UE20CS254

Operating Systems

Suresh Jamadagni

Department of Computer Science and Engineering

Operating Systems

System calls for Process Management

Suresh Jamadagni

Department of Computer Science and Engineering

Process Identifiers

- Every process has a unique **process ID**, a non-negative integer
- The process ID is the only well-known identifier of a process that is always unique
- It is often used as a piece of other identifiers, to guarantee uniqueness.
- Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
- Most UNIX systems implement algorithms to delay reuse, so that newly created processes are assigned IDs different from those used by processes that terminated recently

OPERATING SYSTEMS

fork()

- An existing process can create a new one by calling the **fork** function

```
#include <unistd.h>
pid_t fork(void);
```

- The new process created by fork is called the child process
- Return value in the child is 0
- Return value in the parent is the process ID of the new child
- Return value is -1 on error
- The child is a copy of the parent. The child gets a copy of the parent's data space, heap, and stack

OPERATING SYSTEMS

exit()

- A process can terminate normally in five ways
    1. Executing a return from the main function
    2. Calling the **exit** function.
    3. Calling the _exit or _Exit function.
    4. Executing a return from the start routine of the last thread in the process.
    5. Calling the pthread_exit function from the last thread in the process

A process can terminate abnormally in three ways

1. Calling abort
2. When the process receives certain signals
3. The last thread responds to a cancellation request

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- This kernel code closes all the open descriptors for the process, releases the memory that it was using

OPERATING SYSTEMS

wait() and waitpid()

- A process that calls wait or waitpid can
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes
- If the process is calling wait because it received the SIGCHLD signal, wait will return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

asn't changed or −1 on failure

OPERATING SYSTEMS

waitid()

- waitid() allows a process to specify which children to wait for.
- Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Returns: 0 if OK, −1 on error

OPERATING SYSTEMS

exec()

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created;
- exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

- Return: −1 on error, no return on success

OPERATING SYSTEMS

getpid() and getppid()

```
#include <unistd.h>
pid_t getpid(void);


pid_t getppid(void);
```

- **getpid**() returns the process ID (PID) of the calling process.

- **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 re-parented

OPERATING SYSTEMS

Race condition

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- The fork function is a lively breeding ground for race conditions, if the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork.
- In general, which process runs first cannot be predicted
- A process that wants to wait for a child to terminate must call one of the wait functions.
- If a process wants to wait for its parent to terminate, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

- The problem with this type of loop, called polling, is that it wastes CPU time, as the caller is awakened every second to test the condition.
- To avoid race conditions and to avoid polling, some form of signaling is used between multiple processes

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu