<table-of-contents><table-of-contents></table-of-contents></table-of-contents>	Processes and Threads What are processes? How does the operating system manage them? Nick Urbanik nikku@nicku.org A computing department	(see http://www.opencontent.org/openpub/) <pre> ossl-weis What is a process? 2 </pre>	 A process includes current values of: Program counter Program counter Registers Variables A process also has: Variables A process also has: The program code If's own address space, independent of other processes A user that owns it A user that owns it A group owner An <i>environment</i> and a <i>command line</i> This information is stored in a <i>process control block</i>, or task descriptor or process descriptor a data structure in the os, in the <i>process table</i> See slides starting at §34. Process table 	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 	OSSI – ver. 1.5 Processes - p. 566
Number Number	IPC	Silde 45 Silde 45 Silde 47 Silde 47 Silde 48 Silde 49 Silde 50 Silde 55 Silde 55 Silde 55 Silde 55 Silde 60 Silde 61 Silde 62 Silde 63 Silde 63 Silde 65 Silde 65 Silde 65 Silde 65	 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 thread?	A <i>thread</i> 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 <i>synchronisation</i>, i.e., locking — see slide 61. This shared memory space can lead to complications in programming: 	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." – Darkt Kostn.s

9 9

٩ 9 9 ۹ ٩

۹

Environment of a process Permissions of a Process The environment is a set of names and values A process executes with the permissions of its owner The owner is the user that starts the process Examples: A Linux process can execute with permissions of another PATH=/usr/bin:/bin:/usr/X11R6/bin HOME=/home/nicku user or group SHELL=/bin/bash If it executes as the owner of the program instead of the In Linux shell, can see environment by typing: owner of the process, it is called set user ID \$ set Similarly for set group ID programs **Multitasking** Multitasking — 2 Our lab PCs have one main CPU CPU rapidly switches between processes that are "ready But multiprocessor machines are becoming to run" increasingly common Really: only one process runs at a time Linux 2.6.x kernel scales to 16 CPUs Change of process called a context switch How execute many processes "at the same time"? See slide §36 With Linux: see how many context switches/second using vmstat under "system" in column "cs" OSSI - ver. 1.5 OSSI - ver. 1.5 Multitasking — 3 **Birth of a Process** This diagram shows how the scheduler gives a "turn" on In Linux, a process is born from a fork () system call the CPU to each of four processes that are ready to run • A system call is a function call to an operating system service provided by the kernel process Each process has a parent CPU executes process The parent process calls fork () D The child inherits (but cannot change) the parent environment, open files С Child is *identical* to parent, except for return value of В fork(). Parent gets child's process ID (PID) Child gets 0 A time OSSI - ver. 1.5 **Scheduler Process tree** Processes may have parents and children OS decides when to run each process that is ready to ۰ run ("runable") Gives a family tree The part of OS that decides this is the scheduler In Linux, see this with commands: Scheduler aims to: \$ pstree or Maximise CPU usage Maximise process completion \$ ps axf Minimise process execution time Minimise waiting time for ready processes Minimise response time

Processes - p. 12/6

OSSI - ver. 1.5

When to Switch Processes?

Scheduling statistics: vmstat

- 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.
- 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



Linux Process States — 3

Process States: vmstat

- 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

• 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

OSSI – ver. 1.5 Processes - p. 22/6					
Tools for monitoring processes	Monitoring processes in Win 2000				
 Linux provides: vmstat Good to monitor over time: \$ vmstat 5 procinfo 	 Windows 2000 provides a tool: Start → Administrative Tools → Performance. Can use this to monitor various statistics 				
 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					
SSI-ver. 1.5 Processes - p. 2466					
NE 1.11 -1-LL Law W	Process Monitoring — top				
Process Monitoring with 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 CFU states: cpu user nice system irq softirq iowait idle total 0.0% 0.0% 3.8% 0.0% 0.0% 9.6.1% Mem: 255608k av, 245064k used, 10544k free, 0k shrd, 17044k buff 152460k active, 63236k inactive Swap: 1024120k av, 144800k used, 879320k free 122560k cached DTD USER DEL NI, SIJE PSS CADE STAT & CDU AMEN THE 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 0 256 125 916 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 SW 0.0 0.0 0:00 0 keventd 3 root 15 0 0 0 SW 0.0 0.0 0 ksoftirqd/0 6 root 15 0 0 0 SW 0.0 0.0 0 ksottirqd/0 5 root 15 0 0 0 SW 0.0 0.0 0:11 0 kswapd				
OSSI—ver. 1.5 Processes - p. 2086	OSSI – ver. 1.5 Processes - p. 2766				
top: load average	top: process states				
 08:12:13 up 1 day, 13:34, 8 users, load average: 0.16, 0.24, 0.49 <i>load average</i> is measured over the last minute, five minutes, fifteen minutes 	sleeping Most processes (109/111) are sleeping, waiting for				
Over that time is the average number of processes that are ready to run, but which are not executing	running This is the number of processes that are both ready to run and are executing				
A measure of how "busy" a computer is.	 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) 				
OSSI – ver. 1.5 Processes - p. 20/66	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	х

