Intro

- Monolitic OS
  - + Good performance, well understood
  - - Coupled, very complex internal structure
- Microkernel OS
  - Very small, provides basic functionalities
  - - More robust and extendible, better isolation and protection between kernel and services
  - - lower performacne
- Type 1 Hypervisor - Bare Metal
  - Hardware –> Hypervisor –> OS1, OS2, OS3 (VMWare ESXi)
- Type 2 Hypervisor - Host OS
  - Hardware –> OS –> Hypervisor –> OS1, OS2

Process Abstraction

Every process has PID

States

1. New
2. Ready
3. Running
4. Blocked
5. Terminated

Transitions

1. Create (nil -> New)
2. Admit (New -> Ready)
3. Switch (Ready -> Running)
4. Switch (Running -> Ready)
5. Event Wait (Running -> Blocked)
6. Event occurs (Blocked -> Ready)

Syscall Mechanism

1. User prog invokes library call
2. Library call places syscall No in register
3. Library call exec TRAP to switch to kernel
4. Handler determined by dispatcher using syscall no
5. Syscall executed
6. Syscall ended, restore CPU and return to library, switch from kernel to user mode
7. Library call return to user program

Exceptions are synchronous and have to execute exception handler. Arithmetic Error, Memory Access Error

Interrupts are async external event that can interrupt exec of program, and have to execute interrupt handler

PA in Unix

`fork()` is main way to create new process, returns PID of new process (parent) or 0 for child process.
Differ in `PID, PPID, fork()` return value
`execl(const char *path, const char *arg0..n, NULL)` replaces currently executing process with new one
`fork() + exec()` is the main way of new process for new program
`init` process, PID=1 watches for other processes and respawns when needed, `fork()` creates process tree, init is the root process
`exit()` to end execution of process, status returned to parent process, 0 for normal, !0 for others. On exit, most resources released, some not released (PID, status, CPU time). Return from main implicitly calls exit
`wait(int *status)` blocks and cleans up remainder of child resources. Returns PID of terminated child process. Stores exit status of terminated child process.
zombie when child finishes but parent does not wait()
orphan when parent terminates before child. `init` becomes parent and handles cleanup

Fork Impl

1. Create address space of child
2. Allocate `p' = new PID`
3. Create entry in process table
4. Copy kernel environment of parent process (Priority for scheduling)
5. initialize child process ctx, `PID=p', PPID=parent`
6. Copy mem region from parent (prog, data, stack)
7. Acquire shared resources (files, CWD, etc)
8. Init hardware CTX (copy registers)
9. Add to scheduler queue

Process Scheduling

Concurrent Execution multiple processes progress in execution at same time. This can be **Virual Parallelism** or **Physical Parallelism**
A typical process goes through phases of **CPU Activity** (Computation) and **IO Activity** (Disk, print to screen)

Criteria for Scheduling

- **Fairness** - Fair share of CPU time, and no starvation
- **Utilization** - All parts of computer utilized

When to Perform Scheduling

- **Non-Preemptive** (Cooperative)- Process stays scheduled until blocks / gives up CPU
- **Preemptive** - Given fix time quota to run, at the end, process suspended

Batch Processing

Criteria

- **Turnaround time** Total time taken (finish - arrival)
- **Waiting time** Turnaround - work done
- **Throughput** - Number of tasks finished per unit time
- **CPU Utilization** - % of time when CPU working

Algorithms

First Come First Served - FIFO queue based on arrival time. First task in queue until task blocked or done.
- **No Starvation** No of tasks infront of task X in FIFO is always decreasing
- **Convoy Effect** Long running CPU bound task A followed by IO bound tasks X. While A running, X is waiting in ready queue, once A is blocked on IO, X will execute and go to IO queue, CPU is idling

Shortest Job First - Select Task with shortest CPU time. Need to know total CPU time for task.
- Minimizes average waiting time
- Starvation possible (Biased towards short jobs, long jobs might never get a chance)
- Predicting CPU time
  $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + 1(1-\alpha)\text{Predicted}_n$

