Processes 1

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

June 9, 2017

Overview I

1. POSIX
2. Multiprogramming
   - Three Views
   - Side Effects
   - Process vs Program
   - Process Creation
   - Process Termination
   - Process Hierarchies
   - Process States
   - Process Model
   - Implementation of Processes
   - Modelling Multiprogramming
   - Execution Modes
3. Threads
   - Single-Threaded vs Multithreaded Processes
   - Advantages of Threads

Overview II

- Thread States
- Implementing Threads
- Threads in User Space
- Threads in Kernel Space

POSIX

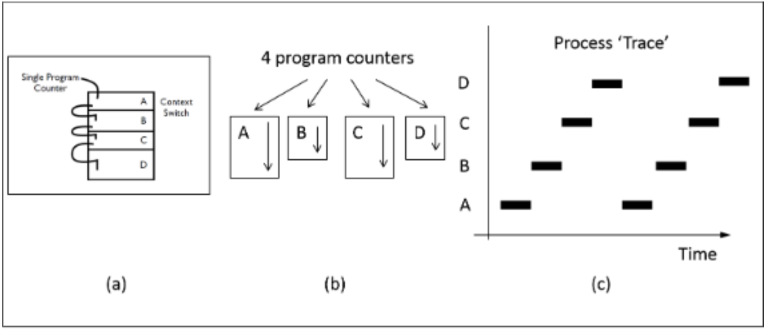
- For programs to run on any UNIX system IEEE developed a standard for UNIX called POSIX.
- Defines a minimal system cll interface comprising of about 100 procedure calls.
- Grouped into four categories:
  - Process management
  - File management
  - Directory and file system management
  - Miscellaneous.

Multiprogramming

- CPU switches from program to program running each for 10s of milliseconds.
- Creates an illusion of parallelism sometimes called pseudo-parallelism
- System designers have developed a conceptual model called sequential processes to keep track of multiple parallel activities.
- Most programs are sequential processes.
- They comprise of a series of instructions, executed sequentially.
- Assuming the input is unchanged they will always produce the same result.

Three views of Multi-programmings

- Computer multiprogramming four programs in memory.
- Four processes each with its own flow of control.
- Trace of ecevution shows all processes making progress.



Side Effects

- Because of CPU switching, the rate at which a process performs its computation will not be uniform and probably not even be reproducible if the same processes are run again.
- Processes must **NOT** be programmed with built in assumptions about timing.
- When a process has a particular critical real-time requirement meaning particular events must occur within a specified number of milliseconds, special measures must be taken.

Process vs Program

- Key idea is that a process is an activity of some kind.
- It has a:
  - Program
  - Input
  - State
  - Output
- A single processor may be shared amongst several processes.
- A process is a program in execution.
- A program is just a list of instructions in memory.

- OS need some way to create and terminate processes.
- Four principle events cause processes to be created:
  - System initialisation
  - Execution of a process creation system call (by another process)
  - A user types a command.
  - A user runs a batch job.
- In UNIX there is only one system call for creating a process: `fork`.
- `fork` creates an exact clone of the calling process. After `fork` there are two processes, the parent and the child, each having:
  - Their own distinct address space.
  - The same memory image.
  - The same environment settings
  - The same open files
- Usually the child process then executes `execve` to change its memory image and run a new program.

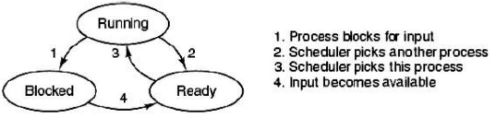
Process Termination

- Terminate due to:
  - Normal exit (voluntary).
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary).

Process Hierarchies

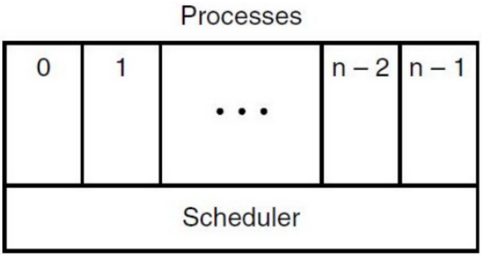
- Some systems, when a process creates another process, the parent process and child process cnotinue to be associated.
- In UNIX, a process and all its children form a process group e.g.:
  - When a user sends a signal from the keyboard it is sent to all members of the prcess group currently associated with the keyboard.
  - When UNIX is started, `init` is present in the boot image. When it starts running, it reads a file telling how many terminals there are and `forks` one new process per terminial. It waits for a user to log in and if login is successful the login process executes a shell to accept commands. Hence, in UNIX all the proceses in the whole system belong to a single tree, with `init` at the rot.
- `pstree` command shows the running processses as a tree.

Process States

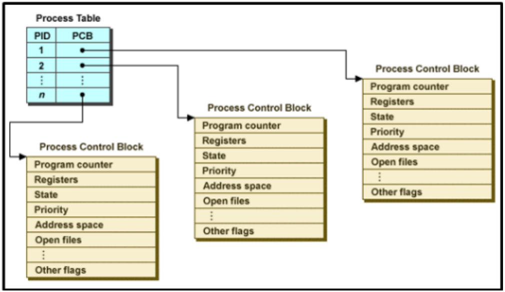


- Transition 1 occurs when the S discovers that the process cannot continue right now.
- Transition 2 & 3 are caused by the schedular (part of OS).
- Transition 4 occurs when the external event for which the process was waiting (such as arrival of input) happens.
- External events send electrical signals called interrupts to the schedular.

Process Model

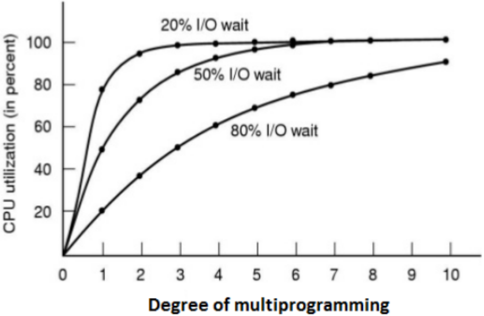


- Using process model, becomes much easier t think about what's going on inside the system.
- In the process model, the lowest level of the OS is the schedular with a variety of proesses on top of it.
- All of the details of interrupt handling and actually starting and stopping te processes is hidden away in the schedular.



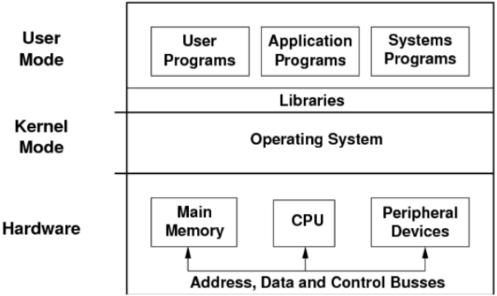
- The OS maintains a table called the process table.
- Individual entries in the tabel calles process control blocks hold important information about the process state:
  - Program counter
  - Stack pointer
  - Memory allocation
  - Open files
  - Other accounting and scheduling information

Modelling Multiprogramming



- Model CPU usage from a probabilistic viewpoint.
- Suppose that a process spends a fraction of its time p waiting for I/O to complete.
- Assum n processes in memory at once.
- Probability that all n process will be waiting for I/O is p^n,
- Hence, CPU Utilisation $= 1 - p^n$

Execution Modes



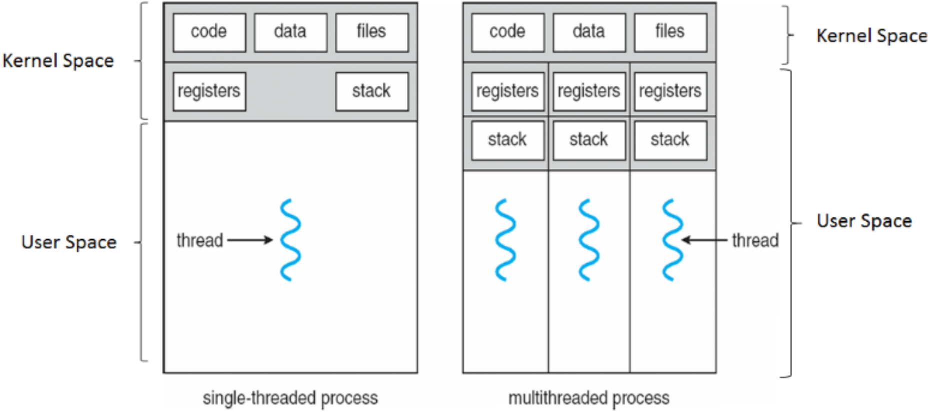
- Most computers have two mode of execution:
  - Kernel (or supervisor) Mode.
  - User Mode
- Processes running in kernel mode have unrestricted access to the machine:
  - The can disable all interrupts
  - Manipulate stacks
  - Unrestricted access to memory and I/O.

