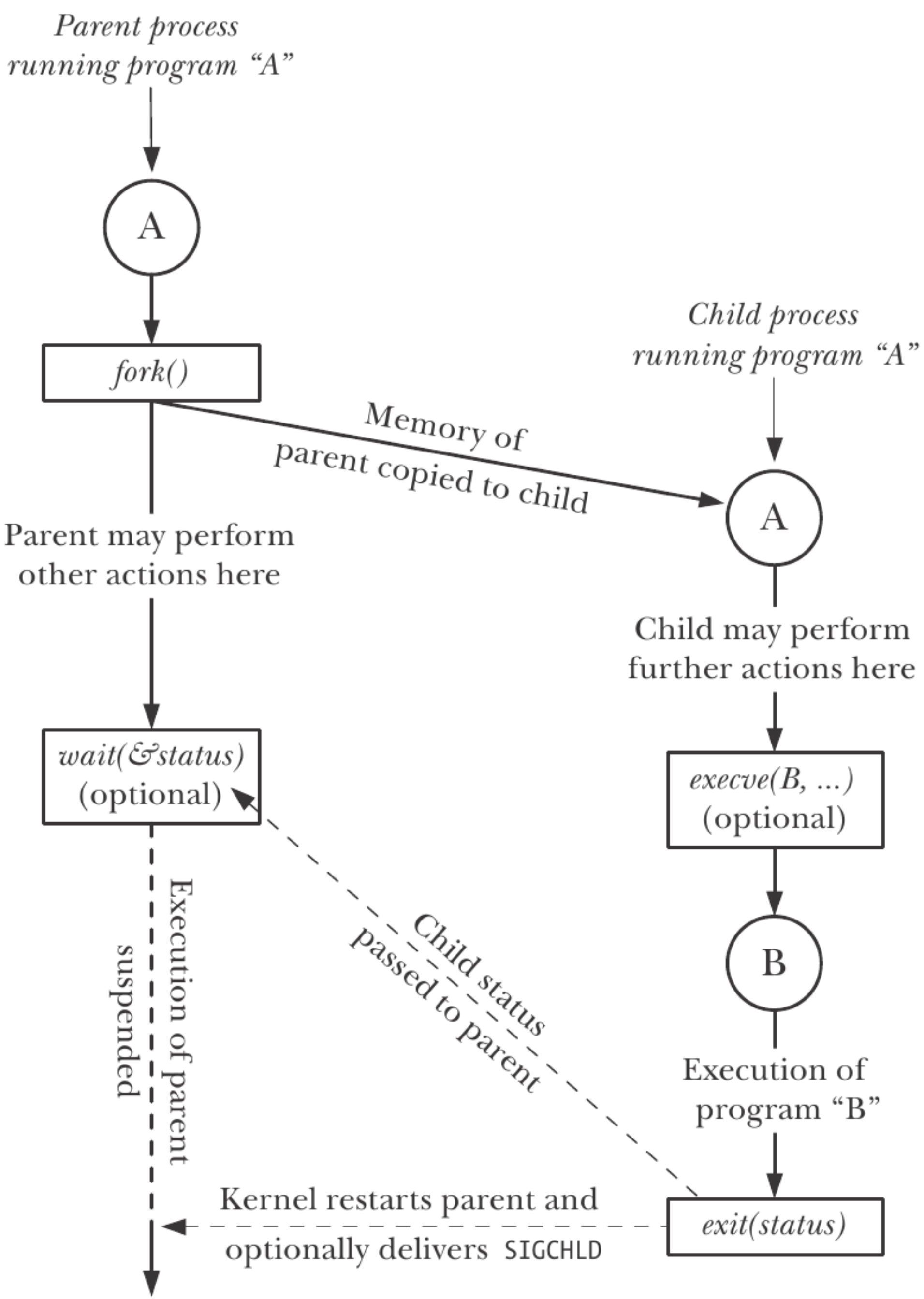
Chapter 8.
Process Control

朱金辉

华南理工大学软件学院

1. Introduction

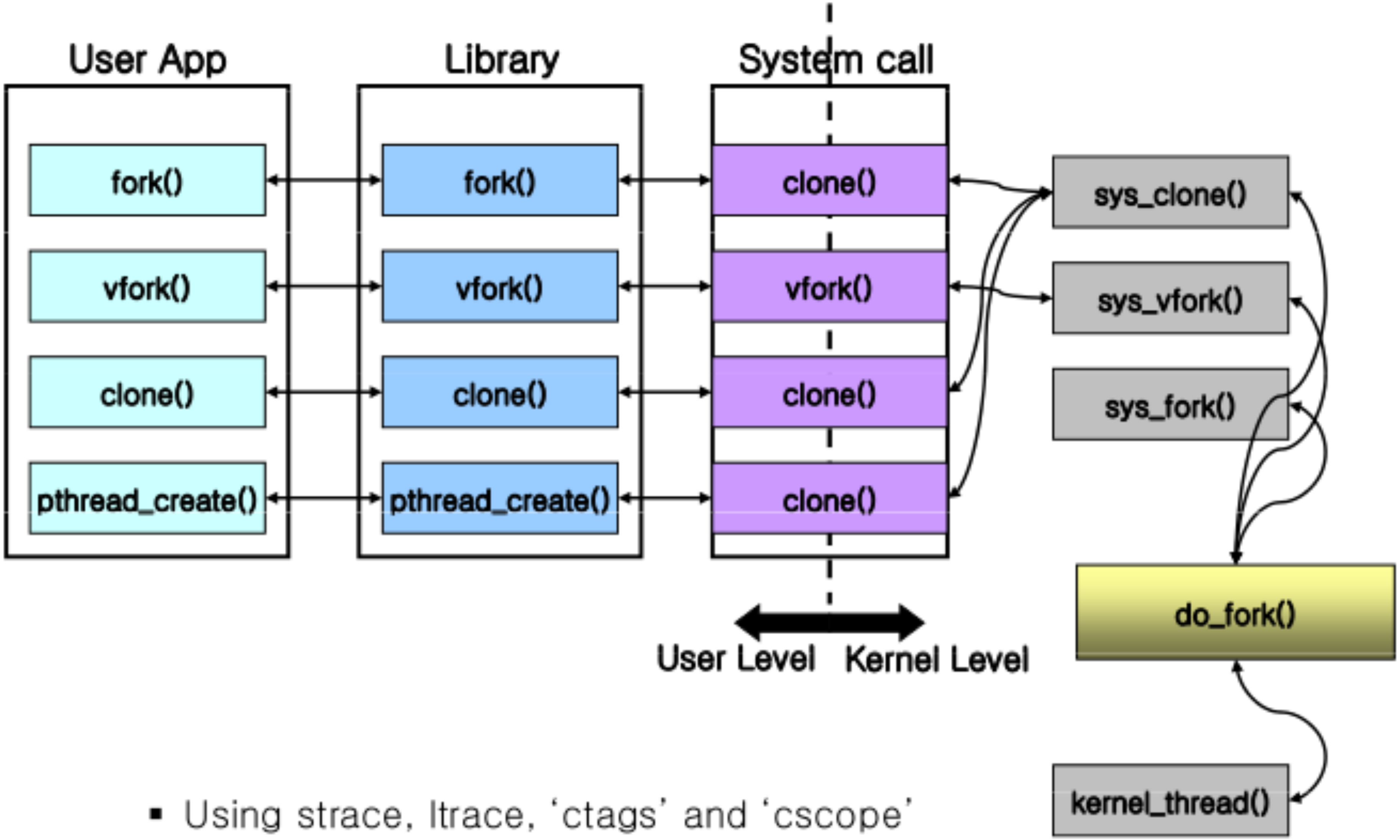
- Creation of new processes (fork)
- Process termination
- Executing programs (exec)
- IDs
- system()
- Process accounting



Overview of the use of fork(), exit(), wait(), and execve()

