1 Introduction to OS

Motivation for OS: Manage resources and coordination (process sync, resource sharing), Simplify programming (abstraction of hardware, convenient services), Enforce usage policies, Security and protection, User program portability: across different hardware, Efficiency: Sophisticated implementations optimised for particular usage and hardware.

1.1 OS Structures

1.1.1 Monolithic

- Kernel is one BIG special program, various services and components are integral part
- Good SE principles with modularisation, separation of interfaces and implementation
- **Advantages**: Well understood, Good performance
- **Disadvantages**: Highly coupled components, Usually devolved into very complicated internal structure

1.1.2 Microkernel

- Kernel is very small & clean, only provides basic and essential facilities: IPC, address space & thread management, etc.
- **Higher level services** built on top of the basic facilities, run as server process outside of the OS, using IPC to communicate
- **Advantages**: Kernel is generally more robust & extensible, better isolation & protection between kernel & high level services.
- **Disadvantages**: Lower performance

1.2 Virtual Machine also known as Hypervisor

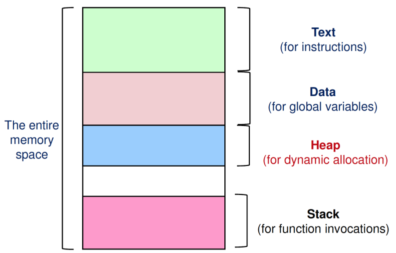
A software emulation of hardware – **virtualisation** of underlying hardware (illusion of complete hardware).

- **Type 1 Hypervisor**:
  Provides individual VMs to guest OS's (e.g. IBM VM/370)
- **Type 2 Hypervisor**:
  Runs in host OS, guest OS runs inside VM (e.g. VMware)

2 Process Abstraction

2.1 Process Abstraction

- **Process** = a dynamic abstraction for executing program
- Information required to describe a running program (Memory context, hardware context, OS context)
- An executable binary consists of two major components: instructions and data
- During execution, more information:
  - **Memory context**: text, data, stack, heap
  - **Hardware context**: General Purpose Registers, Program Counter, Stack Pointer, Stack FP, ...
  - **OS context**: PID, Process state, ...



Note: stack grows upwards in this case

2.2 Stack Memory

- New memory region to store information of a function invocation
- Described by a **stack frame**, containing: Return address of the caller (PC, old SP), Arguments for the function, Storage for local variables, Frame Pointer, Saved Registers
- **Stack Pointer** = The top of stack region (first unused location)
- **Frame Pointer** = points to a fixed location in a stack frame
- **Saved Registers** = memory to temporarily hold GPR value during **register spilling**

2.2.1 Function Call Convention

- On **executing** function call: **Caller**: Pass funct parameters (in reg or stack), save necessary register values, Save Return PC on stack; **Callee**: Save caller's registers, old FP, SP, Allocate space for local vars on stack, adjust SP (Stack Pointer) to point to callee's top of stack.
- On **returning** from function call: **Callee**: Restore saved registers, FP, SP;
  Caller: Continues execution
- Depending on implementation, some parts may not work.
- **Didn't save FP** - $ra will be overwritten, won't work.
- **Gets values by using FP** - If passed again by reference, the saved FP only points back to second caller, not original caller. Won't work.
- **MIPS Assem:** Executing: Caller - Save FP, copy SP to FP, save SP to Stack, Allocate space on stack, write to stack, call callee.
- Callee - Save $ra, reserve space on stack, write to stack, restore registers, deallocate space on stack, restore $ra, return to caller..

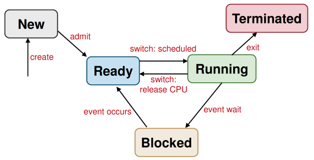
2.3 Dynamically Allocated Memory

Using a separate **heap memory region**

2.4 Process Identification & Process State

- Using process ID **(PID)**, a unique number among the processes.
- OS dependent: Are PID's reused? Are there reserved PID's? Does it limit max number of processes?

5 State Process Model:



- **Process State** = indication of the execution status

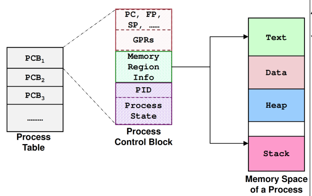
1. **New**: process created, may still be initialising, not yet ready
2. **Ready**: process is waiting to run
3. **Running**: process being executed on CPU
4. **Blocked**: process waiting, can't execute till event is available
5. **Terminated**: process finished execution, may require OS cleanup

Transitions:

- nil -> New (Create)
- New -> Ready (Admit): Process ready to be scheduled
- Ready -> Running (Switch): Process selected to run
- Running -> Ready (Switch): Process gives up CPU voluntarily or preempted by scheduler
- Running -> Blocked (Event wait): e.g. syscall, waiting for I/O, ...
- Blocked -> Ready (Event occurs)

2.5 Process Table & Process Control Block

- **PCB/Process Table Entry** = entire execution context for a process
- **Process Table** = maintains PCB for all processes, stored as one table
- Issues: Scalability, Efficiency
- PC,FP,SP,GPRs|MEM Context info(location + info of stack frame in MEM)|PID,States|



2.6 System Calls

- API to OS – different from normal function call in that have to change from user mode -> kernel mode
- General System Call Mechanism:
  1. User program invokes the library call (using normal function call mechanism)
  2. Library call places the system call number in a designated location (e.g. register)
  3. Library call executes a special instruction to switch user -> kernel mode (commonly known as TRAP)
  4. In kernel mode, the dispatching to the appropriate system call handler by dispatcher
  5. System call handler is executed
  6. System call handler ended, control return to library call, switch kernel -> user mode
  7. Library call return to user program via normal function return mechanism

2.7 Exception & Interrupt

Exception:

- Synchronous, occurring due to program execution
- Effect: have to execute an **exception handler**, similar to a forced function call

Interrupt:

- External events interrupting execution, usually hardware-related
- Asynchronous, occurring independent of program execution
- Effect: execution is suspended, have to execute **interrupt handler**

2.8 Process Abstraction: Unix

- `int fork();` duplicate current executable, returns PID of newly created process (for parent) or 0 (for child)
- `int execl(const char *path, const char *arg0, ..., const char *argN, NULL);` replaces current executing process image, does not return unless error. Will not exit on error.
- `void exit(int status);` status is 0 for normal, else problematic. Does not return.
- `int wait(int *status);` returns the PID of terminated child, status stores exit status. Blocking.

Zombie process = (1) process exited, but parent still running. process is waiting on parent to call wait(), then status will be used and pcb can remove entry.

Orphaned process = (1) parent terminates before child – `init` becomes pseudo-parent, who will call `wait` on children (2) child process terminates but parent did not call `wait` – child becomes orphaned, can fill up processs table

3 Process Scheduling

3 categories of **processing environment**: (1) **Batch Processing**: no user, no interaction, no need to be responsive, (2) **Interactive**: with active user interacting, need to be responsive, consistent in response time, (3) **Real-time Processing**: deadline to meet, usually periodic process

3.1 Criteria for Scheduling Algorithms

- **Fairness**: fair share of CPU time, no starvation
- **Utilization**: all parts of CPU should be utilized

3.2 Types of scheduling policies

- **Non-preemptive (cooperative)** – a process stays scheduled until it blocks/gives up the CPU voluntarily
- **Preemptive**: A process is given a fixed time quota to run (possible to block or yield early), at the end of the time quota, the running process is suspended.

