~~OSC AnalTerm~~

OSC Final Analysis

os是kernel，一段中间层一直运行的程序。{resource allocator / control program

Chapter 1 Introduction:

① interrupt (广义中断) {interrupt (石硬件中断) / trap (软件中断，如÷0/无效访存/用户请求) / exception (软件中出，出错/用户请求) — 重OS服务.

② mode ⇒ 用于区分用户/系统 {user mode / kernel mode / (Vmm mode)}
使用特权指令 privileged instruction
利用中断进行转换.

③ timer 定时器 ⇒ 修改指令是特权指令.
⇒ 防止死循环，一定时间产生中断，控制权给OS.

④ {multi programing - 多程序，内存中放多个任务，让CPU等待时有事做，始终忙碌.
Time sharing/multitasking - 多任务，CPU在多个任务间切换，都迅速响应.
multi-processor.

Chapter 2: OS Structure

① system call ⇒ 用户使用OS功能的方法 (用C/C++写的) (使用特权指令方法).
↳ API：用户使用API(接口)实现Syscall(可移植、细节隐藏) — 一般本身不是内核函数，但会调用内核函数实现功能.
使用后控制权为OS，OS实现对应功能.
System call interface：维护一个列表索引，根据syscall的数字来调用kernel中的syscall，并返回值. 有笔记中图
系统调用号.

② {策略 policy ⇒ 做什么 (Ex: 定时器时长.) ⇒ 可随时改变
机制 mechanism ⇒ 怎么做 (Ex: 定时器) ⇒ 用机制实现策略.
⇒ 区分的目的是灵活性 flexibility，易变.
(策略随通用最好.)

③ OS结构:
1) simple：UNIX / MS-DOS
未明显区分接口及功能，易出错 (没模块化)
2) layer：模块化，分层 (石硬件、OS层、用户层…)
3) micro kernel ⇒ 内核大不好管理. (留内存、进程、通信) (CPU调度)
⇒ 非基本部份抽走变为系统/用户程序.
4) module ⇒ 面向对象，调用其它模块来完成功能
类似micro kernel，但不用通信，更高效.
5) hybrid ⇒ 分层 + 微内核 Mac OS

Chapter 3 Process (可执行文件，不是code)

① ⇒ 被装入内存中执行的程序. ⇒ 静态 (动态)
组成 × 5 {code (text section) / activity → PC、register / stack → temp data / data section → global var / heap → dynamic memory

② state 看笔记图，画
{new / running / waiting / ready / terminate}
{short / mid / long}
进程 restore 必须 record machine state. → for context switch (time-sharing)
saving and restoring context

③ ~~PCB~~
context switch ⇒ 保存的 machine state 在PCB
{context → MEMORY / (time Interrupt → user REG → kernel stack / kernel REG → PCB
重新加载也是从PCB加载.

④ PCB (Process Control Block) 一个进程是一个
⇒ 进程在OS内的表示，见P73
⇒ 主要存了进程信息 (状态、编号、PC、REG …)

⑤ Scheduling of process
⇒ maximize CPU use {job → 所有进程 / device / ready}
⇒ 通过队列实现
{short-term ⇒ 选下一个执行的 (分配CPU). 频繁.
long-term ⇒ 选进入ready队列的. ⇒ {I/O-bound / CPU-bound} (multi program 程度不多页面.)
medium-term ⇒ 暂时将进程移出内存，进入disk，降低多程序运行程度，之后再恢复 (swap)}

⑥ Process API
1) pid ⇒ 每个进程唯一，管理时的依据. 子pid
2) fork() 创建新进程，复制当前的program，返回给父>0 / 子:0
3) exec() 用新程序取代进程空间并执行，常在fork()之后.
4) exit() / wait() / abort() ⇒ Parent 停止 child
全部结束，删除分配空间. ↳ Parent 等 child 直到结束.

看笔记例题改. ⇒ 变量会复制，父子不共享. 是进程，有内存保护

⑦ 进程通信. (for cooperating process). {direct {block / non-block} / indirect}
{~~shared memory~~ / message passing / ~~message passing~~ / shared memory}
消息队列
⇒ 适合多进程/简单/数据少 ⇒ 一组call
不用频繁冲突
⇒ 多进程 cache 不一致/快，无syscall
锁同步 删完消失 与kernel无关
访存.

