Contents

Processes and Threads

What are processes?

How does the operating system manage them?

Nick Urbanik

nicku@nicku.org

A computing department

What is a process?

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

What is a process? — 2

- A process includes current values of:
  - Program counter
  - Registers
  - Variables
- A process also has:
  - The program code
  - It's own address space, independent of other processes
  - A user that owns it
  - A group owner
  - An *environment* and a *command line*
- This information is stored in a *process control block*, or *task descriptor* or *process descriptor*
  - a data structure in the OS, in the *process table*
  - See slides starting at §34.

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 *synchronisation*, i.e., locking — see slide 61.
- 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*

See page 180, ESR in references, §66.

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

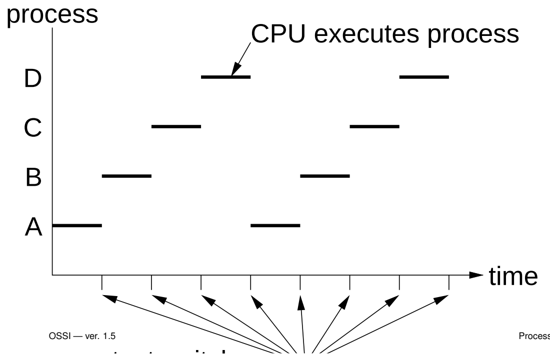
- Our lab PCs have one main CPU
  - But multiprocessor machines are becoming increasingly common
  - Linux 2.6.x kernel scales to 16 CPUs
- How execute many processes “at the same time”?

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



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

Process tree

- Processes may have parents and children
- Gives a family tree
- In Linux, see this with commands:
  `$` **`pstree`**
  or
  `$` **`ps axf`**

Scheduler

- OS decides when to run each process that is ready to run (“runable”)
- 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.

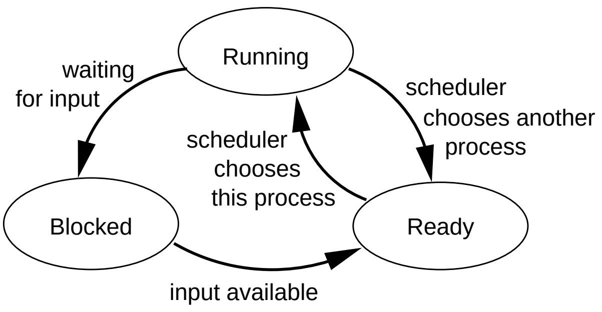
Scheduling statistics: `vmstat`

- The "`system`" columns give statistics about *scheduling*:
  - "`cs`" — number of context switches per second
  - "`in`" — number of interrupts per second
- See slide §36, `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



What is Most Common State?

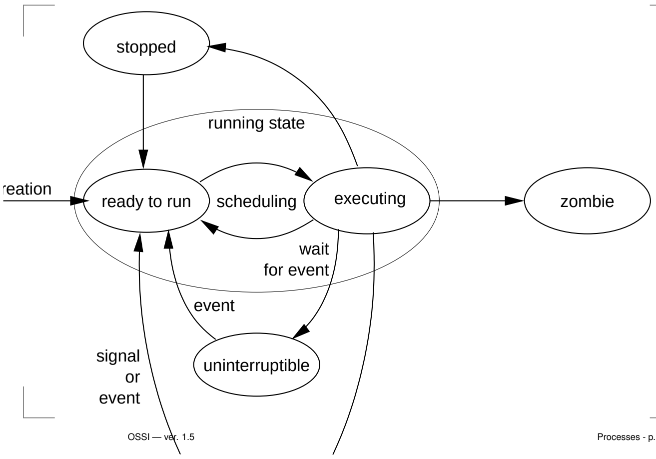
- Now, my computer has 160 processes.
- How many are running, how many are ready to run, how many are blocked?
- What do you expect is most common state?

Most Processes are Blocked

```
9:41am  up 44 days, 20:12,  1 user,  load average: 2.02, 2.06, 2.13
160 processes: 145 sleeping, 2 running, 13 zombie, 0 stopped
```

- 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

Linux Process States



Linux Process States — 2

- *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`)

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

- Linux provides:
- **`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`

Process Monitoring — `top`

```
  08:12:13  up 1 day, 13:34,  8 users,  load average: 0.16, 0.24, 0.49
111 processes: 109 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total    0.0%    0.0%    3.8%   0.0%     0.0%    0.0%   96.1%
Mem:   255608k av,  245064k used,   10544k free,       0k shrd,   17044k buff
       152460k active,              63236k inactive
Swap: 1024120k av,  144800k used,  879320k free                  122560k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 1253 root      15   0 73996  13M 11108 S     2.9  5.5  19:09   0 X
 1769 nicku     16   0  2352 1588  1488 S     1.9  0.6   2:10   0 magicdev
23548 nicku     16   0  1256 1256   916 R     1.9  0.4   0:00   0 top
    1 root      16   0   496  468   440 S     0.0  0.1   0:05   0 init
    2 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kapmd
    4 root      34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd/0
    6 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 bdflush
    5 root      15   0     0    0     0 SW    0.0  0.0   0:11   0 kswapd
```


`top`: load average

```
08:12:13  up 1 day, 13:34,  8 users,  load average: 0.16, 0.24, 0.49
```

- *load average* is measured over the last minute, five minutes, fifteen minutes
- 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

```
111 processes: 109 sleeping, 1 running, 1 zombie, 0 stopped
```

