1 Intro to OS

Use of OS: **Abstraction** over hardware: no need to care about low level details. **Resource allocator**: allow programs to execute simultaneously. **Control program**: prevent errors and enforce security.

OS Structures: *Monolithic* (kernel is one big program) - Unix, *Microkernel* (very small, provides only basic functions - IPC, scheduling, interrupt handler. File system / Process management handled etc. by user), *Layered System* (generalization of monolithic system), *Client-Server model* (variation of microkernel)

Virtual Machines (hypervisor): Software emulation (virtualization) of hardware. Allows running several OS on the same hardware. Type 1: bare metal, direct access to hardware. Type 2: app installed on host OS.

2 Process Abstraction

Memory Context: Text, Data, Stack, Heap. **Hardware Context**: GPR, PC, SP, FP. **OS Context**: Process ID, Process State, PCB, Process Table

Function Call Convention: Different ways to set up/tear down a stack frame (Caller/Callee). There is no universal convention.

Stack Frame: Local variables, Parameters, Saved Registers, Saved SP, Saved FP, Return PC. **Stack Pointer**: Points to top of the stack. **Frame Pointer**: Points to a fixed location in a stack frame. The usage of FP is platform dependent.

Saved Registers: When GPRs are all used up, use memory to hold their values, restore them afterwards. A.k.a **register spilling**

Process States: New, Ready, Running, Blocked, Terminated

State Transitions: *Create* (nil → New), *Admit* (New → Ready), *Switch* (Ready → Running / Running → Ready (preempted/give up voluntarily)), *Wait* (Running → Blocked), *Event occur* (Blocked → Ready)

Process Control Block (PCB): A.k.a process table entry (PTE). The entire execution context for a process. Kernel maintains PCB for all processes

Process Table: A table for storing PCBs

System Calls: switch from user to kernel mode. In C/C++, library version with the *same name* acts as a **function wrapper**. User friendly library version acts as a **function adapter**. General mechanism: user makes library call → Place syscall number in a register → Execute **TRAP** (switch to kernel mode) → Find syscall handler using syscall number → Execute handler → Return control to library call, switch back to user mode

Exceptions: E.g. Divide by 0, overflow, Illegal memory access. **Synchronous** (due to program execution). An exception handler is called; similar to forced function call.

Interrupts: hardware related, e.g. timer, keyboard pressed (**asynchronous)**. Program is suspended. **Interrupt** handler is invoked

unistd.h: fork, execl. **stdlib.h**: exit. **sys/types.h, sys/wait.h**: wait

`int fork()`: Returns PID of newly created process (for parent) or 0 (for child). Duplicates parent.

`int execl(const char *path, const char *arg0, ..., NULL)`: **Replaces** current process. `NULL` used to indicate end of arg list.

Master Process: `init` (root), PID = 1, created by kernel at boot time.

`void exit(int status)`: Status returned to parent. 0 for success, !0 for error. The function **does not return**. Most resources are released e.g. FD, but PID/status/CPU time is needed. `exit` is implicitly called.

`int wait(int *status)`: Returns PID of terminated child. `status` is exit status of child. Use `NULL` if don't need. Blocking. This frees everything not cleared by `exit`. Kills zombies. Variants: `waitpid`, `waitid`

Zombie Process: On process exit, child becomes a zombie. If parent terminates before child: `init` becomes pseudo parent. If child terminates before parent but no `wait`: child becomes zombie, fills up process table

3 Process Scheduling

Scheduling: If n(ready to run process) > n(CPU), which should be chosen? Multiple ways to allocate, influenced by *process environment* and scheduling algorithm

Types of Process Environment: 1. *Batch Processing*: No user interaction, no need to be responsive, 2. *Interactive*: User interacts with system, should be responsive and consistent in response time, 3. *Real Time Processing*: Have deadline to meet, usually periodic

Criteria for Scheduling Algorithms: 1. *Fairness*: All processes/users should get a fair share of CPU time and have no starvation, 2. *Balance*: All parts of the system should be utilized

Types of Scheduling Policies: 1. *Non-preemptive* (Cooperative): A process stays scheduled until it blocks OR gives up CPU voluntarily, 2. *Preemptive*: A process is given a fixed time quota to run; at the end of the quota, another process is picked

Batch Processing: Criteria: *Turnaround time* (total time taken) i.e. Exit - Arrival. Related to **waiting time** (waiting for CPU). *Throughput*: No. of tasks finished per unit time. *CPU utilization*: Percentage of time CPU is working on a task. *Burst Time*: time in execution

FCFS: Executed in the order processes arrive; Guaranteed to have no starvation. Downside: *Convoy effect* - whole system slows down due to a few slow processes. E.g. CPU bound task blocks I/O tasks. Performs best when jobs arrive in order of shortest to longest CPU time.