Threads

- In traditional OS, each process has an address space and a single thread of execution.
- It is sometimes desirable to have multiple threads of control in the same address space running in quasi-parrallel.
- Threads run as though they were (almost) seperate processes (excpet for the shared address space).
- A thread can be considered a lightweight process.

Single-Threaded vs Multithreaded Processes

Multithreaded partition of kernel / User space assumes threads are implemented in user space.



Advantages of Threads

- Decomposing the application into multiple threads that run in quasi-parallel the progrmming model becomes simpler. Also, threads share the same address space and data (unlike processes).
- Since threads are lighter weight than processes they are easier to create and destroy than processes. Creating a thread is typically 10-100 times quicker than creating a process.
- Threads are useful on systems with multiple CPUs.
- Threads yield no performance gain when all of them are CPU bound.
- Multiple threads are appropriate when they are part of the same job and actively cooperating with one another.

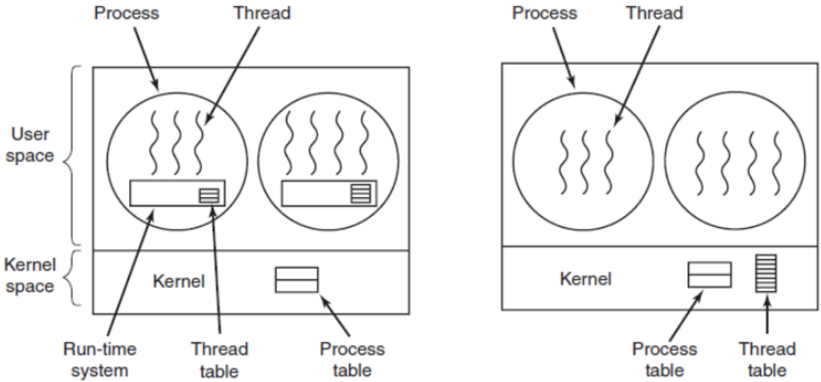
Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Thread States

- A thread can be in one of three states:
  - Running
  - Blocked
  - Ready
- Different ways it can be blocked:
  - When the process blocks, ALL threads block
  - Threads may block waiting for an event
  - Threads may enter a sleep state to be woken by a signaal from another thread.

Implementing Threads

Threads can be implemented in user space or kernel space



Threads in User Space

- Advantages:
  - As far as the kernel is concerned it is running one single threaded process.
  - Can be run on OS that doesn't support multithreading.
  - Thread switching can be made very fast.
  - Each process can have its own scheduling algorithm for threads.
- Disadvantages:
  - If a thread starts running, no other thread in that process will ever run unless the first thread gives up the CPU.
  - Blocking system calls are difficult to implement.
  - Progrmammers generally want threads precisely in applications where the threads block often, and its more efficient to implement as a process.

Threads in Kernel Space

- Advantages:
  - Kernel threads do not require any new, nonblocking system calls.
- Disadvantages:
  - The cost of creating and destroying threads in the kernel is substatial.
  - All calls that might block a thread are implemented as system calls at considerable greater cost than a call to a run-tme system procedure.
- Other issues:
  - What happens when the process `forks`?
  - How can we handle signals?

Summary

- Processes are Ready,Running or Blocked.
- Threads may also Sleep and Wait.
- Two modes of execution User & Kernel processes switch to kernel mode during system calls.
- Threads are lightweight processes.
- Two types of thread implementations:
  - Kernel threads: Managed by the OS
  - User threads: Managed by user.

The End