3.3 Scheduling a process

1. Scheduler is triggered (OS takes over)
2. If context switch is needed: context of current running process is saved, placed on blocked/ready queue
3. Pick a suitable process **P** to run based on scheduling algorithm
4. Setup the context for **P**
5. Let process **P** run

3.4 Scheduling for Batch Processing

Criteria:

- **Turnaround time**: Total time taken = finTime - arrTime
- **Waiting time**: Ready but cannot get CPU = TurnaroundT - Ideal - arr
- **Response time**: First CPU Time - Arrival Time
- **Throughput**: Rate of task completion = nTasksFinished/time
- **CPU Utilisation**: % of time when CPU is working on a task

3.4.1 First-Come First-Served (FCFS)

- Batch processing, non-preemptive
- Tasks are stored on a FIFO queue based on arrival time. Pick the head of queue to run until (task is done OR task is blocked). Blocked task removed from queue, when it is ready again, placed at back of queue like a newly arrived task.
- Guaranteed to have **no starvation**: no of tasks in front of task X in FIFO is always decreasing -> task X will get its chance eventually.
- Shortcoming: **Convoy Effect** – due to non-preemptiveness, one slow process (CPU intensive) slows down the performance of the entire set of processes.

3.4.2 Shortest Job First (SJF)

- Batch processing, non-preemptive
- Select the task with the smallest total CPU time, thus guaranteeing **smallest average waiting time**.
- Shortcomings: Need to know total CPU time for a task in advance (have to guess if not available), **starvation is possible** (biased towards short jobs, long jobs may never get a chance)
- Predicting CPU Time, common approach (**Exponential Average**): $\text{Predicted}_{n+1} = \alpha \text{Actual}_n + (1-\alpha)\text{Predicted}_n$, where α = degree of weighting decrease, higher α discounts older observations faster

3.4.3 Shortest Remaining Time (SRT)

- Batch processing, preemptive
- Select job with shortest remaining (or expected) time.
- Variation of SJF that is preemptive and uses remaining time.
- New job with shorter remaining time can preempt currently running job
- Provide good service for short jobs even when they arrive late

3.5 Scheduling for Interactive Systems

- Criteria:
  - **Response time**: Time between request and response by system
  - **Predictability**: Lesser variation in response time
- Preemptive scheduling algorithms are used to ensure good response time, thus scheduler needs to run periodically.
- **Timer interrupt** = interrupt that goes off periodically based on hardware clock
- Timer interrupt handler **invokes OS scheduler**.
- **Interval of Timer Interrupt (ITI)** typically 1-10ms
- **Time Quantum** = execution duration given to a process, can be constant/variable, must be multiple of ITI (commonly 5-100ms)

3.5.1 Round Robin (RR)

- Interactive, preemptive
- Tasks stored in a FIFO queue, pick task from head of queue until (time quantum elapsed OR task gives up CPU voluntarily OR task blocks)
- Basically a preemptive version of FCFS
- Response time guarantee: given n tasks and quantum q, time before a task get CPU is bounded by $(n-1)q$
- Choice of time quantum: big = better CPU util, longer waiting time; small = bigger overhead (worse CPU util) but shorter waiting time

3.5.2 Priority Scheduling

- interactive, both pre and non-pre
- Assign a priority value to all tasks, select task with highest priority value.
- Preemptive version: highest 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
- Shortcomings: **Low priority process can starve**, worse in preemptive variant
- Possible solutions: **Decrease the priority of currently running process after every time quantum**, Given the current running process a time quantum – this process not considered in the next round of scheduling
- Generally **hard to guarantee/control exact amount of CPU time** given to a process
- **Priority Inversion**: 3 processes, priorities Hi, Mi, Lo. Lo locks resource, Mi pre-empts Lo, Hi arrives and tries to lock same resource as Lo. Then Mi continues executing although Hi has higher priority.

