Unix processes and threads

Henning Schulzrinne
Dept. of Computer Science
Columbia University

Unix processes and threads

What's a process?
- Fundamental to almost all operating systems
- = program *in execution*
- address space, usually separate
- program counter, stack pointer, hardware registers
- simple computer: one program, never stops

What's a process?
- timesharing system: alternate between processes, interrupted by OS:
  - run on CPU
  - clock interrupt happens
  - save process state:
    - registers (PC, SP, numeric)
    - memory map
    - memory (core image) possibly swapped to disk
  - process table
  - continue some other process

Process relationships
- process tree structure: child processes
- inherit properties from parent
- processes can:
  - terminate
  - request more (virtual) memory
  - wait for a child process to terminate
  - overlay program with different one
  - send messages to other processes

Processes
- Reality: each CPU can only run one program at a time
- Fiction to user: many people getting short (~10-100 ms) time slices
  - pseudo-parallelism \(\rightarrow\) multiprogramming
  - modeled as sequential processes
  - context switch
Process creation

- Processes are created:
  - system initialization
  - by another process
  - user request (from shell)
  - batch job (timed, Unix `at` or `cron`)
- Foreground processes interact with user
- Background processes (daemons)

Unix processes

- 0: process scheduler ("swapper")
  - system process
- 1: init process, invoked after bootstrap
  - `/sbin/init`

Unix process creation: forking

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

pid_t fork(void);

int v = 42;
if ((pid = fork()) < 0) {
    perror("fork");
    exit(1);
} else if (pid == 0) {
    printf("child %d of parent %d\n",
           getpid(), getppid());
    v = 0;
} else sleep(30);
```

Processes - example

- task manager in Windows NT, 2000 and XP
- cooperative vs. preemptive

fork()

- called once, returns twice
- child: returns 0
- parent: process ID of child process
- both parent and child continue executing after fork
- child is clone of parent (copy[])
- copy-on-write: only copy page if child writes
- all file descriptors are duplicated in child
  - including file offset
- network servers: often child and parent close unneeded file descriptors
User identities

- Who we really are: real user group ID
- taken from /etc/passwd file:
  
  https://gus.191191191:92/n, schulz@home/hgpu/bin/tcsh

- Check file access permissions: effective user group ID, supplementary group ID
  
  supplementary IDs via group membership: /etc/group
  
  special bits for file: "when this file is executed, set the effective IDs to be the owner of the file" → 
  set-user-ID bit, set-group-ID bit
  
  */usr/bin/passwd needs to access password files

Aside: file permissions

<table>
<thead>
<tr>
<th>S_IUSR</th>
<th>user-read</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IWUSR</td>
<td>user-write</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>user-execute</td>
</tr>
<tr>
<td>S_IGRP</td>
<td>group-read</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>group-write</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>group-execute</td>
</tr>
<tr>
<td>SIROTH</td>
<td>other-read</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>other-write</td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>other-execute</td>
</tr>
</tbody>
</table>

Process identifiers

| pid_t getpid(void) | process identifier |
| pid_t getpgid(pid_t pid); | process group |
| pid_t getpgid(void); | parent PID |
| uid_t geteuid(void); | real user ID |
| gid_t getegid(void); | effective user ID |
| gid_t getgid(void); | real group ID |
| gid_t getegid(void); | effective group ID |

Process properties inherited

- user and group ids
- process group id
- controlling terminal
- setuid flag
- current working directory
- root directory (chroot)
- file creation mask
- signal masks
- close-on-exec flag
- environment
- shared memory
- resource limits

Differences parent-child

- Return value of fork()
- process IDs and parent process IDs
- accounting information
- file locks
- pending alarms

Waiting for a child to terminate

- asynchronous event
- SIGHUP signal
- process can block waiting for child termination

  pid = fork();
  ...
  if (wait(&status) != pid) {
      something's wrong
  }
Waiting for a child to terminate

```
pid_t waitpid(pid_t pid, int *status, int options)
```

- `pid` -1: any child process
- `pid` 0: any child with process group id
- `pid` > 0: any child with `PID = abs(pid)`

Race conditions

- `race` = shared data
- `outcome` depends on order that processes run
- e.g., parent or child runs first
- waiting for `parent` to terminate
- generally, need some signaling mechanism
  - signals
  - stream pipes

exec: running another program

- replace current process by new program
  - text, data, heap, stack
  ```
  int exec(const char *path, char *arg, ...
  int exec(const char *path, const char *arg0, ...
  int exec(const char *path, char *const envp[])
  int exec(char *file, char *const argv[])
  ```
  - `file`: absolute path or one of the PATH entries

exec example

```c
char *envp_init[] = {"USER=unknown", "PATH=/tmp", NULL};
int main(void) {
  pid_t pid;
  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) {
    if (execl("echoall", "echoall", "myarg1", "My ARG2", NULL, envp_init) < 0)
      perror("exec");
    if (waitpid(pid, NULL, 0) < 0) perror("wait error");
    printf("child done\n");
    exit(0);
  }
}
```

system: execute command

```c
#include <stdlib.h>
int system(const char *string);
```

- invokes command string from program
- e.g., `system("date > file");`
- handled by shell (`ls`/`Bin/bash`) 
- never call from setuid programs

Threads

- `process`: address space
- single thread of control
- sometimes want multiple threads of control (flow) in same address space
- quasi-parallel
- threads separate resource grouping
- `execution`
- `thread`: program counter, registers, stack
- aka lightweight processes
- multithreading: avoid blocking when waiting for resources
  - multiple services running in parallel
- state: running, blocked, ready, terminated
Why threads?

- Parallel execution
- Shared resources → faster communication without serialization
- Easier to create and destroy than processes (100x)
- Useful if some are I/O-bound → overlap computation and I/O
- Easy porting to multiple CPUs

Creating a thread

```
int pthread_create(pthread_t *tid, const
pthread_attr_t *, void *(*func)(void *), void *arg);
```
- Start function `func` with argument `arg` in new thread
- Return 0 if ok, -1 if not
- Careful with `arg` argument

Network server example

- Lots of little requests (hundreds to thousands a second)
- Simple model: new thread for each request → doesn’t scale (memory, creation overhead)
- Dispatcher reads incoming requests
- Picks idle worker thread and sends it message with pointer to request
- If thread blocks, another one works on another request
- Limit number of threads

Worker thread

```
while (1) *
    wait for work(buf) *
    look in cache
    if not in cache
        read page from disk
    return page
```

Leaving a thread

```
threads can return value, but typically NULL
```
- Just return from function (return `void *`)
- Main process exits → kill all threads
- `pthread_exit(void *status)`
Thread synchronization

- mutual exclusion, locks: mutex
- protect shared or global data structures
- synchronization: condition variables
- semaphores