14.09.22 Lec 6

Process Management

OS provides abstraction to user programs

→ Hardware is complex

→ Challenging to understand how the hardware actually works

3 fundamental abstractions that OS provides to the user programs

→ may hurt performance in terms of execution time or additional memory or add. storage

→ Process

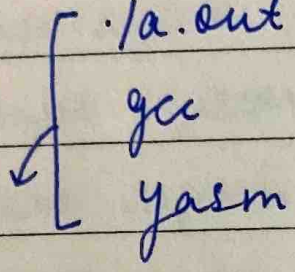
→ Virtual Memory

→ File System

Process

When you run a program, the actual binary has to be loaded into the main memory. There is a "running instance" of the program that is consuming memory, consuming time of the process and

making requests to I/O devices. This running instance is called process.



Processes

While they are in execution

Suppose you have a single processor, you can run multiple processes.

→ Time-sharing

→ Job of process management system of OS is to ensure that no process is unfairly hurt and they all get fair access to the processor's time.

1) How can it use the I/O devices?

→ Processes are >1,00,00 times faster than I/O devices

→ Interrupts are needed to signal the need for I/O

→ User programs can only raise an interrupt & the OS has a device management

System which can directly interface with the actual I/O device

Interrupts when raised → normal execution of the program stops.

→ control of execution jumps to a specific location in the memory depending on the type of interrupt.

→ during bootup, the memory management system of OS loads specific instructions to handle each type of interrupt at different possible locations.

int #1 –
int #2 – ← store the specific interrupt service routines
int #3 – in these locations.

processor execution mode or privilege level

gui is a process

→ Kernel has privilege level 0 so it can execute everything. Whereas user programs have higher values set in the privilege mode, so it cannot access directly the I/O devices

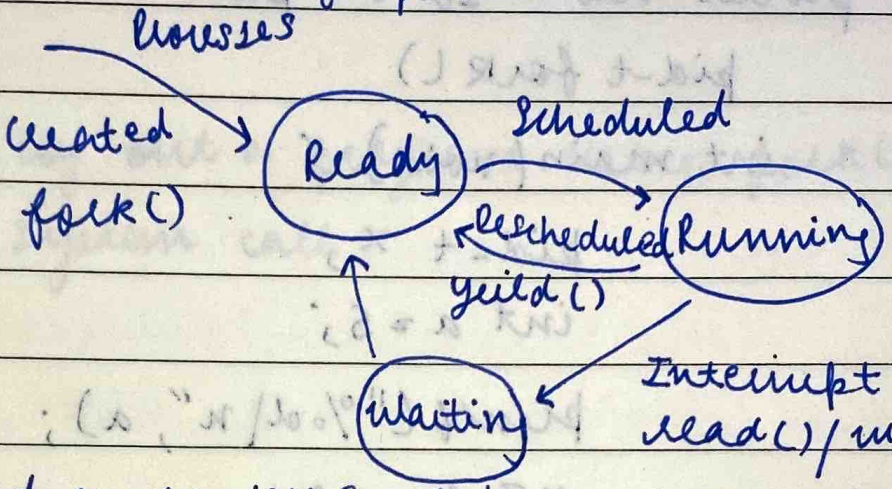
Processes and their privilege levels help build isolation among different programs

All the interrupts cause a context-switch. The context switch implies that a privilige mode is being changed. That has some overhead, because you want to ensure that the execution can resume.

Process-switch: → If the process management wants to stop execution of one process & start execution of another process, then it has to perform a process switch. Here the overhead is higher than in context-switch. requires process management system to save the "state" of the entire process.

Data structure called (PCB) stores details of each process.

States of execution of processes



We discussed mechanisms of processes running there are complex design choices that each OS needs to make. Called policies.

① Scheduling, descheduling policies decided by process scheduler.

② Design of the system calls → how they should be used is also a design choice

User program ⟶ (System calls) OS ⟶ hardware

In VM → every inst^n goes thru OS (diff system calls)

Elsewhere → limited directed execution, everything doesn't go through the OS.

The fork() system calls — for unix

↳ create a new process

Each process has a Id or pid

```
pid_t fork()
int main(void) {
    pid_t x;
    int a = 5;
    printf("%d\n", a);
    x = fork();
    if (x == 0) {                    a = a+1
        /* child process */          printf
    }
    else if (x > 0) {
        /* parent process */         a = a+1/2
    }
    else {
        printf("Error");
    }
    printf("%d", a);
    return 0;
}
```

5 5
6 6
6 7

non deterministic output. should be avoided

Race conditions → different parts of the code are racing against each other to execute first. avoid.

glibc

The exec system calls allows you to load a binary into the main memory for execution.

Exercise → Try out a program using fork() system call.

17.09.22
Saturday

fork() creates a new address space of the child process

parallel / concurrent programs can be written

Native fork() has to copy alot of data which is not used in practice.

→ performance optimization

Copy-on-write (COW) → copy of data / code is made only when it is changed in the child / parent process.

Parallelism

Multi-core system → parallelism (MIMD)

→ because two or more inst" are executed simultaneously.

GPU → (SIMD) parallelism

→ single inst" on 2 or more data variables

Can only be used in programs that have concurrency.

concurrent $\not\Rightarrow$ parallelism

parallelism $\Rightarrow$ concurrent

i.e. even if inst^n are executed in a diff. sequence / diff. processor, the result is correct.

fork → allows to create concurrent programs and $\%$ take adv of multi core systems thread

posix_spawn() → first calls fork() then exec() s.t. add^n optimisations related to system variables are possible

instead of copying, programmer wants to the address space

control the variables after creating the new process. (can be done using vfork().

(locks on variables)

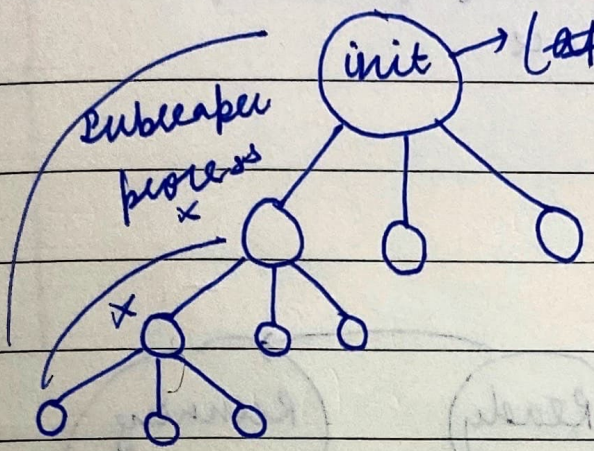
```
x = fork();
if (x > 0) {
        /* Parent */   a = a + 1
else if (x == 0)
        /* child */    a = a + 2
        exec(" ./a.out", ...);
else    { /* error */ }
```

```
int x[] = {0, 0},
    y[] = {1, 2},
    sum[] = {0, 0};

pid_t pid;
pid = fork()

if (pid > 0) {
    sum[0] = x[0] + y[0]; printf("%d",
}   wait();                    sum[0]);
else if (pid == 0) {
    sum[1] = x[1] + y[1]; printf("%d",
}                              sum[1]);
```

The wait() system call ensures that the parent process waits until child process has completed its execution.

Child process if its own parent finishes execution assigned a new parent.
Traditionally init process was assigned as the parent

In linux, concept of subreaper process was created

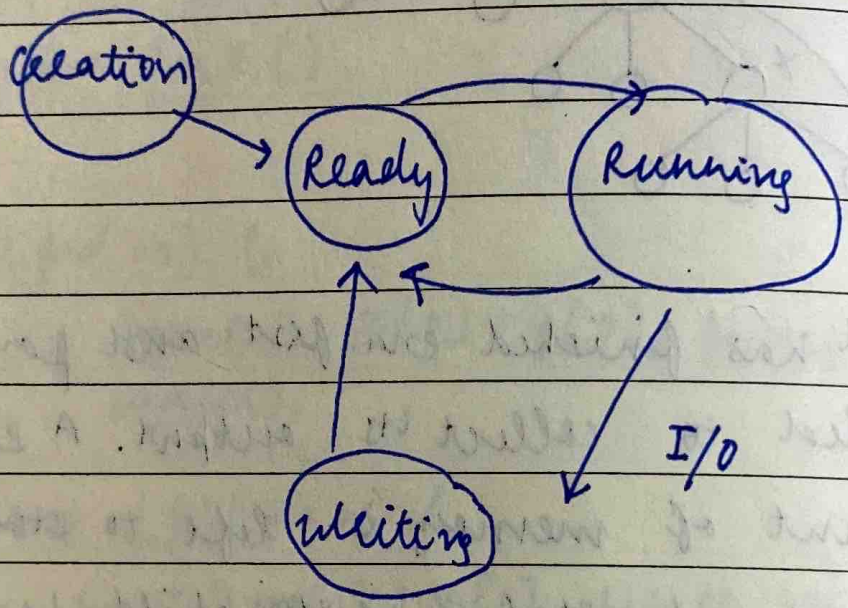


child has finished exn. first and parent has failed to collect its output. A small amount of memory is left to store the output and status of the child process. Such a process is called zombie process.

21.09.22

Process Management

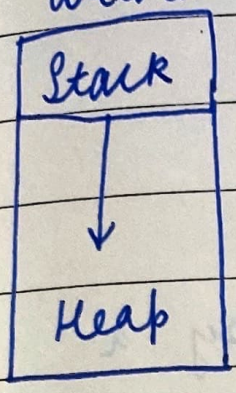
1) Create process using fork
2) Process Address space
3) Process life cycle

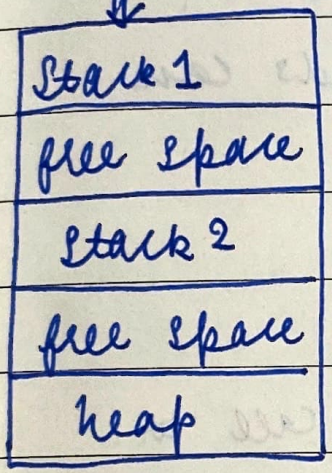


D1) All I/O would force the process to go to waiting state

D2) Any type of parallelism requires substantial amount of overhead in creating a new process.

Threads help to solve disadvantages
Threads are multiple lines of execution within a single process.





We need 2 stacks as each thread needs to do bookkeeping of which f^n calls which

```
int main() {
    x = malloc()
    abc();
    thread_create(def);
    xyz();
}
void def() {
}
```

```
void abc() {
    ijk();
}
```

can run in parallel

Threads enable concurrent access to data structures

Effort required by us to enable such access and manage shared data structures.
Responsibility of handling such race conditions is on the programmer.

User-level threads → threads created by a process internally w/o informing the kernel. Switching execution b/w the 2 threads can be done by the process itself.
"green threads"
↳ not an OS concept

Kernel-level threads → Use a system call to create a thread. This system call automatically creates a new stack & also separate PC register values. Switching execution b/w the threads also has to be done via system calls, which makes it time consuming.

The Clone() system call creates threads
posix-thread-create
posix_thread_join

```
int main(void) {
        posix_thread_create(abc);
        /* computation of your
                        own thread */
        posix_thread_join(abc);
}

void abc() {
        /* computation of abc
                        thread */
}
```

```
for (i=0; i<100; i++) {
        sum[i] = x[i] + y[i];
}
```

(not threaded) → sequential

concurrency

```
void compute_sum(int arr_res[], int x[], int y[],
                        int start, int end) {
    for(i=start; i<end; i++) {
        sum[i] = x[i] + y[i];
    }
}

int main(void) {
    int x[100], y[100], sum[100]; posix_thread
                                    *t[4]
    for(int i=0; i<4; i++) {
```