What is OS

Special layer of software that provides application software access to hardware resources:
- Convent abstraction of complex hardware device
- Protected access to shared sources
- Security and authentication
- Communication among logical entities

OS includes
- kernel manage all the physical devices
- system call configure kernel or build things
- driver handle the interaction
- shell render a simple command line

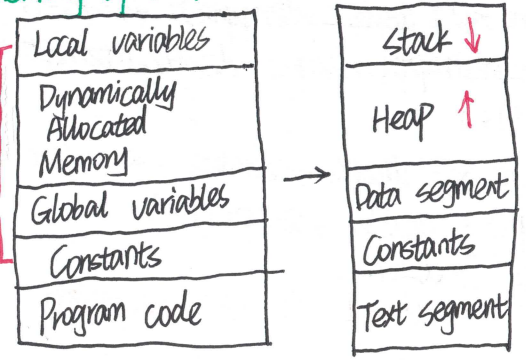
OS do
- Provide abstractions to apps (File systems; Process, threads; VM, containers)
- Manage resources (Memory, CPU, storage)

What is System call
- is a function call
- exposed by the kernel
- abstract away most low-level details

NOT System Call
printf, scanf, malloc, free, fopen, fclose

OS System Call
mkdir, rmdir, chown, chmod, open, fchdir, chmod, getpid, pause, pipe, read, signal, stat, time, wait, write

Memory of a Process



Four Fundamental OS Concept

Thread • Single unique execution context. fully describes program state
• PC, Register, Execution Flags, Stack

Address Space • Programs execute in an address space that is distinct from the memory space of the physical machine.

Process • An instance of an executing program is a process consisting of an address space and 1/more thread of control

Dual mode operation / Protection • Only the system has the ability to access certain resources. • The OS and the hardware are protected from user programs and user programers are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses

CPU need structure to hold PC, SP (Stack Pointer), Register

How switch from one CPU to the next
1. Save PC, SP and register in current state
2. Load PC, SP and register from new state block.

What triggers switch
Timer, voluntary yield, IIO

3 Types of Mode Transfer = Unprogrammed
System Call, Interrupt, Trap or Exception

Multithreaded Processes

code	data	files
reg stack	reg stack	reg stack
S	S	S

Banker
[Avail] = [FreeResources]
Add all nodes Unfinished
do{
  done = true;
  Foreach node UF{
    if([Request] ≤ [Avail]){
// [Max] - [Alloc] ≤ [Avail]
      remove node from UF
      [Avail] = [Avail] + [Alloc]
      done = false;
    }
  }
} while(!done)

fork()

It creates the child process by cloning from the parent process, including all user-space data: PC [CPU reg], Program code [File & Memory], Memory, Opened files [Kernel's internal]
However, DO NOT CLONE Return value of fork(), PID, Parent process, Running time, File locks.

exec()

The process is changing the code that is executing and never return to the original code. Will replace: Program Code, Memory (local v, global v, dynamically allocated memory), Reg value (PC)
However, the kernel-space info is preserved (PID, Process relationship)

wait()

suspend the calling process to waiting state and return when
- one of its child (running → terminated) • signal is received
- return immediately if it has no child.

fork() inside the kernel

1. copy kernel space 2. update kernel space (PID, Running time, pointer to my parent / List of children) ☆ Array of opened files → preserv
3. copy user space 4. return value

exec() inside the kernel

1. clear local v & dynamically allocated memory, reset global v based on new code, change constants to the new program code
The kernel will also reset the register value (PC)

exit() inside the kernel

1. clean up most of the allocated kernel space memory
2. clean up the exit process's user space memory
3. notify the parent with SIGCHLD ← State: zombie only has pid

wait() inside the kernel

1. register a signal handling routine 2. when SIGCHID comes, the corresponding signal handling routine is invoked (remove the signal, destroy the child process in the kernal space) 3. deregister the signal handling routine for the parent 4. return the PID of child.

exit() turns a process into a zombie when
1. The process calls exit()
2. The process return from main()
3. The process terminates abnormally

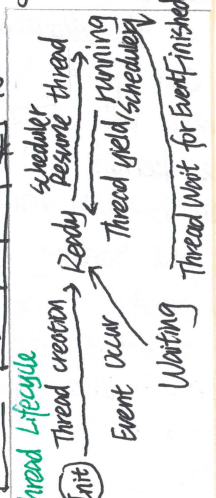
hen invoking a system call [kernel mode bit = 0]
. PC is pointing to the region (code) when in kernel
. CPU switches from userspace to kernel space,
reads the PID of the process from the kernel
. it switches back to the userspace memory,
continue running program code

ime
User Time: CPU time spent on codes in userspace mem.
Sys Time: CPU time spent on codes in kernelspace mem.

rocess Lifecycle
Forked → Ready ⇌ Running → Zombie
Block / Waiting
ontext switch → Save (same process: TCB, reg, PC, SP; diff process: PCB, file-descriptor)
he scheduler decides to schedule another process in
he ready queue. Then the schedule has to load the
ontext of that process from the main memory to CPU
xpensive]: • Save and restore reg • Switch address space
• cache, buffer cache & TLB misses

PU-bound Process
pend most of time in CPU, user-time > sys-time [AI]
/O-bound Process [real_time > usertime + systime]
pend most of time in I/O, sys-time > user-time ['ls']
urnaround time Finish_time - Arrive_time
aiting time Turnaround time - task_time
R: The responsiveness of the process is greater, not frozen

hread state
shared state (PC)
o state) Per-Thread state
Heap
Global variable
Code
Thread Control Block
Stack Info
Saved Reg
Thread Metadata
Stack



Multithread Processes
• PCB point to multiple TCB
• switching thread within a block is simple
• switching thread across blocks requires changes to memory and I/O address tables
Multiprocessing ≡ Multiple CPUs
Multiprogramming ≡ Multipe Jobs / Process
Multithreading ≡ Multiple threads per process

Multiprocessing { A → / B →
Multiprogramming A → B →

Threading Model
One-to-One, Many-to-One, Many-to-Many

IPC
Shared Objects < FIFO, Pipe (> system), semaphore → user
Message Passing
Signal

Race condition
• Happen whenever shared object + multiple process + concurrently
• The outcome of the computation depends on the execution sequences of the process

Mutual exclusion is a requirement
• If it could be achieved, the problem of race condition would be gone.

A critical section
the code segment that access shared object.
☆ can be designed for accessing more than one shared object

Entry and exit implementation
☆ Mutual Exclusion ☆ Bounded Waiting ☆ Progress

Achieving Mutual Exclusion
1. Spin-based lock 2. Spin Smarter 3. Semaphore
↓ violate progress ↓ suffer from priority inversion

Spin Smarter
```
int turn; int interested[2] = {FALSE, FALSE};
void lock(int process) {
  int other; other = 1-process; interested[process] = TRUE;
  turn = other; while(turn == other && interested[other]);
void unlock(int process) { interested[process] = FALSE; }
```
→ A Low priority process L is inside the critical region, but a high priority process H gets the CPU and want to enter the critical region. But H cannot lock (since L has not unlock).

Producer-consumer
SO: semaphore mutex = 1; avail = N, fill = 0;
ES: void producer { int item; while(true){ item = produce_item();
wait(&avail); wait(&mutex); insert_item(item);
post(&mutex); post(&fill); } }
ES: void consumer { int item; while(true){ wait(&fill);
wait(&mutex); item = remove_item(); post(&mutex);
post(&avail); } }

Dining philosopher (LEFT ((i+N-1)%N) Right ((i+1)%N)
SO: int state[N]; semaphore mutex = 1, p[N] = 0
MF: void philosopher(int i) { think(); take(i); eat(); put(i); }
SE: void take(int i) { wait(&mutex); state[i] = Hungry;
captain(i); post(&mutex); wait(&p[i]); }
SE: void put(int i) { wait(&mutex; state[i] = Think;
captain(LEFT); captain(RIGHT); post(&mutex); }
HF: void captain(int i) { if(state[i] = H & state[L] != E & state[R] != E) { state[i] = E; post(&p[i]); }

wait & post
```
void wait(semaphore *s) { disable_interrupt(); *s = *s - 1;
if(*s < 0) { enable_interrupt(); sleep(); disable_interrupt(); }
enable_interrupt(); }
void post(semaphore *s) { disable_interrupt(); *s = *s + 1;
if(*s <= 0) wake-up(); enable_interrupt(); }
```

Four requirement for DeadLock
• Mutual Exclusion • Hold and Wait • No preemption