SJF: Select task with smallest total CPU time - needs to know CPU time in advance. Can be guessed through previous CPU bound phases. **Exponential Average**: $\texttt{Predicted}_{n+1} = \alpha\texttt{Actual}_n + (1 - \alpha)\texttt{Predicted}_n$

SRT: A *preemptive* variation of SJF; select job with shortest remaining/expected time. New job with shorter remaining time can preempt current job. Good for short jobs that arrive late.

Criteria for Interactive Systems: 1. *Response time*: time between request and response (start execute - arrive time), 2. *Predictability*: Variation in response time; less variation = more predictable. Preemptive algorithms are used to ensure good response time → scheduler runs periodically. **Timer Interrupt**: goes off periodically based on clock. Cannot be intercepted by any other program. Interrupt handler invokes scheduler. **Time Quantum**: Execution duration given to a process. Can be constant or variable. *Must* be multiples of interrupt interval.

Round Robin: Preemptive version of FCFS. Response time guarantee: Given n tasks and quantum q, wait time is bounded by $(n-1)q$. Timer interrupt needed. The choice of quantum is important - **Big quantum**: better CPU utilization, longer wait time, behave more like FCFS, longer response time. **Small quantum**: bigger overhead (worse CPU utilization), shorter wait time. Performs same as FCFS when length of jobs are all the same and time quantum is never hit. Performs poorly for long jobs when there are many context switches (larger T/A time).

Priority Scheduling: *Preemptive version*: Higher priority process can preempt running process with lower priority. *Non-preemptive version*: Late coming high priority process has to wait for next round of scheduling. **Cons**: low priority process can starve, high priority process hogs CPU. Solutions: decrease priority over time/fix a time quantum.

MLFQ: If Prio(A) > Prio(B), A runs. If Prio(A) == Prio(B), A and B run in RR. New job → highest prio. If a job uses its the full quantum → prio reduced. Else, prio retained. Processes that game the system: calculated the accumulated CPU time used and reduce priority if CPU usage exceeds time quantum. Periodically bump up priority so a process doesn't get stuck in the lowest priority.

Lottery Scheduling: Each process has X

tickets, wins X% of the time. **Responsive**: newly created process participates in the next lottery. **Good level of control**: parent can distribute tickets to its children. An important process can be given more tickets, proportional to their usage. Each *resource* can have its own set of tickets.

4 Process Alternative: Threads

Motivation: Processes are expensive: memory duplication, context switch overhead, IPC is hard

Unique thread info: thread id, registers (GPR and special), stack

Benefits of Threads: 1. *Economy*: less resources required to manage threads compared to processes , 2. *Resource sharing*: thread share resources, no need to duplicate or have additional mechanism to pass information around , 3: *Responsiveness*: program appears more responsive, 4. *Scalability*: take advantage of multiple CPUs

Problems of Threads: Parallel execution of threads = parallel system call leading to synchronization issues

User Thread: Implemented by a user library. Kernel is not aware of threads in the process. **Pros**: Can have multithreaded program on **any** OS; thread operations are just library calls; More flexible and configurable e.g. custom thread scheduling policy. **Cons**: OS not aware of threads, scheduling is done at process level. One thread blocked → process blocked → all threads blocked. Cannot exploit multiple CPUs.

Kernel Thread: Thread is implemented in the OS; thread operations handled via **system calls**. Thread-level scheduling is possible; kernel schedules by threads instead of processes. Kernel may make use of threads for its own execution. **Pros**: Kernel scheduling - more than 1 thread of the same process can run on multiple CPUs. **Cons**: Thread operations is now a system call; slower. Generally less flexible, used by all multithreaded programs. If implemented with many features: expensive, overkill. Too little features: not flexible enough for some programs

Hybrid Model: Have both user + kernel threads. OS schedule on kernel threads only; user thread **binds** to a kernel thread.

POSIX Threads (pthread): `#include <pthread.h>`. Compilation: `gcc XX.c -lpthread`. Datatypes: `pthread_t` (thread ID type), `pthread_attr` (thread attributes type)

pthread Creation: `int pthread_create( pthread_t *tidCreated, const pthread_attr_t *threadAttributes, void* (*startRoutine) (void*), void *argForStartRoutine)`
Returns 0 for success, !0 for errors. `tidCreated`: thread id for created thread, `threadAttributes`: control behaviour of the new thread, `startRoutine`: function pointer to the function to be executed, `argForStartRoutine`: arguments to the start routine function

pthread Termination:
`void pthread_exit(void* exitValue)`
`exitValue`: value to be returned to whoever synchronize with this thread. If `pthread_exit()` is not used, a pthread will automatically terminate when the end of `startRoutine` is reached. In this case, there is no way to return an exit value.

pthread Synchronization:
`int pthread_join(pthread_t threadID, void ** status)`
Returns 0 for success, !0 for errors. `threadID`: the thread to wait for. `status`: Exit value returned by thread.