Shortest Remaining Time - Select job with shortest remaining / expected time. New jobs with shorter time can preempt currently running job

Interactive

Criteria

- **Response Time** - Time between request and response
- **Predictability** - Less variation == more predictable

Timer Mechanism

Timer Interrupt goes of periodically, calling the scheduler
Interval of Timer Interrupt(ITI) Scheduler invoked on every interrupt, ranging from 1ms to 10ms
Time Quantum Duration given to process. Multiple of timer interrupt, 5ms to 100ms

Algorithms

Round Robin - FIFO queue, where first taslk run until time quantum elapsed / gives up CPU / blocks. Placed at and of queue
- **Response Time Guarentee** - nth task will run by $(n-1)q$
- **Choice of Quantum** - Big quantum (better utilization, longer wait time), Smaller Quantum (worse overhead, but shorter wait time)

Priority Scheduling - Task with highest priority will be executed first. Preemptive (Higher priority task can preempt lower priority), or Non-preemptive (late coming high priority process has to wait for next round of scheduling)
- **Starvation** Low priority tasks can starve. Solution:
  - Decrease priority of current running process after every quantum
  - Process not considered in next round of scheduling
  - Lower priority task can lock a resource higher priority needs. Lower preempts higher priority task

Multi-level Feedback Queue - Designed to solve issue of scheduling without knowledge.
- **Adaptive** - Learns process behaviour automatically
- **Minimises** response time for IO bound and turnaround time for CPU bound
- **Rules**
  - If Priority(A) > Priority B -> A runs
  - if Priority(A) == Priority(B) -> A and B run in RR
  - New Job -> Highest Priority
  - On fully using time quantum -> Priority reduced
  - If gives up / blocks before time quantum -> Priority Maintained

Lottery Scheduling - Give lottery ticket to processes for system resources. When scheduling needed, lottery ticket randomly chosen, winner granted resource
- **Responsive** - New process can participate
- **Good level of control** - Process can be given Y tickets, important process can have more tickets, and each resource can have own set of tickets

Threads

Process is expensive as it Duplicates memory, process context, and context switching requires saving and restoring process info. Hard for multiple process to communicate due to independent memory space. Add threads so multiple parts of same program execute concurrently

Thread Model

- Single process can have multiple threads
- Threads share Memory Context (Text, Data, Heap)
- Threads share OS Context (PID, files, etc)
- Each thread needs ThreadID, Register, Stack

Context Switching - Thread switch only requires changing hardware ctx (register, FP, SP), whereas Process Switch requires (OS, Hardware, Memory CTX)

Benefits
- **Economy** - Multiple threads in same process requires less resource than multiple processes
- **Resource Sharing** - Share most resources of process, no need for additional mechanism for passing info
- **Responsive** - Multithreaded can appear responsive
- **Scalable** - Can take advantage of multiple CPU

Problems
- **System Call Concurrency** - Parallel exec of multiple threads -> parallel sys calls -> Guarentee correctness and determine correct behavior
- **Process Behaviour**
  - fork() called in thread, only 1 thread is cloned
  - If thread calls exit(), all threads exit
  - If thread calls exec(), all threads exit and new executable runs

User Thread

Implemented as user library, runtime system in process will handle thread related operations. Kernel not aware of threads

Advantages
- Multithreaded program on ANY OS
- Thread operations are library calls
- More flexible and configurable

Disadvantages
- OS not aware of threads, scheduling done at process level
- One thread blocked –> Process blocked –> All threads blocked
- Cannot use multi-CPU

Kernel Thread

Thread implemented in OS, operations handled as syscalls. Kernel can schedule by threads instead of process **Advantages**
- Can schdule on thread level - More than 1 thread in same process can run on multiple CPUs

Disadvantages
- Thread ops now syscall - Slower and more resources
- Less flexible (If impl with many features, overkill), (If impl with less features, not flexible enough)

