Overview

Definitions

How does OS execute processes?

- How do kernel & processes interact
- How does kernel switch between processes
- How do interrupts fit in

What's the difference between threads/processes

Process States

Priority Scheduling

Process

These are all possible definitions:

- A program in execution
- An instance of a program running on a computer
- Schedulable entity (*)
- Unit of resource ownership
- Unit of protection
- Execution sequence (*) + current state (*) + set of resources

(*) can be said of threads as well

Alternative definition

Thread:

- Execution sequence + CPU state (registers + stack)

Process:

- n Threads + Resources shared by them (specifically: accessible heap memory, global variables, file descriptors, etc.)

In most contemporary OS, n >= 1.

In Pintos, n=1: a process is a thread – as in traditional Unix.

- Following discussion applies to both threads & processes.

Context Switching

Multiprogramming: switch to another process if current process is (momentarily) blocked

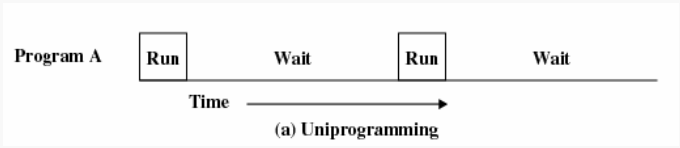
Time-sharing: switch to another process periodically to make sure all process make equal progress

- this switch is called a context switch.

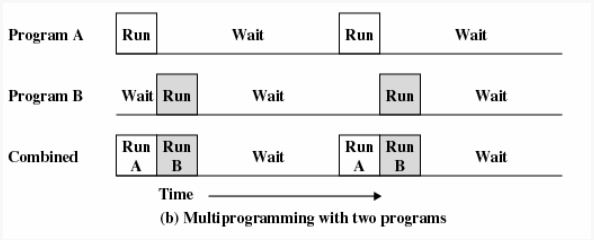
Must understand how it works

- how it interacts with user/kernel mode switching
- how it maintains the illusion of each process having the CPU to itself (process must not notice being switched in and out!)

Single Program vs Multiprogramming

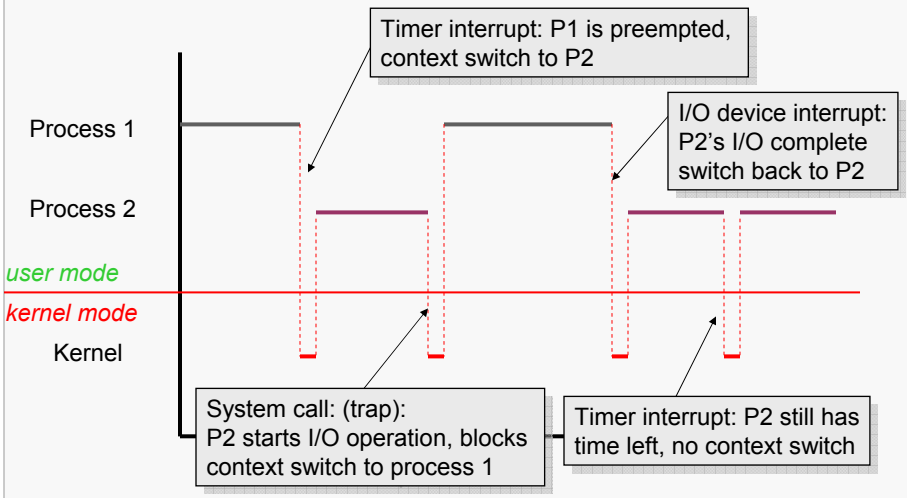


(a) Uniprogramming

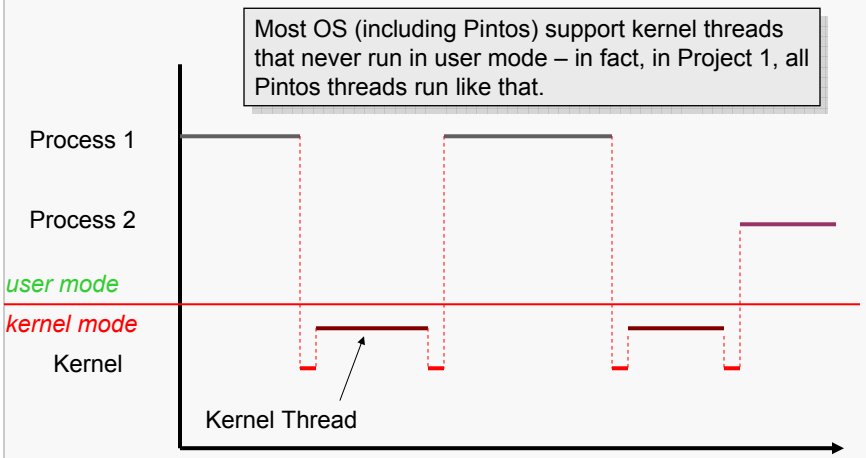


(b) Multiprogramming with two programs

Context Switching



Aside: Kernel Threads



Careful: "kernel thread" not the same as kernel-level thread (KLT) – more on KLT later

Mode Switching

User → Kernel mode
- For reasons external or internal to CPU

External (aka hardware) interrupt:
- timer/clock chip, I/O device, network card, keyboard, mouse
- asynchronous (with respect to the executing program)

Internal interrupt (aka software interrupt, trap, or exception)
- are synchronous
- can be intended: for system call (process wants to enter kernel to obtain services)
- or unintended (usually): fault/exception (division by zero, attempt to execute privileged instruction in user mode)

Kernel → User mode switch on iret instruction

Context vs Mode Switching

Mode switch guarantees kernel gains control when needed
- To react to external events
- To handle error situations
- Entry into kernel is controlled

Not all mode switches lead to context switches
- Kernel code's logic decides when – subject of scheduling

Mode switch always hardware supported
- Context switch (typically) not – this means many options for implementing it!

Protection

Multiprogramming requires isolation

OS must protect/isolate applications from each other, and OS from applications

This requirement is absolute
- In Pintos also: if one application crashes, kernel should not! Bulletproof.

Three techniques
- Preemption
- Interposition
- Privilege

Protection #1: Preemption

Resource can be given to process and access can be revoked

- Example: CPU, Memory, Printer, "abstract" resources: files, sockets

CPU Preemption using *interrupts*

- Hardware timer interrupt invokes OS, OS checks if current process should be preempted, done every 1ms in Linux
- Solves infinite loop problem!

Q.: Does it work with all resources equally?

Protection #2: Interposition

OS hides the hardware

Application have to go through OS to access resources

OS can interpose checks:

- Validity (Address Translation)
- Permission (Security Policy)
- Resource Constraints (Quotas)

Protection #3: Privilege

Two fundamental modes:

- "kernel mode" – privileged
  - aka system, supervisor or monitor mode
  - Intel calls its PL0, Privilege Level 0 on x86
- "user mode" – non-privileged
  - PL3 on x86

Bit in CPU – controls operation of CPU

- Protection operations can only be performed in kernel mode. Example: hlt
- Carefully control transitions between user & kernel mode

```
int main()
{
        asm("hlt");
}
```


OS as a Resource Manager

OS provides illusions, examples:

- every process is run on its own CPU
- every process has all the memory of the machine (and more)
- every process has its own I/O terminal

"Stretches" resources

- Possible because resource usage is bursty, typically

Increases utilization

Resource Management

Multiplexing increases complexity

Car Analogy (by Rosenblum):

- Dedicated road per car would be incredibly inefficient, so cars share freeway. Must manage this.
- (abstraction) different lanes per direction
- (synchronization) traffic lights
- (increase capacity) build more roads

More utilization creates contention

- (decrease demand) slow down
- (backoff/retry) use highway during off-peak hours
- (refuse service, quotas) force people into public transportation
- (system collapse) traffic jams

Resource Management

OS must decide who gets to use what resource

Approach 1: have admin (boss) tell it

Approach 2: have user tell it

- What if user lies? What if user doesn't know?

Approach 3: figure it out through feedback

- Problem: how to tell power users from resource hogs?

Goals for Resource Management

Fairness
- Assign resources equitably

Differential Responsiveness
- Cater to individual applications' needs

Efficiency
- Maximize throughput, minimize response time, support as many apps as you can

These goals are often conflicting.
- All about trade-offs