3.5.3 Multi-level Feedback Queue (MLFQ)

- interactive, preemptive
- Adaptive, minimising both response time for IO-bound and turnaround time for CPU-bound
- Rules:
  - Priority(A) > Priority(B) -> A runs
  - Priority(A) == Priority(B) -> A and B in RR
  - New job -> highest priority
  - If a job fully utilised its time slice -> priority reduced
  - If a job gives up/blocks before it finishes the time slice -> priority retained
- **Shortcomings**:
- (1) Starvation – processes in the lowest priority queue are in danger of being starved for CPU attention. (if process has long cpu requirements, can starve. if too many interactive Ps come in.)
- **Possible Problems + solutions**:
- **(1) Change of heart**: process with long cpu then i/o intensive phase. **Timely Boost**: after some time period S, move all jobs to the highest priority. Allows re-evaluation of processes as cpu or io bound, makeing sure change of heart processes are treated correctly.
- **(2) Gaming the system**: processes gaming the scheduler by running for 99% of time quantum, then relinquish the CPU. **Accounting Matters**: Accumulate total CPU usage across TQs

3.5.4 Lottery Scheduling

- interactive, preemptive
- Give out "lottery tickets" to processes. When a scheduling decision is needed, a ticket is chosen randomly among eligible tickets.
- In the long run, a process holding X% of tickets can win X% of the lottery held and use the resource X% of the time.
- **Reponsive**: newly created process can imediately participate in next lottery
- **Good level of control**: A process can be given lottery tickets to be distributed to its child process, an important process can be given

more lottery tickets, each resource can have its own set of tickets (different proportion of usage per resource per task)
- **Simple** implementation

4 Process Alternative – Threads

- Motivation:
  - Process is expensive: under `fork()` model – duplicate memory space and process context, context switch requires saving/restoration of process information
  - Hard for independent processes to communicate with each other: independent memory space – no easy way to pass information, requires Inter-Process Communication (IPC)
- A traditional process has a single thread of control – only one instruction of the whole program is executing at any one time. Instead, we add more threads of control such that multiple parts of the program are executing simultaneously conceptually.

4.1 Process and Thread

- A single process can have multiple threads
- Threads in the same process shares: **Memory Context** (text, data, heap), and **OS Context** (PID, other resources like files, etc.)
- Unique information needed for each thread: Identification (usually thread id), Registers (general purpose & special), "stack"
- Process context switch involves: OS Context, Hardware Context, Memory Context
- Thread switch within the same process involves: Hardware context (registers, "stack" – actually just changing FP and SP)

4.2 Benefits

- **Economy**: requires much less resources
- **Resource sharing**: no need for additional information passing mechanism
- **Responsiveness**: multithreaded programs can appear much more responsive
- **Scalability**: Multithreaded program can take advantage of multiple CPU's

4.3 Problems

- **System call concurrency** – have to guarantee correctness and determine the correct behaviour
- **Process behaviour** – impact on process operations, e.g. does `fork()` duplicate threads? If single thread executes `exit()`, hwo abut the whole process, etc.

4.4 Thread Models

- **User Thread**
  - Implemented as a user library, a runtime system in the process handles thread operations
  - Kernel is not aware of threads in the process.
  - **Advantages**: Multithreaded program on ANY OS, thread operations are just library calls, more coniguirable and flexible (such as customised thread scheduling policy)
  - **Disadvantages**: OS is not aware of threads, scheduling is performed at process level. One thread blocked -> process blocked -> all threads blocked, cannot exploit multiple CPUs
- **Kernel Thread**
  - Implemented in the OS, thread operation as system calls.
  - Thread-level scheduling is possible
  - Kernel may make use of threads for its own execution
  - **Advantages**: Kernel can schedule on thread level
  - **Disadvantages**: Thread operation is a syscall (slower and more resource intensive), generally less flexible (used by all multithreaded programs – many features: expensive, overkill for simple program, few features: not flexible enough for some)
- **Hybrid Thread Model**:
  - Have both kernel and user threads, OS schedule on kernel threads only, user thread can bind to a kernel thread.
  - Great flexibility (can limit concurrency of any process/user)