⑧ pipe {ordinary - 单向，创建者发给receiver(父子) ⇒ 是一种直接通信方法
named - 所有进程均可访问 ⇒ 双向，谁都可用多个工具}
两个个通信. pipe() ⇒ syscall
用完保留

☆ 进程是OS资源分配最小单位
线程是OS调度最小单位

Chapter 4 Thread ⇒ CPU使用基本单元 ⇒ 响应快 经济(共享) 利用多处理器

⇒ 看笔记图 {共享: 代码、数据、文件 ⇒ 与Process区别 ⇒ 每个线程唯一, 与栈不同
私有: REG、栈、TLS ⇒ 线程存可见 不是局部, 更像静态var (存local var)

concurrency & Parallelism

① 多线程模型

⇒ 描述用户线程与 kernel thread 关系

{多对一: 同时只有一个thread访问kernel, 用不好multiprocessor, 一个阻塞则OS阻塞. 内核管理 efficient
一对一: 创建开销大, 并发最好
多对多: 多路复用, 一个阻塞可调度另个
二级: 有一对一, 有多对多. two-level

② Thread lib ⇒ Pthread API

{~~Runner()~~ pthread_create() 创建单独线程.
pthread_join() ⇒ 父等待子完成. wait()
pthread_exit() 终止.

③ 线程池 ⇒ 存放一定数量等待执行的线程, 空闲可用, 有请求时从里面取出并执行

优点 {比创建一个新的更快.
限制了可用thread数量(不耗过多资源) 限制并发thread数量
更多可用策略, 执行与创建分开.

④ Issues:

1) Thread Cancellation {异步 asynchronus: 直接
延迟 deferred: 不断检查是否该cancel 可自行cancel

2) Signal ⇒ 通知一个进程某事件发生 如ctrl+C
⇓
{default
user-defined
handle(处理程序) ⇒ OS执行

3) LWP ⇒ virtual processor 连着 kernel thread
u-t和k-t之间映射OS
⇓ 给user提供内核服务
schedule user thread 接口.
不会upcall, 不会直接换user thread
由kernel thread来
⇒ 开始时一个kt对一个LWP
block之后调新的LWP, 最effient是一个ut一个LWP
会阻塞, I/O就绪, ut重分给第一个.

Chapter 5 CPU Scheduling (I/O burst / CPU burst)

① Concept {CPU scheduler: schedule
Dispatcher: context switch / switch user mode / restart user program
调度程序. ↳ 将控制给到schedule的process.

② Timing {① running to waiting (ready → running)
② running to ready
③ waiting to ready
④ Terminate
⇒ nonpreemptive 只0、④
否则 preemptive

③ Scheduling Criteria ⇓ 评判 {CPU utilization ⇒ busy
Throughput ⇒ complete exercution per → 单位时间工作量. 完成几个P
Turnaround Time ⇒ T_complete - T_arrival
waiting Time ⇒ time in ready queue!
Response time ⇒ request to response time (not output)
就arrive

④ Algorithms ⇒ 要能画 Gantt Chart ↓ 1 2 3 4 5 → time, A → process

1) FCFS
⇒ 平均waiting time 长
⇒ convoy effect: 所有其它进程一起等一个大process释放CPU

2) SJF ⇒ 实际是最短next CPU burst ⇒ 难在不知道长度.
⇒ 平均 wait time 最小
{抢占 ⇒ shortest-remaining-time-first
非抢占 ⇒ 等当前P完事 比当前P余量和新P.

3) priority-scheduling
{抢占
非抢占
会导致 starvation 低priority一直等 ⇒ 用aging解决, 随时间提高等待的进程的优先级. 权成1最

4) RR Round-Robin (抢占式)
⇒ CPU循环执行就绪队列, 每次一个time slice, 用FIFO队列
⇒ 未完成的加入队列尾部, 已完成的直接切. ⇒ 要考虑切换上下文时间!
⇒ 平均wait time 长, response 短. 也算wait time!

5) MLQ
⇒ 分成 {foreground P (RR) 一般会5级以上,
background P (FCFS) 高优先抢占低优先
⇒ 本质是加了队列优先级, 不同队列重要性、调度不同.
⇒ 队列是永久的 与MLFQ区别

6) MLFQ. ⇒ 定义可能不同, 多小心
(1) 新到任务先给最高优先 都用FIFO队列
(2) 高优先必先行. 抢占式; 同优先RR. 细心!
(3) 越低级time slice越长
(4) 定期将所有任务升到top (防starvation).
(5) 未完成就降级, 到下一级队列尾部

⑤ thread schedule ⇒ 区分user & kernel.
⇒ OS只管kernel, user要 → LWP → kernel.
一对一 只有这个 {SCS: System-Contention Scope ⇒ kernel thread抢CPU
PCS: Process-Contention Scope ⇒ process抢thread lib (LWP)

⑥ Multi-Processor schedule
{asymmetric ⇒ 主处理器调度, 其它执行
symmetric ⇒ 每个处理器自我schedule ⇒ 控制复杂, 只有一个ready queue给全局可能私有

processor affinity 喜如性.
⇒ SMP系统上进程运行在同一个处理器上
避免cache反复访存 ↳ 处理器间不共享, 要重来, 命们新.

{soft → 尽力
hard → 必定

HRRN ⇒ schedule Alg.
⇒ Highest response ratio next
$rr = 1 + \frac{waiting}{run}$ 收益率.
优调rr高的.