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.

Suspended Processes

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

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

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 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.*/include/linux/sched.h
if you have installed the kernel-source software
package

Context Switch

OS does a context switch when:

OSSI - ver. 1.5

- 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

What is in a PCB

- In slide §3, we saw that a PCB contains:
 - a process ID (PID)

OSSI - ver. 1.5

- 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

OSSI - ver 15

- 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.s
 List of allocated recourses (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.

e - n 37/6

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 t₀, t₁, t₂, t₃, t₄, t₅ and t₆ 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.



In slide §40,

OSSI - ver. 1.5

OSSI - ver. 1.5

0991 - ver 1.5

- 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 VO
 That is why each one is put on the *wait queue* once during its execution.
- Each process has performed I/O once

PCB Example: Diagram



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 $PC = CODE_{16}$, the VO instruction is at address $CODE_{16} 2 = CODC_{16}$
- for process P2, which blocks with program counter $PC = FEED_{16}$, the ν O instruction is at address $FEED_{16} 2 = FEEB_{16}$
- for process P3, which blocks with program counter $PC = D1CE_{16}$, the ν_0 instruction is at address $D1CE_{16} 2 = D1CC_{16}$

Major process Control System Calls

fork() — start a new process

OSSI - ver. 1.5

- 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 processe

IPC

Inter Process Communication

How Processes can Talk to Each Other

Processes - p. 44

Processes - n 45/66

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

OSSI - ver. 1.5

kill() system call

SIGKILL signal.

. 48/66

PCB, so cannot be queued

programming notes), or

- 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 — Signals
 Some signals can be generated from the keyboard, i.e.,

(Control-C) — interrupt (SIGINT); (Control-\) — quit

A process sends a signal to another process using the

signals are implemented as single bits in a field in the

ignore the signal (unless it is SIGKILL or SIGSTOP)
 A process *cannot ignore*, or handle a SIGSTOP or a

A KILI signal will *always terminate* a process (unless jit is in interruntible sleep)

(SIGQUIT), (Control-Z) — stop (SIGSTOP)

A process may respond to a signal with:
 a *default action* (usually process terminates)
 a *signal handler* function (see trap in shell

IPC — Shared Memory

- Shared Memory a Common block of memory shared by many processes
- Fastest way of communicating

OSSI - ver. 1.5

Requires synchronisation (See slide 61)

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
- Llondling the same signal, with a function call.
- Handling the same signals with a function call: signal_handler() { echo "Received a signal; terminating." rm -f \$temp_file exit 1
 - trap^{ssi}sïdhal_handler INT QUIT TERM

Threads and Processes

- Threads in a process all share the same address space
- Communication easier
- Overhead less

1

- 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

Lightweight processes that can talk to each other easily

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)

OSSI — ver. 1.5

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 (mutual exclusion)
- semaphores

OSSI - ver. 1.5

p. 56/66

condition variables

OSSI — ver. 1.5

OSSI - ver. 1.5

OSSI - ver. 1.5

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: very_important_count++;
- executed by two threads on a multiprocessor machine (SMP = symmetric multiprocessor)

Race Condition — one possibility

thread 1	thread 2	
read very_important_count (5)		
add 1 (6)		
write very_important_count (6)		
	read very_importan	nt_co

read very_important_count (6)
add 1 (7)
write very_important_count (7)

Example — another possibility

thread 1	thread 2				
<pre>read very_important_count (5)</pre>					
	<pre>read very_important_count (5)</pre>				
add 1 (6)					
	add 1 (6)				
<pre>write very_important_count (6)</pre>					
	<pre>write very_important_count (6)</pre>				

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

OSSI - ver. 1.5

- 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

OSSI - ver. 1.5

- shared memory
- signals
- semaphores

Summary and References

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

OSSI - ver. 1.5

- 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.