Resources shared: 1. Code (text) segment, 2. Global (static) data, 3. Heap memory, 4. Open files (file descriptors), 5. Current Working Directory, 6. Signal handlers, 7. Process address space

Thread switch: Registers (just FP and SP pointers), stack (for function calls and local variables). Much "lighter" than process.

5 Inter-Process Communication (IPC)

Motivation: Memory space is independent, hard to share information across processes

Common Mechanisms: 1. *Shared Memory*, 2. *Message Passing*. **Unix-specific Mechanisms**: 1. *Pipe*, 2. *Signal*

Shared Memory: Lifecycle: shmget → shmat → shmdt → shmctl. Pros: efficient, easy to use. Cons: synchronization, harder to implement

Message Passing: P1 sends message to P2, P2 receives message. Send/receive usually provided as system call. Additional Properties: *Naming* (how to identify the other party), *Synchronization* (sync/async send/receive). Messages are stored in kernel memory space.

Direct Communication: Explicitly name the other party (sender/receiver) e.g. `Send(P2, Msg)` - send to P2, `Receive(P1, Msg)` - receive from P1. There is only **one** link between sender/receiver

Indirect Communication: Send to mailbox/port. `Send(MB, Msg)` - send to MB, `Receive(MB, Msg)` - receive from MB. One mailbox can be shared among **several** processes. Pros: portable, easier synchronization. Cons: inefficient, hard to use

Types of message passing: 1. *Blocking* primitives (synchronous, wait until send/receive), 2. *Non-blocking* primitives, (asynchronous, fire and forget).

Unix Pipes: stdin/stdout/stderr. `|` is known as **piping**. General idea: create a communication channel with 2 ends, pipe the results from one process to another (Producer/Consumer relationship). Pipes function as **circular bounded byte buffer** with *implicit synchronization* - Writers wait when buffer is full, readers wait when buffer is empty. Variants: multiple readers/writers, half-duplex (unidirectional), full-duplex (bidirectional). Import from `#include <unistd.h>`. Syntax: `int pipe(int fd[])`. Returns 0 for success, !0 for error. An **array** of file descriptors is returned. `fd[0]` for reading end (read from here), `fd[1]` for writing end (write to here)

Unix Signals: e.g. `SIGINT`, `SIGSEGV`, `SIG_ERR`. An asynchronous notification regarding an event, sent to a process/thread. The recipient of the signal must handle the signal by 1. a default set of handlers, or 2. user supplied handler. Common signals: Kill, Stop, Continue, memory error, Arithmetic error etc.

6 Synchronization

Race Condition: Result of executing concurrent programs is non-deterministic, based on the order in which resources are accessed.

Solution to RC: *Critical Section* (CS) – at any point in time, only **one** process can execute in the CS while others are prevented from entering.

Properties of Correct CS Implementation: (1) *Mutual Exclusion*: one process in CS at any time, rest cannot enter, (2) *Progress*: if no process is in CS, one waiting processes should be granted access, (3) *Bounded Wait*: upper bound on wait time, will eventually enter CS, (4) *Independence*: processes not in CS should not block other processes

Symptoms of Incorrect CS Impl: (1) *Deadlock*: All processes blocked, no progress, (2) *Livelock*: Processes keep changing state to avoid deadlock but make no progress; usually related to deadlock avoidance mechanism, (3) *Starvation*: Some processes are blocked forever

Levels of CS Impl: 1. Assembly level, 2. High Level Language (HLL), 3. High Level Abstraction

Assembly Level: `TestAndSet Register, MemoryLocation`: Loads the current content in `MemoryLocation` into `Register` and stores 1 into `MemoryLocation`. This is a single machine instruction i.e. atomic. To enter CS: continue checking while `TestAndSet(Lock)` returns 1. Before exiting: set `*Lock = 0`. Pros: it is a correct CS impl. Cons: **busy waiting**, wasteful of processing power

HLL Impl (Peterson's Algorithm): `want[n]` array and `turn` variable. Assumption: writing to `turn` is atomic. Cons: busy waiting, too low level, not general enough