spin() ← checks time and returns after 1 second.

Virtualizing memory

many running programs → same memory address (virtualization). Takes physical resources such as CPU, memory or disk & virtualises them.

Concurrency

pthread_create() creates two threads, updating shared variables correctly

Persistence

preventing data losses due to system crash (file system)

syscalls: open(), write(), close().

systems use "journalling" (or) "copy-on-write" ordering writes to recover to to a reasonable state.

Goals of OS

- Provide high performance, i.e., minimize overheads of the OS, e.g. extra time, space
- Protection b/w applications and b/w OS & applications (principle: isolation)
- Provide high degree of reliability
- Energy efficiency, security, mobility

History of OS

- Batch processing: Number of jobs very set up & run by operator. Glorified libraries, OS didn't do much, low level I/O.
- System call introduced, file system as library, privilege levels
- trap instruction: system call initiated and transfers control to trap handler. & raises privilege level to kernel mode

  ↳ OS done? return-from-trap instruction.

- multiprogramming: OS loads number of jobs & switches rapidly b/w them
- memory protection, one program shouldn't hog up all memory.
- DOS (Disk Operating System, Microsoft): Memory protection issues
- Mac OS (v9): cooperative scheduling, a thread getting stuck could force sys to reboot

- Importance of UNIX:
- shell, pipes
- provided compiler for C programming language
- Bill Joy: Berkeley Systems Distr. (BSD) { small kernel written in C was modified easily }
- LINUX (Linus Torvalds), built from ideas of UNIX.

Process Abstraction

- time sharing : users can run many concurrent processes as they like (cost = performance)
- mechanisms = low-level machinery
- space sharing : resource (e.g. disk) is divided among users.

Process = machine state

1. Memory
   - Instructions lying in memory, data that program reads
   - address space : memory that process can address
   - Registers : state of execution
   - PC / IP, stack pointer, frame pointer

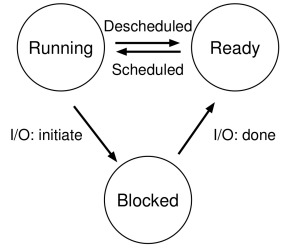
Process API

- create
- destroy
- wait
- miscellaneous control
- status

Process Creation :

1. "Load" code of program into memory (disk → memory)
   (modern OS don't load code all at same time, "lazy")
2. Allocate memory for runtime stack (local vars, fn parameters, return addr)
3. Create some initial memory for heap (dynamic alloc)
4. I/O initialization (e.g. file descriptors 0, 1, 2)

Process States :



1. Running : executing instructions on processor
2. Ready : ready to run
3. Blocked : Can't run until some event takes place (e.g. I/O)

Data Structures (of OS)

1. process list : list of processes that are ready, running, track blocked processes
2. register context : content of register state

* other states like initial, final (zombie)

* PCB : a C-structure which maintains information of each process

Process API

ref : lab notes

exec()
- loads code (and static data) from executable and over-writes current code segment.
- Separation of fork() & exec() lets UNIX shell run code after call to fork(), before exec().

Shell
- Just a user program, shows prompt and waits for input

UNIX pipes
- output of one process is connected to an in-kernel pipe (queue)

other parts of API :
- kill () used to send signals to process, including directives

LDE (Mechanism)

Challenges to virtualization via time sharing (of CPU)
1. Implement virtualization without adding excessive overhead
2. Run processes efficiently while retaining control
- Needs both hardware & OS support

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Table 6.1: **Direction Execution Protocol (Without Limits)**

Issues :
1. How can OS make sure program doesn't do anything that we don't want it doing
2. How does OS stop it from running and switch to another process, thus, implementing time sharing

Problem #1 (Restricted Operations)
- User mode : limited access, Kernel mode : complete access to resources
- Special instructions trap (into kernel), return-from-trap (back to user mode), instructions to tell hardware where trap table is. (programs use trap to invoke syscalls) OS uses return-from-trap
- syscalls for user mode provided.

How does trap know which part of OS code to run? {trap table @ bootup}

trap → trap-handler → OS.

phase 1

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of... syscall handler

phase 2

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

Table 6.2: **Limited Direction Execution Protocol**

Problem #2: Switching b/w processes

① Co-operative approach (wait for system calls)
- Transfer control to OS using syscalls (yield system call)
- If e.g. divide by zero, generates a trap.
- OS waits for syscall to regain control.

② Non-cooperative approach : OS takes control
- Timer interrupt (raise an interrupt every so milliseconds), pre-configured interrupt handler runs.
- OS starts timer during boot (privileged operation)
- Hardware must save enough of program at interrupt to save state for subsequent return-from-trap.

Saving and restoring context

Context switch : decision to switch b/w processes

1. Save a few register values for the currently executing process (onto kernel stack for eg.) {executes low-level-assembly}
2. Restore soon-to-be executing process from kernel stack.
3. Execute return-from-trap instruction

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B ...

Table 6.3: **Limited Direction Execution Protocol (Timer Interrupt)**

- What if another interrupt happens while handling one ?
  (wait for concurrency :))

1. Disable interrupts during interrupt handling (ensures no one will be delivered to CPU when interrupt is being handled)
2. Locking mechanisms to protect concurrent access to internal data structures

Note : reboots are a robust way to deal with hard to handle situations (i.e. restoring previous safe state)

Scheduling (Introduction)

Understanding high-level policies that OS-scheduler employs

Workload assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (i.e. they perform no I/O)
4. Run-time of each job is known

Scheduling metrics

Single metric : turnaround time ($T_{turnaround} = T_{completion} - T_{arrival}$)

(∵ All jobs arrive at same time , $= T_{completion}$)

Policies

1. FIFO/FCFS
   poor policy if huge first process (convoy effect)

2. Shortest Job First (SJF)
   - we can prove SJF is an optimal scheduling algorithm

Relax arrival at same time.

3. Shortest Time-to-Completion First (STCF)
   - optimal under turnaround metric.
   - But, need response ↦ new metric

$$T_{response} = T_{first\ run} - T_{arrival}$$

4. Round Robin
   - Good for response time (P scheduled for time slice)
   - Amortization : perform fixed cost operation fewer times
   - Pretty bad for turnaround metric (one of worst policies)

Incorporating I/O

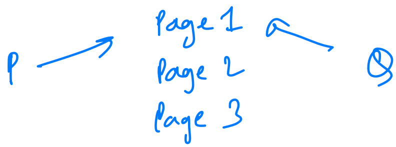
- Overlap enables higher utilization
  (when interactive processes perform I/O, other CPU-intensive jobs run)

we don't know running time ?

- multi-level feedback queue { TBD }

- **Note:**

1. exec() does not alter PCB's fd table
2. Trigger context switch irrespective of scheduling policy :
3. Interrupt Desc Table (IDT) stores address of handler.
4. Kernel / User switch gives o/p in eax register.
5. If fork(), child or parent runs completely, then the other.
6. Copy-on-write optimization (defers the copy until object is modified)



Page 3 modified