Fork API

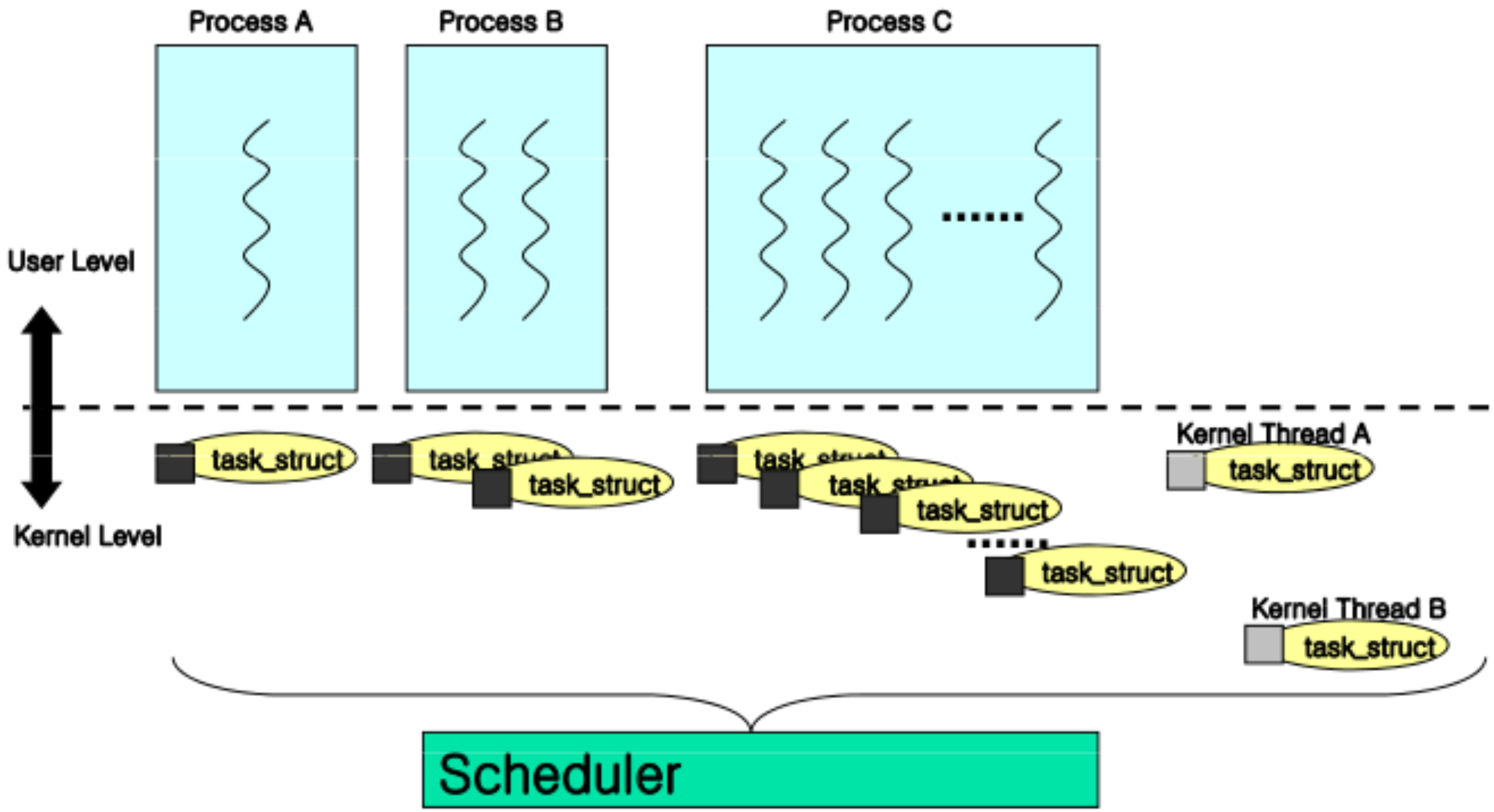
- Flow controls



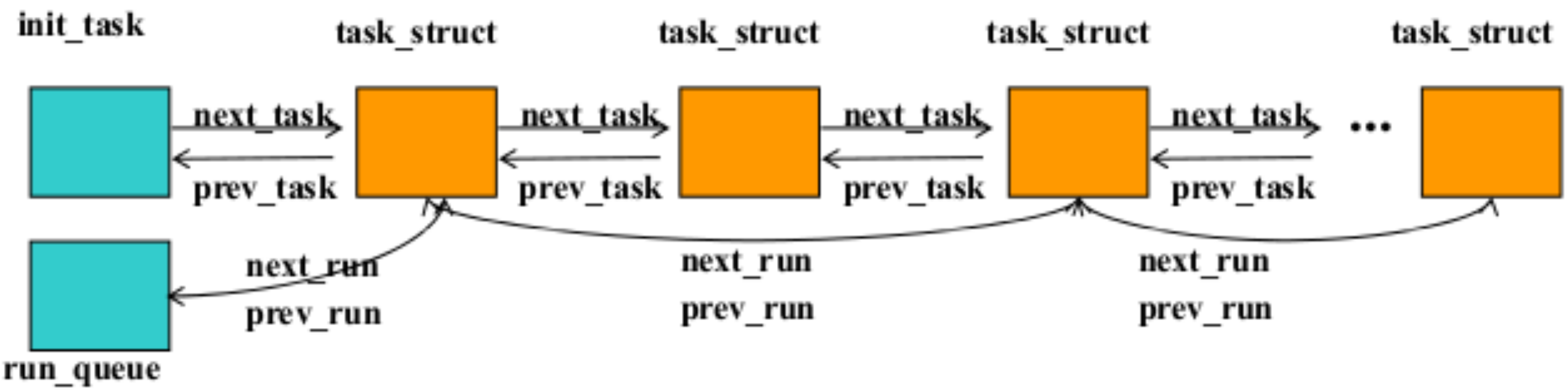
- Using strace, ltrace, 'ctags' and 'cscope'

