

Ateneo de Manila University

Process Control and Execution



Department of Information Systems and
Computer Science

S.Y. 2001-2002

<http://sysads.ateneo.net/wyu/>

wyy@admu.edu.ph

Multiprocess Programming

- ★ is basically a single running program
- ★ can be a system initiated program or user space program
- ★ each process is identified by a unique process ID (PID)
- ★ process are treated as separate program and do not communication with each other
- ★ special mechanisms for process communications:
 - signals
 - semaphores
 - shared memory
 - pipes
 - message passing

Section I

Current Process

Current Process Information

★ defined in the `unistd.h` and `sys/types.h` header files

★ `pid_t getpid(void);`

★ `pid_t getppid(void);`

- gets the process identification numbers

- `getpid` gets the current process id

- `getppid` gets the parent process id of the current process

Process Groups

- ★ is a collection of one or more processes
- ★ the process group leader's PID is the same as the Process Group ID (PGID)
- ★ defined in the `unistd.h` and `sys/types.h` header files

★ `int setpgid(pid_t pid, pid_t pgid);`

★ `int setpgrp(void);`

- sets the process group of the current process
- returns -1 if an error occurs
- `setpgid ()` sets the process group of `pid` to `pgid`
- If `pid` is zero, the process ID of the current process is used
- If `pgid` is zero, the process ID of the process specified by `pid` is used
- `setpgrp ()` is similar to `setpgid (0 , 0)`

```
★ pid_t getpgid(pid_t pid);
```

```
★ pid_t getpgrp(void);
```

- returns the process group ID of the process specified by `pid`
- If `pid` is zero, the process ID of the current process is used
- returns -1 if an error occurs
- `getpgrp ()` is similar to `getpgid (0)`

Session

- ★ is a collection of one or more process groups
- ★ defined in the `unistd.h` and `sys/types.h` header files

- ★ `pid_t setsid(void);`
 - creates a new session and returns the session ID (SID)
 - sets the process group ID and session ID of the current process to SID
 - if the calling process is a session leader the current SID is returned
 - returns -1 if an error occurs

- ★ `pid_t getsid(pid_t pid);`
 - gets the session ID of the process `pid`
 - `getsid (0)` returns the session ID of the calling process
 - returns -1 if an error occurs

Exit Functions

- ★ defined in the `stdlib.h` header file
- ★ `void exit(int status);`
 - terminates the current process
 - performs process cleanup before returning control to the operating system
- ★ `void _exit(int status);`
 - terminates the current process
 - does not perform system cleanup
- ★ `int atexit(void (*function)(void));`
 - registers a function to be called upon normal program termination
 - functions are called exit handlers
 - exit handlers are executed in last in first out fashion
 - returns a -1 if registration is not successful

Environment Variables

★ defined in the `stdlib.h` header file

★ `char *getenv(const char *name);`

- searches the environment list for the environment variable defined by `name`
- returns a pointer to the value in the environment
- returns `NULL` if an error occurred

★ `int setenv(const char *name, const char *value, int overwrite);`

★ `void unsetenv(const char *name);`

- correspondingly sets the environment variable defined by the `name` and `value` pair
- overwrites existing data if `overwrite` is non-zero
- returns `-1` if an error occurs

- `unsetenv()` unsets the environment variable

★ `extern char **environ;`

- points to an array of strings that contains the environment

- each environment variable is defined as a `name=value` pair

- defined in the `unistd.h` header file and the `_GNU_SOURCE` must be defined

Section II

Process Management

Spawning a New Process

★ defined in the `unistd.h` and `sys/types.h` header files

★ `pid_t fork(void);`

★ `pid_t vfork(void);`

- creates a child process
- returns the pid of the child for the parent and zero for the child
- both parent and child continue executing the instructions after the `fork` functions is executed
- returns -1 on error
- `vfork()` is used when the only purpose of the forked processes is to call the `exec()`-like functions
- `vfork()` typically uses the address space of the parent but not entirely and not always

Typically Calling Sequence

```
if ( ( pid = fork ( ) ) < 0 )  
  
    printf ( "error forking\n" );  
  
else if ( pid == 0 ) {  
  
    /* execute child stuff here */  
  
} else {  
  
    /* place parent stuff here */  
  
}
```

Process Termination

- ★ defined in the `sys/wait.h` and `sys/types.h` header files
- ★ `pid_t wait(int *status);`
- ★ `pid_t waitpid(pid_t pid, int *status, int options);`
 - functions used to block until a process terminates
 - `wait()` waits until a child has exited
 - in `wait()` if the child has already exited this functions immediately returns
 - `waitpid()` waits until a child specified by `pid` returns
 - in `waitpid()` can have the values:
 - * `< -1` to wait for any process in that process group `-pid`
 - * `-1` makes `waitpid` behave like `wait`
 - * `0` waits for any child in the same process group

- * > 0 wait for process defined by `pid`
- returns the PID of the child, returns -1 if an error occurred, returns 0 if no child was available
- if the value of `status` is not NULL then `status` points to the location of the status information

Potential Problems with Multiprocess Programming

- ★ runaway forking
- ★ data dependencies
- ★ synchronization and race conditions

Section III

Remote Execution

Exec and Friends

★ defined in the `unistd.h` header file

```
★ int execl( const char *path, const char *arg,  
    ... );
```

```
★ int execlp( const char *file, const char *arg,  
    ... );
```

```
★ int execl( const char *path, const char *arg ,  
    ..., char* const envp[] );
```

```
★ int execv( const char *path, char *const  
    argv[] );
```

```
★ int execvp( const char *file, char *const  
    argv[] );
```

```
★ int execve (const char *filename, char *const
```

```
argv [], char *const envp[]);
```

- executes a file
- this replaces the current process image with the new process
- `execl()`, `execlp()`, `execle()`, `execv()`,
`execvp()` are frontends to the `execve()` function
- returns -1 if an error occurs



Copyright © 2000-2001 by William Emmanuel S. Yu. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at <http://opencontent.org/opl.shtml>).