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

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
 1253 root      15   0 73996  13M 11108 S     2.9  5.5  19:09   0 X
```

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., $73{,}996 \times 1024$ bytes $\approx$ 72MB, where here 1MB $= 2^{20}$ 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

Suspended Processes

- Could add more states to process state table:
  - ready and suspended
  - blocked and suspended

Process Control Blocks

The Process Table

Data structure in OS to hold information about a process

OS Process Control Structures

- Every OS provides *process tables* to manage processes
- In this table, the entries are called *process control blocks* (PCBs), *process descriptor*s or *task descriptor*s. 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.*/include/linux/sched.h` if you have installed the `kernel-source` software package

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 §36
  - 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*)

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 (I/O), 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

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.

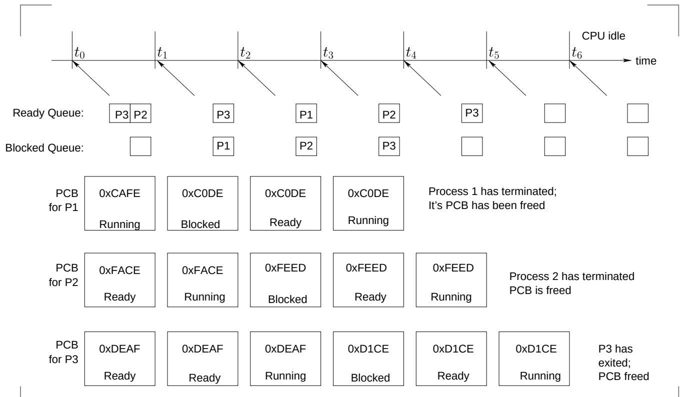
Program Counter in PCB

- 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 (§36)

Process Control Blocks—Example

- 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 at which 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*, or *Wait queue*, where the processes have been blocked because they are waiting for I/O to finish.

PCB Example: Diagram



PCB Example — Continued

- In slide §40,
  - 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 = C0DE_{16}, the I/O instruction is at address $\text{C0DE}_{16} - 2 = \text{C0DC}_{16}$
  - for process P2, which blocks with program counter PC = FEED_{16}, the I/O instruction is at address $\text{FEED}_{16} - 2 = \text{FEEB}_{16}$
  - for process P3, which blocks with program counter PC = D1CE_{16}, the I/O instruction is at address $\text{D1CE}_{16} - 2 = \text{D1CC}_{16}$

Process System Calls

How the OS controls processes

How you use the OS to control processe

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

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
  - each process has a message queue, like a mailbox
  - processes are suspended when attempt to read from empty queue, or write to full queue.

IPC — Shared Memory

- *Shared Memory* — a Common block of memory shared by many processes
- Fastest way of communicating
- Requires synchronisation (See slide 61)

IPC — Signals

- Some *signals* can be generated from the keyboard, i.e., Control-C — interrupt (`SIGINT`); Control-\ — quit (`SIGQUIT`), Control-Z — stop (`SIGSTOP`)
- A process sends a signal to another process using the **`kill()`** system call
- signals are implemented as single bits in a field in the PCB, so cannot be queued
- A process may respond to a signal with:
  - a *default action* (usually process terminates)
  - a *signal handler* function (see **`trap`** in shell programming notes), or
  - ignore the signal (unless it is `SIGKILL` or `SIGSTOP`)
- A process *cannot ignore*, or handle a `SIGSTOP` or a `SIGKILL` signal.
  - A `KILL` signal will *always terminate* a process (unless it is in interruptible sleep)

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

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 have own. . .

- 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 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 *synchronise* 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 (**mut**ual **ex**clusion)
  - semaphores
  - condition variables

Race Condition

Race Conditions

- *race condition* — where outcome of computation depends on sheduling
- an error in coding
- Example: two threads both access same list with code like this:

```
if ( list.numitems > 0 ) {
        // Oh, dear, better not change to
        // other thread here!
        remove_item( list ); // not here!
        // ...and not here either:
        --list.numitems;
}
```


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:
  `very_important_count++;`
- executed by two threads on a multiprocessor machine (SMP = **s**ymmetric **m**ulti**p**rocessor)

Race Condition — one possibility

thread 1	thread 2
read `very_important_count` (5)	
add 1 (6)	
write `very_important_count` (6)	
	read `very_important_count` (6)
	add 1 (7)
	write `very_important_count` (7)

Example — another possibility

thread 1	thread 2
read `very_important_count` (5)	
	read `very_important_count` (5)
add 1 (6)	
	add 1 (6)
write `very_important_count` (6)	
	write `very_important_count` (6)

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 and References

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:

- *Operating Systems: A Modern Perspective: Lab Update*, 2nd Edition, Gary Nutt, Addison-Wesley, 2002. A nice text book that emphasises the practical (like I do!)
- William Stallings, *Operating Systems*, Fourth Edition, Prentice Hall, 2001, chapters 3, 4 and 5
- Deitel, Deitel and Choffnes, *Operating Systems*, Third Edition, Prentice Hall, 2004, ISBN 0-13-1182827-4, chapters 3, 4 and 5
- Paul Rusty Russell, *Unreliable Guide To Locking* `http://kernelnewbies.org/documents/kdoc/kernel-locking/lklockingguide.html`
- Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley, 2004, ISBN 0-13-142901-9.