Linux task model

- Internal structure

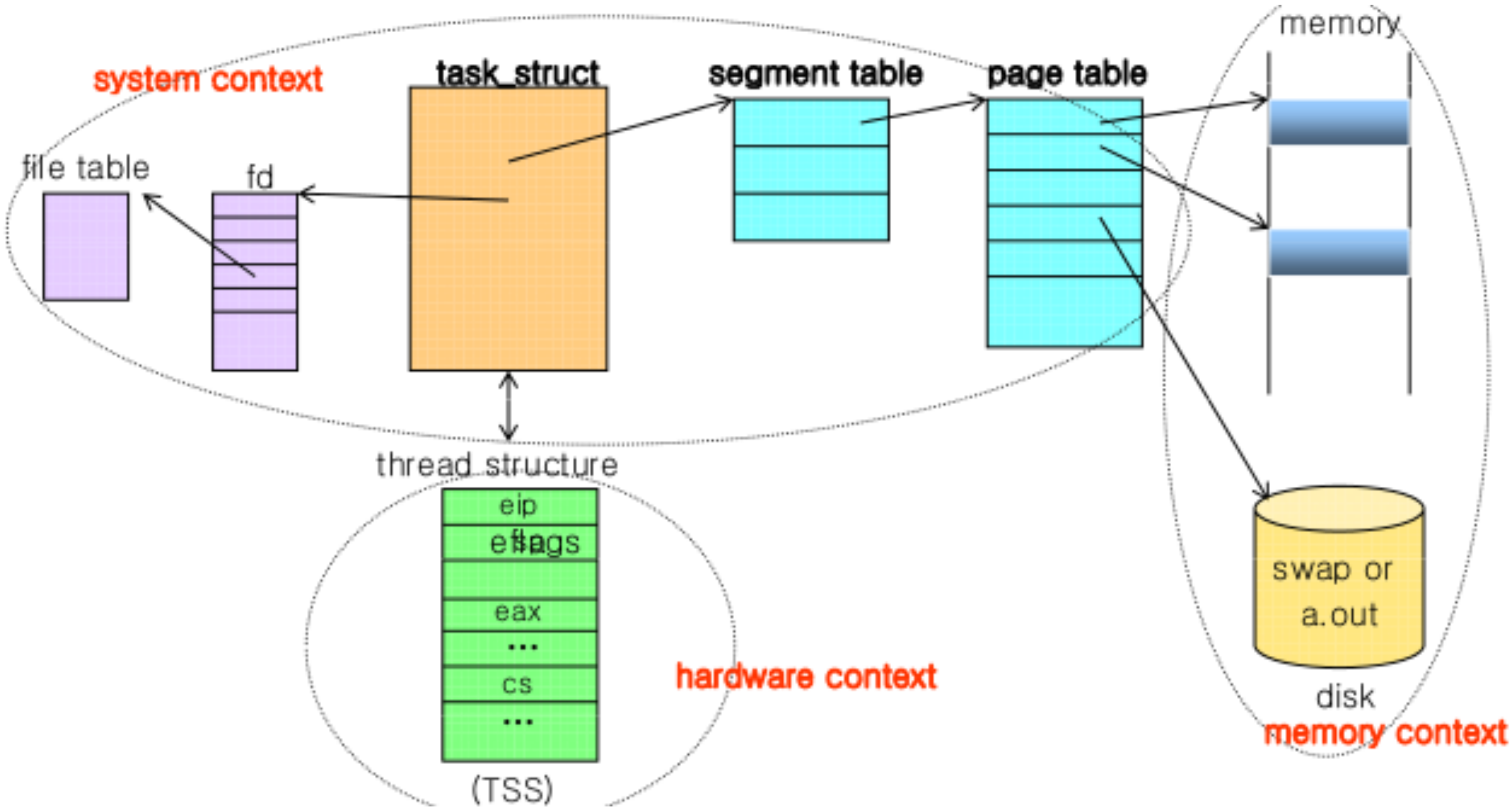


Task queue

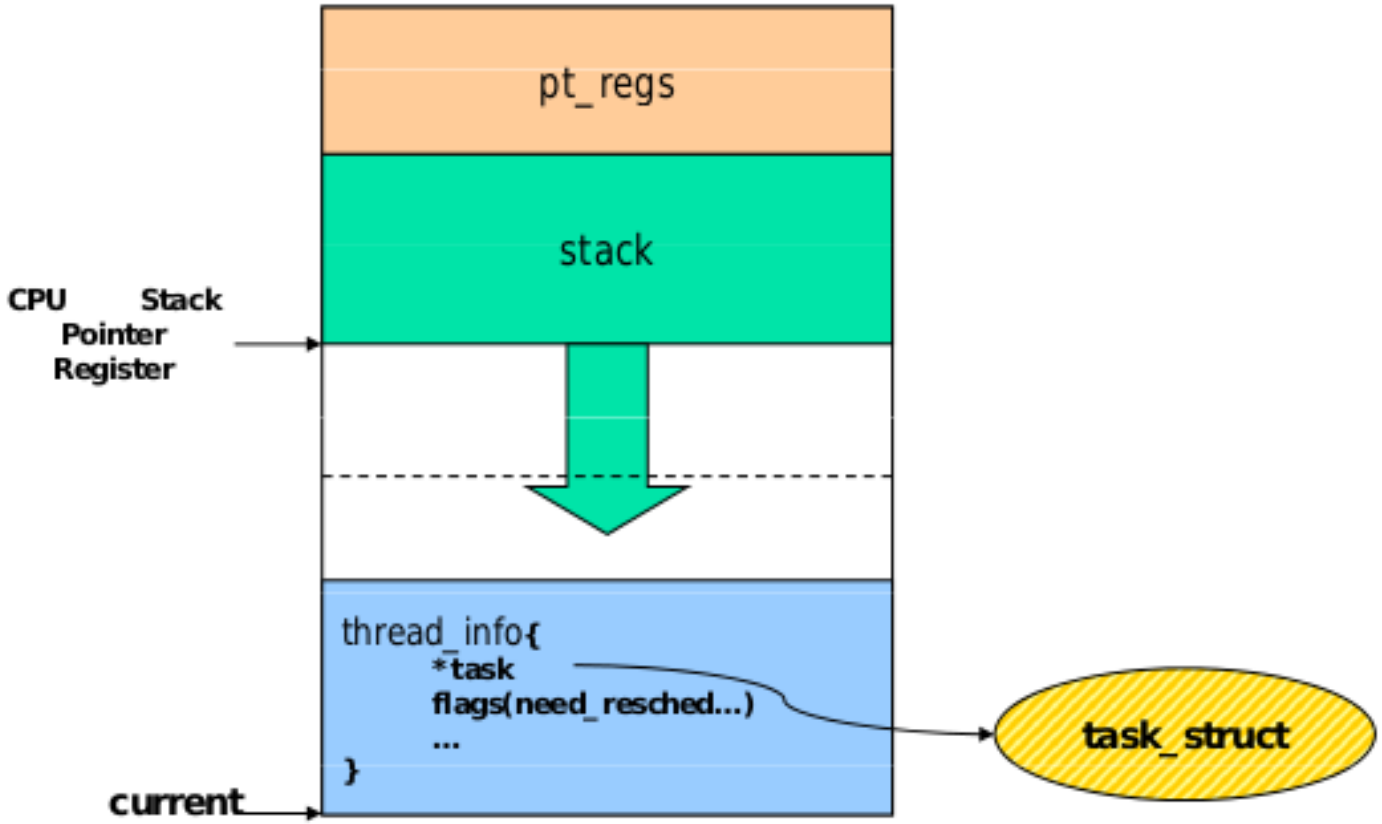


init_task
task_struct
task_struct
task_struct
task_struct
next_task
prev_task
next_task
prev_task
next_task
prev_task
next_task
prev_task
...
next_run
prev_run
next_run
prev_run
next_run
prev_run
run_queue

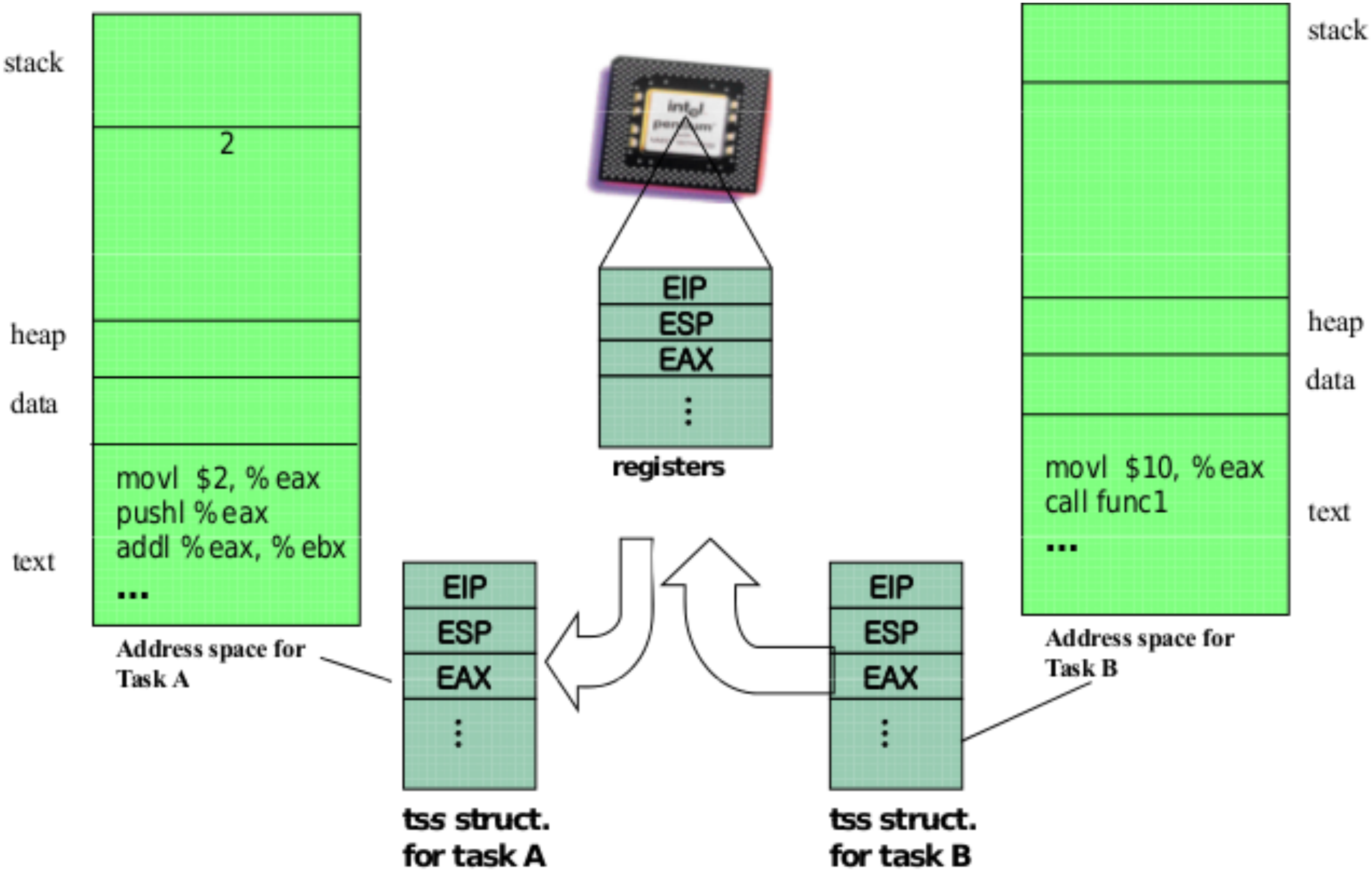
Task context



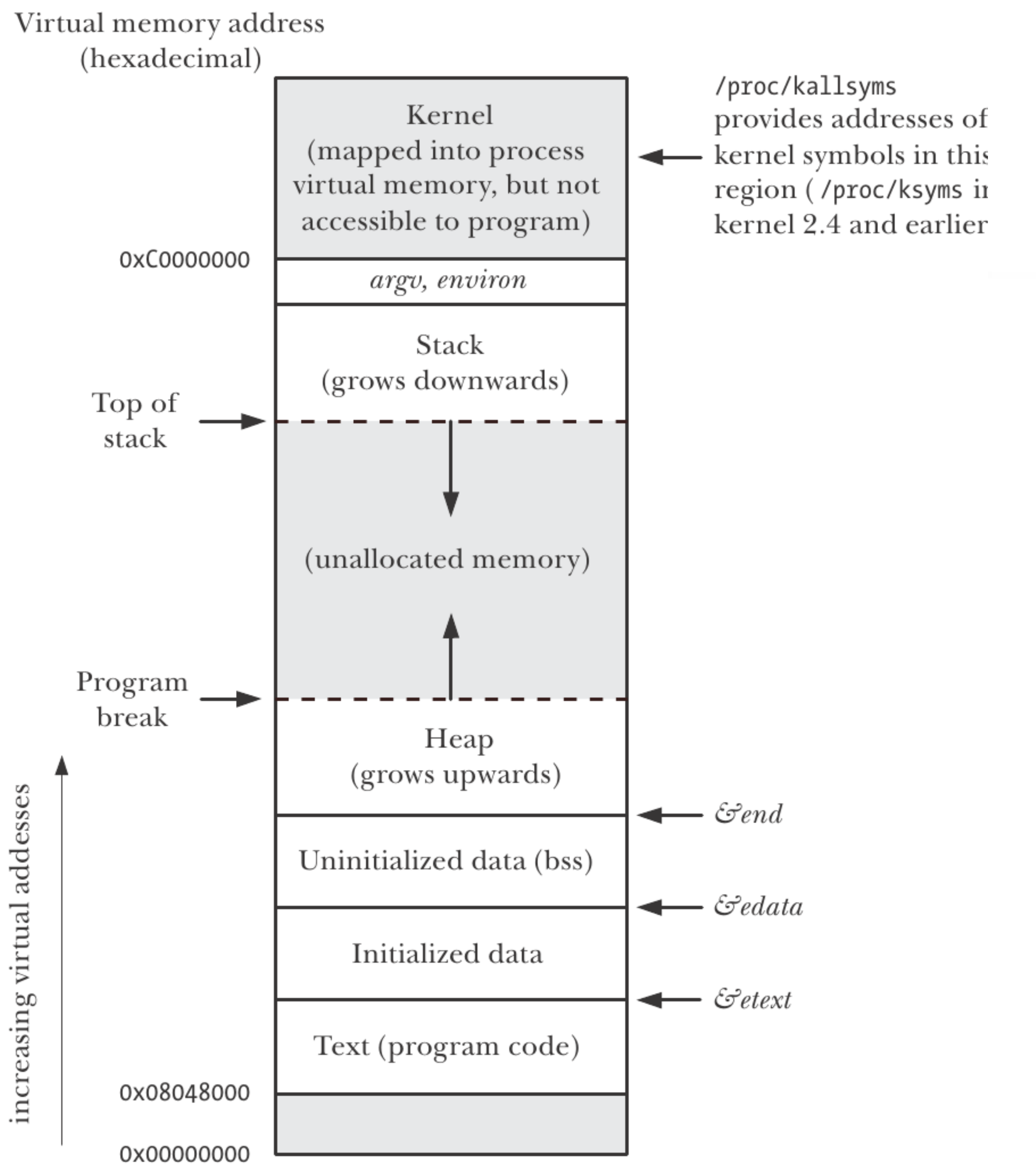
Kernel stack