4.5 Threads on Modern Processor (Intel Hyperthreading)

- Threads started off as software mechanism: Userspace lib -> OS aware mechanism
- Hardware support on modern processors, supplying multiple sets of registers to allow threads to run natively and parallelly on the same core: **Simultaneous Multi-Threading (SMT)**

4.6 POSIX Threads: `pthread`

- Standard by IEEE, defining API and behaviour.
- `int pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine);`
- `int pthread_exit(void* exitValue)`
- `int pthread_join(pthread_t threadID, void **status);`
- except for `pthread_exit`, return 0 = success

5 Inter-Process Communication

- 2 common IPC mechanisms: Shared-Memory & Message Passing
- 2 Unix-specific IPC mechanisms: Pipe and Signal

5.1 Shared-Memory

- General idea: Process p_1 creates a shared memory region (master process) M, process p_2 attaches m to its own memory space. p_1 and p_2 can now commmunicate suing memory region M
- OS involved only in creating and attaching shared memory region
- **Advantages**: Efficient (only initial steps involves OS), Ease of use (information of any type or size can be written easily)
- **Disadvantages**: Synchronisation (shared resource -> need to synchronise access), Implementation is usually harder
- In SysV: (1) create shared memory region M `shm_get()`, (2) Attach M to process memory space `shmat()`, (3) Read/Write M, (4) Detach M from memory space after `useshmdt()`, (5) Destroy M `shmctl()` (any 1 process with permission (usually master), can only destroy if M is not attached)
- In POSIX: create shm_open() -> size setup ftruncate() -> map/attach mmap() -> read/write -> unmap/detach munmap() -> close() -> unlink shm_unlink()

5.2 Message Passing

- General idea: process p_1 prepares a message M and send it to process p_2, p_2 receives the message M
- Message has to be stored in kernel memory space, every send/receive operation is a syscall
- **Advantages**: Portable (can be easily implemented on different processing environment), Easier synchronisation (using synchronous primitive)
- **Disadvantages**: Inefficient (usually requiring OS intervention), Harder to use (message usually limited in size and/or format)

5.2.1 Naming (how to identify the other party in the comm):

- **Direct Communication**
  - Sender/receiver explicitly name the other party
  - Characteristics: 1 link/pair of communicating processes, need to know the identity of the other party
- **Indirect Communication**
  - Message are sent to/received from message storage (known as mailbox or port)
  - Characteristic: 1 mailbox can be shared among a number of processes

5.2.2 Synchronisation (behaviour of the sending/receiving ops)

- **Blocking primitives** (synchronous): sender/receiver is blocked until message is received/has arrived
- **Non-blocking Primitive** (asynchronous): sender resume operation immediately, receiver either receive message if available or some indication that message is not ready yet.

5.3 Unix Pipes

- A communication channel with 2 ends, for reading and writing.
- A pipe can be shared between 2 processes (producer-consumer)
- Behaviour: like an anonymous file, FIFO (in-order access)
- Pipe functions as **circular bounded byte buffer with implicit synchronisation**: writers wait when buffer full, readers wait when buffer empty
- Variants: Multiple readers/writers, half-duplex (unidirectional) or full-duplex (bidirectional)
- `int pipe(int fd[]);` returns 0 = success. `fd[0]` reading end, `fd[1]` writing end

5.4 Unix Signal

- An async notification regarding an event sent to a process/thread
- Recipient of signal handle by a default set of handlers OR user-supplied handler
- Common signals in Unix: SIGKILL, SIGSTOP, SIGCONT, etc.

6 Synchronization

6.1 Race Condition

- There is a Race Condition when:
- 2/more processes execute concurrently in interleaving fashion AND share a modifiable resource resulting in non-deterministic execution.
- Solution: designate code segment with race condition as **critical section** where at any point in time only 1 process can execute.

6.2 Critical Section

Properties of correct implementation:

- **Mutual Exclusion**: if a process is executing in critical section, all other processes are prevented from entering it
- **Progress**: If no process is in critical section, one of the waiting processes should be granted access
- **Bounded Wait**: After a process p_i requests to enter the critical section, $\exists$ an upper-bound of number of times other processes can enter the critical section before p_i
- **Independence**: process not executing in critical section should never block other processes

Symptoms of incorrect synchronisation:

- **Deadlock**: all processes blocked -> no progress
- **Livelock**: processes keep changing state to avoid deadlock and make no other progress, typically processes are not blocked
- **Starvation**: some processes are blocked forever

6.3 Implementations of Critical Section

6.3.1 Test-and-set: an atomic instruction

- Load the current content at `MemoryLocation` into `Register`, Stores a 1 into `MemoryLocation`
- Disadvantage: busy waiting – wasteful use of processing power
- To stop code before critical section: while (TestAndSet(Lock) == 1);

6.3.2 Peterson's Algorithms

```
bool flag[2] = {false, false};
int turn;
```

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
↪ {
  // busy wait
}
// critical section
flag[0] = false;
```

```
flag[1] = true;
turn = 0;
while (flag[1] && turn == 0)
↪ {
  // busy wait
}
// critical section
flag[1] = false;
```

Disadvantages:

- **Busy Waiting**, wasteful use of processing power
- **Low level**: higher-level programming construct desirable to simplify mutex and less error prone
- **Not general**: general synchronisation mechanism is desirable, not just mutex

6.3.3 Semaphore

A generalised synchronisation mechanism, providing a way to block a number of processes and a way to unblock one/more sleeping process(es)

- `wait(S)`: if S is (+)-ve, decrement. If S is now (-)ve, go to sleep
- `signal(S)`: increment S, if pre-increment S negative, wakes up 1 sleeping process

Properties

- Given $S_{\text{initial}} \geq 0$, where #signal(S) = no of `signal()` executed, #wait(S) = no of `wait()` completed
- **Invariant**: $S_{\text{current}} = S_{\text{initial}}$ + #signal(S) – #wait(S)
- Binary semaphore, $S = 0$ or 1 known as mutex (mutual exclusion) Deadlock still possible
- `wait(S)`: if S is (+)-ve, decrement. If S is now (-)ve, go to sleep
- `signal(S)`: increment S, if pre-increment S negative, wakes up 1 sleeping process

6.4 Classical Synchronisation Problems

- **Producer-Consumer**: produce only if buffer not full, consume only if buffer not empty
- **Reader-Writers**: writer exclusive access, reader can share
- **Dining Philosophers**: assign partial order to the resources, establishing convention that all resources will be requested in order. E.g. label forks 1-5, and always pick up lower-numbered fork first.

6.5 Synchronisation Implementations

- POSIX semaphores
- pthread_mutex_t: `pthread_mutex_lock`, `pthread_mutex_unlock`
- pthread_cond_t: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

6.5.1 Producer Consumer, Blocking Version

```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```

Producer Process

```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```

Consumer Process

Initial Values: count = in = out = 0; mutex = S(1), notFull = S(K), notEmpty = S(0)

6.5.2 Tanenbaum Solution

```
#define N 5
#define LEFT ((i + N - 1) % N)
#define RIGHT ((i + 1) % N)
#define THINKING 0
#define HUNGRY 1
sem_t mutex;
sem_t s[N];
void philosopher(int i)
{
    while (TRUE)
    {
        Think();
        takeChpStcks(i);
        Eat();
        putChpStcks(i);
    }
}
void takeChpStcks(i)
{
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
    signal(mutex);
    wait(s[i]);
}
```

```
#define EATING 2
#define TRUE 1
int state[N];
#include <semaphore.h>
```

```
void safeToEat(i)
{
    if ((state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING))
    {
        state[i] = EATING;
        signal(s[i]);
    }
}
void putChpStcks(i)
{
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

```
void philosopher( int i ){
    while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
    }
}
```

Or we can have limited eaters