Unix processes and threads

Henning Schulzrinne
Dept. of Computer Science
Columbia University

Unix processes and threads

- Process model
  - creation
  - properties
  - owners and groups
- Threads
  - threads vs. processes
  - synchronization

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 → *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)

Processes - example

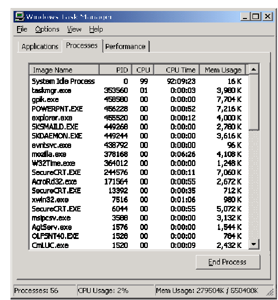
```
bart:~> ps -ef
     UID   PID  PPID  C    STIME TTY      TIME CMD
    root     0     0  0   Mar 31 ?        0:17 sched
    root     1     0  0   Mar 31 ?        0:09 /etc/init -
    root     2     0  0   Mar 31 ?        0:00 pageout
    root     3     0  0   Mar 31 ?       54:35 fsflush
    root   334     1  0   Mar 31 ?        0:00 /usr/lib/saf/sac -t 300
    root 24695     1  0 19:38:45 console  0:00 /usr/lib/saf/ttymon
    root   132     1  0   Mar 31 ?        1:57 /usr/local/sbin/sshd
    root   178     1  0   Mar 31 ?        0:01 /usr/sbin/inetd -s
  daemon    99     1  0   Mar 31 ?        0:00 /sbin/lpd
    root   139     1  0   Mar 31 ?        0:37 /usr/sbin/rpcbind
    root   119     1  0   Mar 31 ?        0:06 /usr/sbin/in.rdisc -s
    root   142     1  0   Mar 31 ?        0:00 /usr/sbin/keyserv
     hgs  2009  2007  0 12:58:13 pts/16   0:00 -tcsh
  daemon   182     1  0   Mar 31 ?        0:00 /usr/lib/nfs/statd
    root   152     1  0   Mar 31 ?        0:00 /yp/ypbind -broadcast
```


Unix processes

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

Processes - example

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



Unix process creation: forking

```c
#include <sys/types.h>
#include <unistd.h>
pid_t  fork(void);
```

```c
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++;
} else sleep(10);
```


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:

    `hgs:7C6uo:5815:92:H. Schulzrinne:/home/hgs:/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

`S_IRUSR`	user-read
`S_IWUSR`	user-write
`S_IXUSR`	user-execute
`S_IRGRP`	group-read
`S_IWGRP`	group-write
`S_IXGRP`	group-execute
`S_IROTH`	other-read
`S_IWOTH`	other-write
`S_IXOTH`	other-execute

Process identifiers

`pid_t getpid(void)`	process identifier
`pid_t getpgid(pid_t pid);`	process group
`pid_t getppid(void);`	parent PID
`uid_t getuid(void);`	real user ID
`uid_t geteuid(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
- SIGCHLD 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
  *statloc, int options)
```

pid=-1	any child process
pid>0	specific process
pid=0	any child with some 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 execl(const char *path, char *arg, ...);
int execle(const char *path, const char *arg0,
   /* (char *) 0, char *const envp[] */);
int execv(const char *path, char *const arg[]);
int execvp(char *file, char *const argv[]);
```

`file`: /absolute/path or one of the PATH entries

exec example

```
char *env_init[] = {"USER=unknown", "PATH=/tmp",
   NULL};

int main(void) {
  pid_t pid;
  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) {
    if (execle("echoall", "echoall", "myarg1",
      "MY ARG2", NULL, env_init) < 0)
      perror("exec");
  }
  if (waitpid(pid, NULL, 0) < 0) perror("wait error");
  printf("child done\n");
  exit(0);
}
```


system: execute command

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

- invokes command string from program
- e.g., `system("date > file");`
- handled by shell (/usr/bin/ksh)
- 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
- also called 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

Thread variants

- POSIX (pthreads)
- Sun threads (mostly obsolete)
- Java threads

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, >0 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