the program was switched out of executing by a context switch — see slide §30
  - scheduling priority
  - the user that owns the process
  - the group that owns the process
  - pointers to the **parent process**, and **child processes**
  - Location of process’s data and program code in memory,
  - list of allocated resources (including open files)

---

**Execution Context**

- Also called state of the process (but since this term has two meanings, we avoid that term here), process context or just context
- The execution context is all the data that the OS must save to stop one process from executing on a CPU, and load to start the next process running on a CPU
- This includes the content of all the CPU registers, the location of the code, ... includes most of the contents of the process’s PCB.

---

**Context Switch**

- OS does a context switch when:
  - stop current process from executing, and
  - start the next ready to run process executing on CPU
- OS saves the execution context (see §37) to its PCB
- OS loads the ready process’s execution context from its PCB
- When does a context switch occur?
  - When a process blocks, i.e., goes to sleep, waiting for input or output (io), or
  - When the scheduler decides the process has had its turn of the CPU, and it’s time to schedule another ready-to-run process
- A context switch must be as fast as possible, or multitasking will be too slow
- Very fast in Linux OS
**Program Counter in PCB**

- What value is in the program counter in the PCB?
- If it is **not** executing on the CPU,
  - The address of the next CPU instruction that **will be** fetched and executed the next time the program starts executing.
- If it is **executing** on the CPU,
  - The address of the first CPU instruction that **was** fetched and executed when the process began executing at the last context switch (**§39**).

**PCB Example: Diagram**

- The diagram in slide (**§40**) shows three processes and their process control blocks.
- There are seven snapshots: $t_0$, $t_1$, $t_2$, $t_3$, $t_4$, $t_5$, and $t_6$, where the scheduler has changed process (there has been a context switch—§36).
- On this particular example CPU, all I/O instructions are 2 bytes long.
- The diagram also shows the queue of processes in the:
  - *Ready queue* (processes that are ready to run, but do not have a CPU to execute on yet)
  - *Blocked queue* (processes that are waiting for CPU)

**PCB Example — Continued**

- In slide (**§45**):
  - The times $t_0$, $t_1$, $t_2$, $t_3$, $t_4$, $t_5$, and $t_6$ are when the scheduler has selected another process to run.
  - Note that these time intervals are **not** equal, they are just the points at which a scheduling change has occurred.
  - Each process has stopped at one stage to perform I/O
  - That is why each one is put on the wait queue once during its execution.
  - Each process has performed I/O once

**What is the address of I/O instructions?**

- We are given that all I/O instructions in this particular example are **two bytes** long (slide §39).
- We can see that when the process is sleeping (i.e., blocked), then the program counter points to the instruction after the I/O instruction.
- So for process P1, which blocks with program counter PC = C0DE16, the I/O instruction is at address C0DE16 − 2 = C0DC16
- for process P2, which blocks with program counter PC = FEED16, the I/O instruction is at address FEED16 − 2 = FEED16
- for process P3, which blocks with program counter PC = D1CE16, the I/O instruction is at address D1CE16 − 2 = D1CC16

**Major process Control System Calls**

- **fork()** — start a new process
- **execve()** — replace calling process with machine code from another program file
- **wait(), waitpid()** — parent process gets status of its’ child after the child has terminated, and cleans up the process table entry for the child (stops it being a zombie)
- **exit()** — terminate the current process

**Process System Calls**

**How the OS controls processes**

**How you use the OS to control processes**

**IPC**

**Inter Process Communication**

**How Processes can Talk to Each Other**
**Problem with Processes**

- Communication!
- Processes cannot see the same variables
- Must use **Inter Process Communication (IPC)**
- IPC Techniques include:
  - pipes, and named pipes (FIFOs)
  - sockets
  - messages and message queues
  - shared memory regions
- All have some overhead

**Interprocess Communication (IPC)**

- **Pipe** — circular buffer, can be written by one process, read by another
- Related processes can use **unnamed pipes**
  - used in shell programming, e.g., the vertical bar ‘|’ in
  - `$ find /etc | xargs file`
- Unrelated processes can use **named pipes** — sometimes called **FIFOs**
- Messages — POSIX provides system calls `msgsnd()` and `msgrcv()`
  - Message is block of text with a type
- Others
  - Signals are implemented as single bits in a field in the
  - PCB
- Signals are broadcasted across all processes
- Signals cannot be queued
- Signals can be handled with a function call
  - `trap signal_handler INT QUIT TERM`
- Signals are handled when a process is in interruptible sleep
- Signals can be ignored
- Signals can be caught
- Signals can be sent
- Signals are handled with a function call
  - `trap signal_handler INT QUIT TERM`

**IPC — Shared Memory**

- **Shared Memory** — a Common block of memory shared by many processes
- Fastest way of communicating
- Requires synchronisation (See slide 51)

**IPC — Signals**

- Some **signals** can be generated from the keyboard, i.e.,
  - **Control-C** — interrupt (SIGINT)
  - **Control-Z** — stop (SIGSTOP)
- **Signals**
  - **SIGINT** — a Common block of memory shared by many processes
  - Fastest way of communicating
  - Requires synchronisation (See slide 51)

**Signals and the Shell**

- We can use the **kill** built in command to make the
  - `kill()` system call to send a signal
- A shell script uses the **trap** built in command to handle a signal
- Ignoring the signals `SIGINT`, `SIGQUIT` and `SIGTERM`:
  - `trap "" INT QUIT TERM`
- Handling the same signals by printing a message then exiting:
  - `trap "echo "Got a signal; exiting.";exit 1" INT QUIT TERM`
- Handling the same signals with a function call:
  - `signal_handler() {
     echo "Received a signal; terminating."
     rm -f $temp_file
     exit 1
   }
   trap signal_handler INT QUIT TERM`

**Threads**

- **Lightweight processes that can talk to each other easily**
  - `stack pointer`
  - `register values`
  - `scheduling properties, such as policy or priority`
  - `set of signals they can each block or receive`
  - `own stack data (local variables are local to thread)`

**Threads and Processes**

- **Threads**
  - In a process
    - all share the same address space
    - Communication easier
    - Overhead less
    - Problems of locking and deadlock a major issue
- **Processes**
  - Have separate address spaces
  - Communication more indirect: **IPC** (Inter Process Communication)
  - Overhead higher
  - Less problem with shared resources (since fewer resources to share)
Threads share a lot

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- A number of threads can read and write to the same memory locations, and so you need to explicitly synchronize access.

Problem with threads:

- Avoid 2 or more threads writing or reading and writing same data at the same time
- Avoid data corruption
- Need to control access to data, devices, files
- Need locking
- Provide three methods of locking:
  - mutex (mutual exclusion)
  - semaphores
  - condition variables

Race Conditions

- race condition — where outcome of computation depends on scheduling
- an error in coding
- Example: two threads both access same list with code like this:
  
```c
if ( list.numitems > 0 ) {
    // Oh, dear, better not change to other thread here!
    remove_item( list ); // not here!
    // ... and not here either:
    --list.numitems;
}
```

Race Condition

Critical Sections

- critical resource — a device, file or piece of data that cannot be shared
- critical section — part of program only one thread or process should access contains a critical resource
  - i.e., you lock data, not code
- All the code in the previous slide is a critical section
- Consider the code:
  
```c
very_important_count++;
```
- executed by two threads on a multiprocessor machine (SMP = symmetric multiprocessor)

Example — another possibility

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read very_important_count (5)</td>
<td>read very_important_count (5)</td>
</tr>
<tr>
<td>add 1 (6)</td>
<td>add 1 (6)</td>
</tr>
<tr>
<td>write very_important_count (6)</td>
<td>write very_important_count (6)</td>
</tr>
</tbody>
</table>

Solution: Synchronisation

- Solution is to recognise critical sections
- use synchronisation, i.e., locking, to make sure only one thread or process can enter critical region at one time.
- Methods of synchronisation include:
  - file locking
  - semaphores
  - monitors
  - spinlocks
  - mutexes
File Locking

For example, an `flock()` system call can be used to provide exclusive access to an open file. The call is atomic:
- It either:
  - completely succeeds in locking access to the file, or
  - it fails to lock access to the file, because another thread or process holds the lock
- No “half-locked” state
- No race condition

Alternatives can result in race conditions; for example:
- thread/process 1 checks lockfile
- thread/process 2 checks lockfile a very short time later
- both processes think they have exclusive write access to the file
- file is corrupted by two threads/processes writing to it at the same time

Summary — Process States, Scheduling

Scheduler changes processes between ready to run and running states
- context switch: when scheduler changes process or thread
- Most processes are blocked, i.e., sleeping: waiting for I/O
- understand the process states
- why a process moves from one state to another
- Communication between processes is not trivial; IPC methods include:
  - pipes
  - messages
  - shared memory
  - signals
  - semaphores

Summary — Processes and Threads

With Linux and Unix, main process system calls are `fork()`, `exec()` and `wait()`.
- Threads are lightweight processes
  - part of one process
  - share address space
  - can share data easily
  - sharing data requires synchronisation, i.e., locking

References

There are many good sources of information in the library and on the Web about processes and threads. Here are some I recommend: