

CPU Events Relative to Real Time

Item	Time	Scaled Time in Human Terms (2 billion times slower)
Processor Cycle	0.5 ns (2GHZ)	1 second
Cache Access	1 ns	2 seconds
Memory Access	15ns	30 seconds
Context Switch	5,000ns (5 μ s)	167 minutes
Disk Access	7,000,000ns (7 ms)	162 days
Time Quantum	100,000,000ns (100ms)	116 days

Screen Filling Comparisons

Modem type	Bits per second	Time needed to display	
		Text	Graphics
1979 telephone modem	300	1 min	6 hours
1983 telephone modem	2,400	6 secs	45 mins
current telephone modem	57,600	0.28 secs	109 secs
current DSL modem	768,000	0.02 secs	8 secs

POSIX Extensions

code	extension	Solaris 9
AIO	asynchronous input and output	yes
CX	ISO C standard extension	yes
FSC	file synchronization	yes
RTS	real time signals	yes
SEM	semaphores	yes
THR	threads	yes
TMR	timers	yes
TPS	thread execution scheduling	yes
TSA	thread stack address attribute	no
TSF	thread-safe functions	yes
XSI	XSI extension	yes

Thread Safe Functions

A function is “thread safe” if it can be safely invoked by multiple threads

Async-Signal Safe Functions

A function is async-signal safe if that function can be called without restriction from a signal handler

Process Termination

Normal or abnormal.

- Cancel pending timers and signals
- Release virtual memory resources
- Release other process-held system resources such as locks
- Close open files

Zombies

- If a parent process terminates before a child finishes, the child cannot be terminated by its parent. We call these child processes zombies.
- Orphaned child processes become zombies when they terminate.
- System init process (process whose ID is 1) gets rid of orphaned zombies.

Normal Termination

- Return from main.
- Call to C function `exit` or `atexit`
- Call to `_exit` or `_Exit` system call.

(note that C function `exit` calls user-defined exit handlers that usually provides extra cleanup before calling on `_exit` or `_Exit`).

exit and _exit

- Take an integer parameter *status* that indicates the status of the program.
- 0 normal termination.
- Programmer defined non-zero indicates error.
- At exit C function installs user-defined exit handler. Last-installed, first executed.

exit, _Exit, _exit Synopsis

SYNOPSIS

```
#include <stdlib.h>
```

```
void exit (int status);
```

```
void _Exit (int status);
```

ISO C

SYNOPSIS

```
#include <unistd.h>
```

```
void _exit (int status);
```

POSIX

atexit Synopsis

SYNOPSIS

```
#include <stdlib.h>
```

```
int atexit (void (*func)(void));
```

ISO C

atexit installs a user-defined exit handler. If successful, atexit returns 0 and executes the handler function. If unsuccessful, atexit returns a non-zero value

Abnormal Termination

- Call abort.
- Process a signal that causes termination.

fork System Call

Creates child process by copying parent's memory image

SYNOPSIS

```
#include <unistd.h>
```

```
pid_t fork(void)
```

POSIX

fork return values

- Returns 0 to child
- Returns child PID to parent
- Returns -1 on error

Fork Attributes

Child inherits:

- Parent's memory image
- Most of the parent's attributes including environment and privilege.
- Some of parent's resources such as open files.

Child does not inherit:

- Parent pid.
- Parent time clock (child clock is set to 0).

fork Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main(void)
{
    pid_t childpid;
    childpid = fork()
    if(childpid == -1 {
        perror("failed to fork");
        return 1; }
    if(childpid == 0) {
        fprintf(stderr, "I am the child, ID = %ld\n", (long)getpid());
        /* child code goes here */
    } else if (childpid > 0) {
        fprintf(stderr, "I am the parent, ID = %ld\n", (long)getpid());
        /* parent code goes here */
    }
}
```

fork (Chain)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main(void)
{
    int i;
    int n;
    pid_t childpid;
    n = 4;
    for (i = 1; i < n; ++i)
        if (childpid = fork())
            break;
    fprintf(stderr, "This is process %ld with parent %ld\n",
            (long)getpid(), (long)getppid());
    sleep(1);
}
```

fork (Fan)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void)
{
    int i;
    int n;
    pid_t childpid;
    n = 4;
    for (i = 1; i < n; ++i)
        if ((childpid = fork()) <= 0)
            break;
    fprintf(stderr, "This is process %ld with parent %ld\n",
            (long)getpid(), (long)getppid());
    sleep(1);
}
```

fork (Tree)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void)
{
    int i;
    int n;
    pid_t childpid;
    for (i = 1; i < 4; ++i)
        if ((childpid = fork()) == -1)
            break;
    fprintf(stderr, "This is process %ld with parent %ld\n",
            (long)getpid(), (long)getppid());
    sleep(1);
}
```

wait Function

SYNOPSIS

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

```
pid_t wait (int *stat_loc);
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

POSIX

pid_t wait(int *stat_loc)

- Causes caller to pause until a child terminates, or stops until the caller receives a signal.
- If wait returns because a child terminates, the return value (of type pid_t) is positive and is the pid of that child.
- Otherwise wait returns -1 and sets errno.
- stat_loc is a pointer to an integer variable.
- If caller passes something other than NULL, wait stores the return status (terminate status?) of the child.
- POSIX specifies the following macros for testing the return status: WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, and WSTOPSIG.
- Child returns status by calling exit, _exit, or return.

Chain with wait

```
#include ...
void main(void) {
    int i;
    int n;
    pid_t childpid;
    int status;
    pid_t waitreturn;
    n = 4;
    for (i = 1; i < n; ++i)
        if (childpid = fork())
            break;
    while(childpid != (waitreturn = wait(&status)))
        if ((waitreturn == -1) && (errno != EINTR))
            break;
    fprintf(stderr, "I am process %ld, my parent is %ld\n",
            (long)getpid(), (long)getppid()); }
}
```

wait_r

Restarts the wait function if it is interrupted by a signal

waitpid (pid,status,options)

If the pid parameter is:

- greater than 0, wait for child with pid number
- equal to 0, wait for any child in the same process group as the caller
- equal to -1, wait for any child
- less than -1, wait for any child in the process group specified by the absolute value of pid

waitpid Status Parameter

- The second waitpid parameter is used the same way as the wait parameter (i.e., NULL or &status)
- The second parameter can be tested by status (WIF...) operators

waitpid examples

waitreturn = waitpid (apid, &status,0);

wait for the specific child apid

waitreturn = waitpid (-1, NULL, NOHANG);

wait for any child, but do not delay if a child is currently not available

waitreturn = waitpid(0, NULL,0)

wait for any child in the process group of the calling process

waitreturn = waitpid(-4828,NULL,0)

wait for any child in the process group 4828

exec1, execlp, execl

“l” – Passes command directly as a parameter in exec.

- **exec1** searches for command in fully qualified pathname passed as exec parameter or in current directory
- **execlp** uses PATH environment variable to find command
- **execl** uses environment passed as exec parameter to find command

execv, execvp,execve

“v” – Passes command as member of argument array (i.e., argv[] or makeargv[])

- **execv** searches for arg array command in fully qualified pathname passed in exec or in current directory
- **execvp** uses PATH environment variable to find arg array command
- **execve** uses environment passed as exec parameter to find arg array command

execl Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
void main(void) {
    pid_t childpid;
    int status;
    if ((childpid = fork()) == -1) {
        perror("Error in the fork");
        exit(1);
    } else if (childpid == 0) {          /* child code */
        if (execl("/usr/bin/ls", "ls", "-l", NULL) < 0) {
            perror("Exec of ls failed");
            exit(1); }
    } else if (childpid != wait(&status)) /* parent code */
        perror("A signal occurred before the child exited");
    exit(0); }
```

execvp Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
void main(int argc, char *argv[]) {
    pid_t childpid, waitreturn;
    int status;
    if ((childpid = fork()) == -1) {
        perror("The fork failed");
        exit(1);
    } else if (childpid == 0) {          /* child code */
        if (execvp(argv[1], &argv[1]) < 0) {
            perror("The exec of command failed");
            exit(1); }
    } else                               /* parent code */
        while(childpid != (waitreturn = wait(&status)))
            if ((waitreturn == -1) && (errno != EINTR))
                break;
    exit(0); }
```

Attributes Preserved by Calls to exec

Preserved Attribute	Relevant System Call
Process ID	getpid()
Parent process ID	getppid()
Process group ID	getpgid()
Session ID	getsid()
Real user ID	getuid()
Real group ID	getgid()
Supplementary group IDs	getgroups()
Time left on alarm signal	alarm()
Current working directory	getcwd()
Root directory	
File mode creation mask	unmask()
File size limit*	ulimit()
Process signal mask	sigprocmask()
Pending signals	sigpending()
Time elapsed	times() Continued on next slide

exec Attributes (Continued)

Preserved Attribute	Relevant System Call
resource limits*	getrlimit(), setrlimit()
controlling terminal*	open(), tcgetpgrp()
interval timers*	ualarm()
nice values*	nice()
semadj values*	semop()

An * indicates an attribute that is inherited in the POSIX:XSI Extension

Background Processes

- Child process becomes background process when it executes `setsid()`.
- Child that becomes background process never returns to parent.
- Background processes cannot be interrupted with `ctrl-c`.

Daemon

A daemon is a background process that runs indefinitely.

Examples:

- Solaris 2 pageout daemon
- Mailer daemon

Background Processes

...

```
childpid = fork());
  if (childpid == -1) {
    perror("The fork failed");
    return 1 }
  if (childpid == 0) { /* child becomes a background process */
    if (setsid() == -1)
      perror("Could not become a session leader");
  else {
    execvp(argv[1], &argv[1]);
    perror("Child failed to exec command"); }
    return 1; } /* child should never return */
  return 0 } /* parent exits */
```