Hybrid Thread

Both Kernel and User thread. OS Schedule Kernel thread, user thread binds to kernel thread

Posix Theads: pthread

`pthread_t` : TID, `pthread_attr` : attribute of thread
`pthread_create`
`pthread_exit`
`pthread_join`

IPC

Hard for cooperating processes to share information as memory space is independent, and IPC mechanisms is needed

Shared Memory

1. Process P_1 creates shared memory M
2. Process P_1 and P_2 attach M to its own memory space
3. P_1 and P_2 communicate using M

Advantages

- **Efficient** - Only initial step require OS
- **Ease of Use** - Behaves similar to normal memory space, information of any type / size can be written

Disadvantages

- **Synchronization** - Need to synchronize access
- Implementation is harder

Message Passing

1. Process P_1 prepares message M and sends it to P_2
2. Process P_2 receives message
3. Send / receive provided as syscalls

Direct Communication

- Sender / Receiver explicitly name other party (Domain Socket)
- 1 link per pair of communicating process
- Need to know identity of other party

Indirect Communication

- Sender / Receiver send to mailbox / port. (Message Queue)
- 1 mailbox shared among processes

Synchronization Behaviours

- **Blocking Primitives** - Receive() blocked until message
- **Non-Blocking primitives** - Receive() will receive message / indicate message not ready

Advantages

- **Portable** - Easily impl on diff processing env (WAN, distributed systems)
- **Easier sync** - Sender and receiver implicitly synchronized

Disadvantages

- **Inefficient** - OS intervention, extra copying

Pipe

Process has 3 different comms channels, `stdin, stdout, stderr`

Process P writes n bytes, Q consumes m bytes. FIFO, must access data in order

Byte buffer with implicit synchronization, Writers wait when full, Readers wait when empty

Signal

Asynchronous notification regarding event sent to process/thread. Must be handled by default set of handlers / user supplied. Common signals: Kill, Interrupt, Stop, etc.

Synchronization

Race Condition

Problems

- When 2 or more processes execute concurrently and share modifiable resource -> Can cause sync problems
- Execution of single sync process is deterministic
- Execution concurrent processes non-deterministic (order in which resource is accessed / modified)

Solution: Designate code segment with race condition as critical section. At any point in time, only 1 process can execute in CS. Other processes prevented from entering CS.

Critical Section

Properties

- **Mutual Exclusion** - P_1 in CS, all other processes prevented
- **Progress** - No process in CS, one waiting process given access
- **Bounded Wait** - After P_1 requests to enter CS, upper bound on other process in CS before P_1
- **Independence** - Not executing in CS should never block other process

Incorrect Synchronization

- **Deadlock** - All processes blocked -> No progress
- **Livelock** - Deadlock avoidance mechanism -> Process keeps changing state to avoid deadlock and makes no progress
- **Starvation** - never makes progress in execution as its perpetually denied resources

Implementation of CS

Test and Set

Common machine instruction to aid synchronization

Behavior

1. Load current content at MemLoc to register
2. Store 1 into MemLoc

This is operated as single operation

```
EnterCS(int *Lock) { while TestAndSet(Lock) == 1}
ExitCS(int *Lock) { *Lock = 0}
```

EnterCS will set the value of the lock to 1 and return its previous value. If teh value is 1, that means that its still held by another process. If the value is 0, then the lock has been released, and this process can run. The value is set to 1 imediately. This detects the "change" in Lock.

Observation and Comments

Implementation works, however spinlock wastes resources (processing power)

Random Notes

To abuse MLFQ, you can spawn a child process and do work there.

Stack Frame setup / teardown

1. On exec func call
   1.1. Caller: Pass args with register and/or stack
   1.2. Caller: Save Return PC on Stack
   1.3. Callee: Save registers used by callee (Old FP, SP)
   1.4. Callee: Alloc space for local vars on stack
   1.5. Callee: Adjust SP to point to new stack, adjust FP
2. On Return from func call
   2.1. Callee: Restore saved registers, FP, SP
   2.2. Continue execution in caller