**DAY-22(27-11-2024)**

**SYSTEM PROGRAMMING:**

* Multi-tasking can be running by using multi-threading and forks
* KERNEL MODE:
* The cpu operates in a privileged mode which is unrestricted access to hardware resources.
* It can execute all instructions and access protected areas of memory.
* OS kernel runs in this mode to manage system level operations like memory management, process scheduling and device control
* USER MODE:
* The cpu operates in a restricted mode where applications are limited to accessing specific memory areas and cannot directly interact with hardware,it must request the kernel via **system calls**

User 🡪kernel 🡪 hardware

* Eg : in c program we cannot modify the hardware stings directly we use system calls (read() or write()) instead of directly manipulating .
* SYSTEM CALLS:
* It acts as an interface between user programs and the operating system.
* They enable user mode applications to request kernel services like file handling, process management and network communication
* To connect server to client vice versa by using IP address (like apartment) and port number (like house).
* Need of system calls :
* Types of system calls:
* Process control: creation ,termination and management of process (eg : fork(),exec(),exit())

Exit() 🡺 it terminate completely ,your interrupting the process

Fork() 🡺 create a multi process

Exec() 🡺 to execute of one program from another program

* File management ; operations like reading ,writing,open and close files eg: open(),close(),read(),write()
* Device management: interact with hardware devices

Eg: ioctl(),read()

* Information maintenance : accessing system information

Eg: getpid(),gettimeofday()

* Communication: managing inter-process communication

Eg : pipe(),socket()

Fcntl 🡺 it gives all the details about files like locking and unlocking and mode of the file and when it is modified

* Command to view system calls== strace
* Strace is a debugging tool in linux used to trace system calls made by a program .it shows all interactions between the program and os kernel.
* Strace ./your\_program
* Process basics:
* Process control block(PCB) :
* it is a data structure maintained by the OS to store all the information about a specific process
* Contents of PCB:

1. Process id 🡺 unique identifier
2. Process state🡺 current state eg: running,ready,waiting
3. Program counter🡺 address of next instruction to execute
4. Registers 🡺 values of CPU registers
5. Memory management information
6. Accounting information
7. i/o information
8. scheduling information

* PCB is critical for the context switching process it saves and restores the state of the process during switching
* Process: it is a program in execution it is the basic unit of execution

1) process id

2) parent process

3) priority 🡺 determines process scheduling

4) execution context 🡺 includes the PC, CPU register

5) resource usage

* PROCESS STATE:

1. **New** : created
2. **Ready**: process is ready to execute but its is waiting for the CPU time.
3. **Running**: process is currently being executed by the CPU
4. **Waiting:** process is waiting for the event eg: I/O completion
5. **Terminated**: finished execution

* SCHEDULING: determines which process gets the CPU and for how long It is managed by the scheduler in the OS

Types of scheduling:

1. Long term
2. short term
3. medium term

* Steps in context switching:

1. Save current process
2. Update
3. Restore
4. Switch

* In system programming we have to make sure that “defunct” process should not be there
* Ps-aux🡺gives the information about process states
* getpid() returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)
* getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented (either init(1) or a "sub‐reaper" process defined via the prctl(2) PR\_SET\_CHILD\_SUBREAPER operation).
* Fork() 🡺 it creates a child process by duplicating the calling process. New process is referedd to as child process .calling process is referred to as the parent process

🡪**The child process and parent process have the separated memory spaces**

**🡪At the time of fork() both memory spaces have the same content.**

**🡪 Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.**

**🡪** The child process is an exact duplicate of the parent process except

for the following points:

\* The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.

\* The child's parent process ID is the same as the parent's process ID.

\* The child does not inherit its parent's memory locks (mlock(2),mlockall(2))

\*Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.

\* The child's set of pending signals is initially empty (sigpending(2)).

\* The child does not inherit semaphore adjustments from its parent (semop(2)).

\* The child does not inherit process-associated record locks from its parent (fcntl(2)). (On the other hand, it does inherit fcntl(2) open file description locks and flock(2) locks from its parent.)

\* The child does not inherit timers from its parent (setitimer(2),alarm(2), timer\_create(2)).

\* The child does not inherit outstanding asynchronous I/O operations from its parent (aio\_read(3), aio\_write(3)), nor does it inherit any asynchronous I/O contexts from its parent (see io\_setup(2)).

🡪Fork is return child PID in parent process

🡪fork returns Zero in the child process

🡪fork returns -1 if its error

Program:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

int main()

{

// long int i;

/\* int pid =getpid();

pid\_t ppid=getppid();

printf("my own pid =%d\n",pid);

printf("for %d parent is %d\n",pid,ppid);

getchar();\*/

13 printf("\n i am parent\n");

14 pid\_t id=fork();

15 printf("\n hello world\n");

16 printf("\n\n");

17 return 0;

18 }

i am parent 🡺 parent related content

hello world 🡺 parent process

hello world 🡺 child

here these both hello world for the parent and child process(duplicates)

* WAIT:
* Parent have to wait until child dies
* The child will exectute first then parent if and only if no statements before fork
* First child is dies and parent waits until child dies and then parent dies
* All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
* Waits until child changes it state
* A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.
* In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state
* wait() and waitpid()

The wait() system call suspends execution of the calling process until one of its children terminates. The call wait(&wstatus) is equivalent to:

waitpid(-1, &wstatus, 0);

* WIFEXITED(wstatus) : returns true if the child terminated normally, that is, by calling exit(3) or \_exit(2), or by returning from main().
* WEXITSTATUS(wstatus) :returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or \_exit(2) or as the argument for a return statement in main(). This macro should be employed only if WIFEXITED returned true.
* Question : using nested if else create a fork and also write fork for the parent and also for the child

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t pid1, pid2, pid3;

// First fork: creating the child of the main process (parent)

pid1 = fork();

if (pid1 < 0) {

// Error handling

perror("First fork failed");

exit(1);

}

else if (pid1 > 0) {

// This is the parent process after first fork

printf("Main (Parent) process: PID = %d, Child1 PID = %d\n", getpid(), pid1);

// Second fork: the parent creates another child (Child2)

pid2 = fork();

if (pid2 < 0) {

// Error handling

perror("Second fork failed");

exit(1);

}

else if (pid2 > 0) {

// Parent process after second fork

printf("Parent process: PID = %d, created Child2 with PID = %d\n", getpid(), pid2);

}

else {

// Child2 process (created by the parent)

printf("Child2 process: PID = %d, Parent PID = %d\n", getpid(), getppid());

}

}

else {

// This is the first child process (created by the main process)

printf("Child1 process: PID = %d, Parent PID = %d\n", getpid(), getppid());

// Third fork: the first child (Child1) creates another child (Child3)

pid3 = fork();

if (pid3 < 0) {

// Error handling

perror("Third fork failed");

exit(1);

}

else if (pid3 > 0) {

// Child1 process after third fork

printf("Child1 process: PID = %d, created Child3 with PID = %d\n", getpid(), pid3);

}

else {

// Child3 process (created by Child1)

printf("Child3 process: PID = %d, Parent PID = %d\n", getpid(), getppid());

}

}

return 0;

}