Hardware context switch

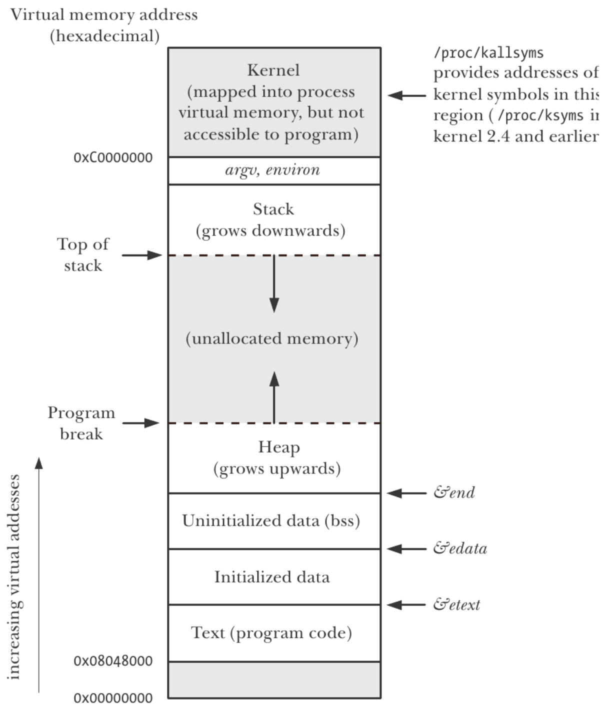


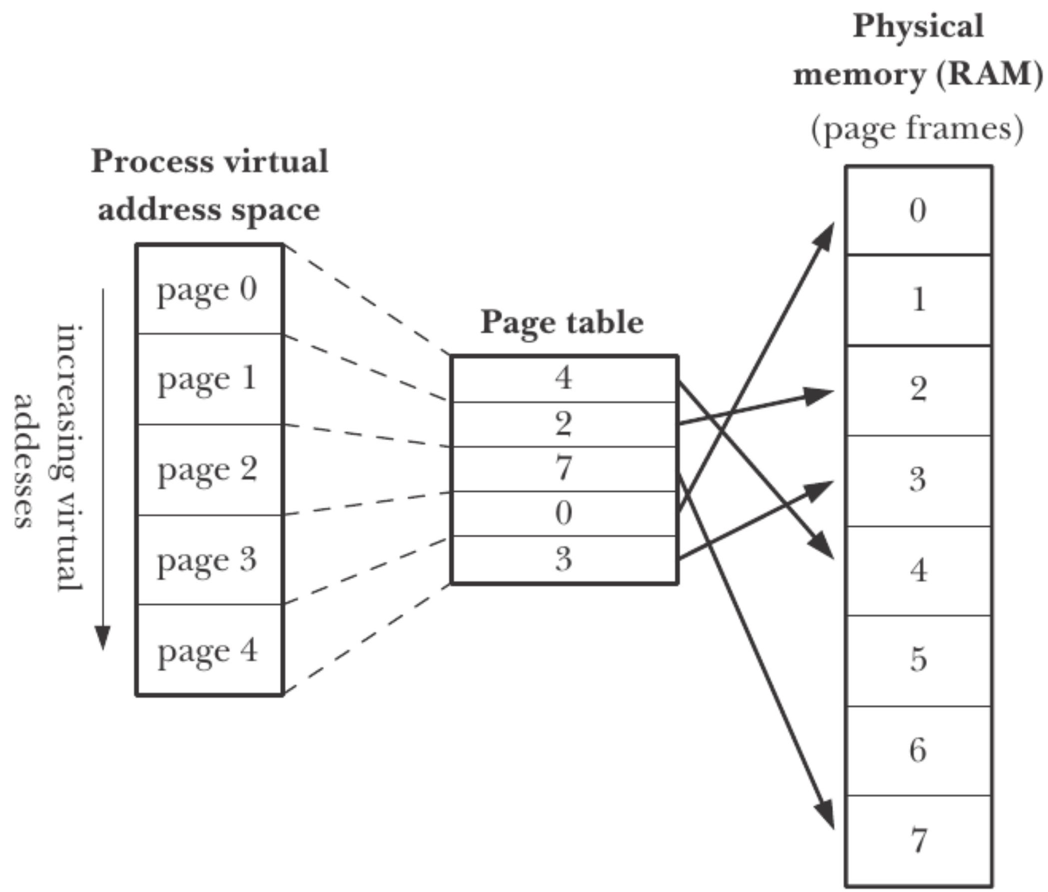
Virtual Memory



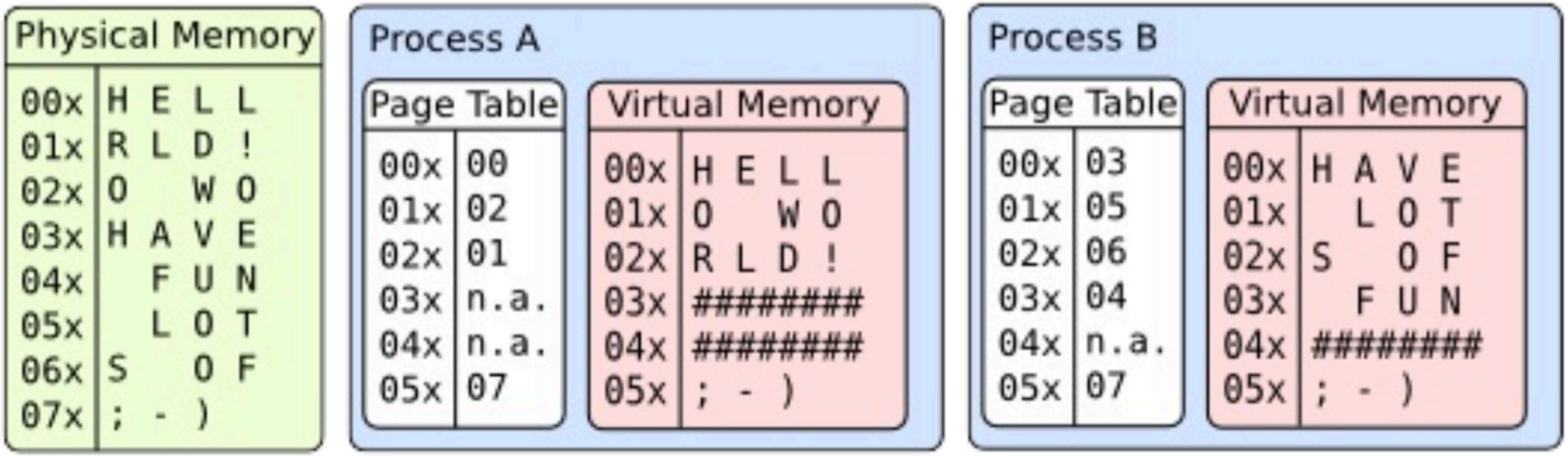
Typical memory layout of a process on Linux/x86-32

Virtual mem & Physical mem



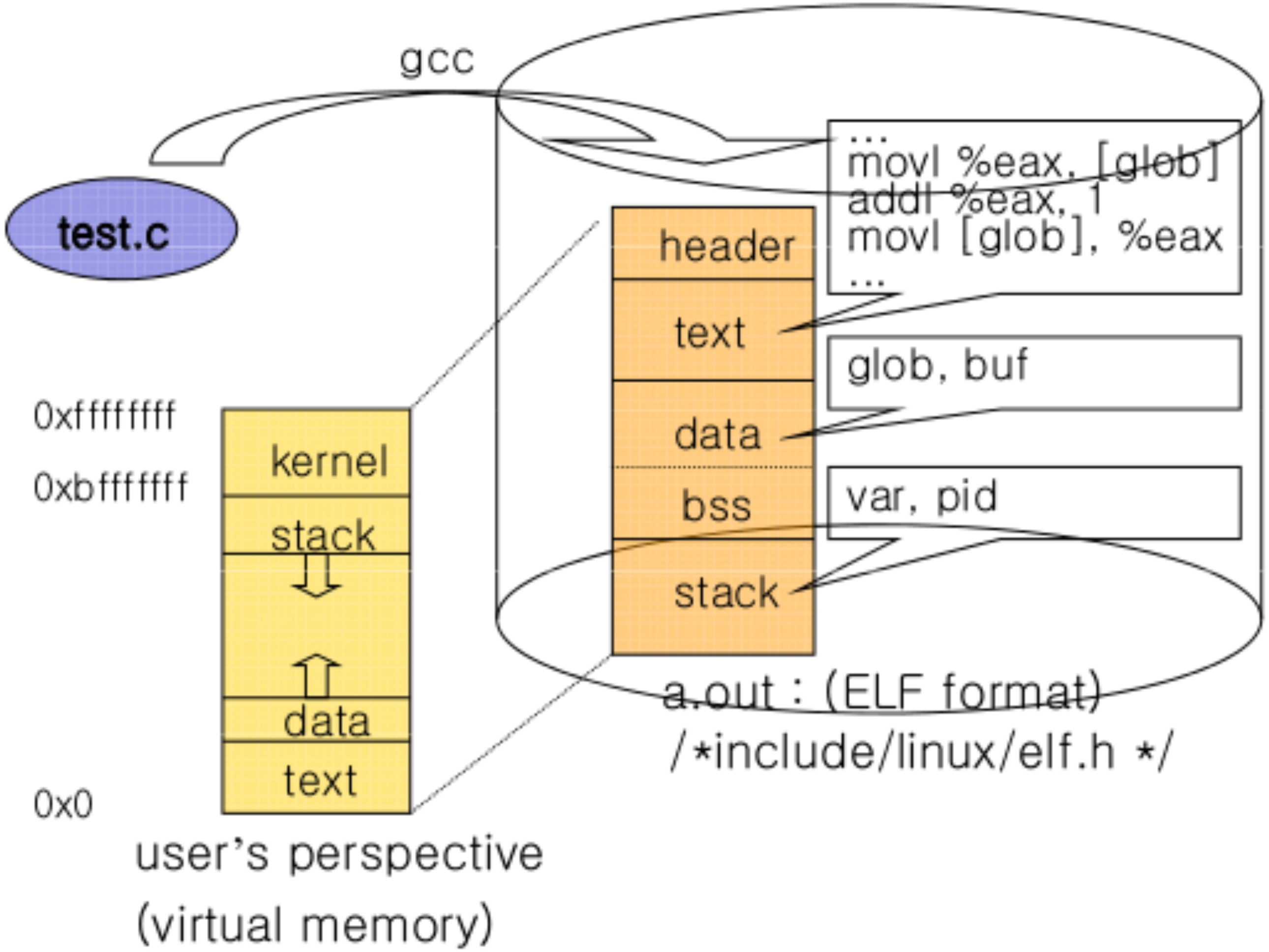


Example

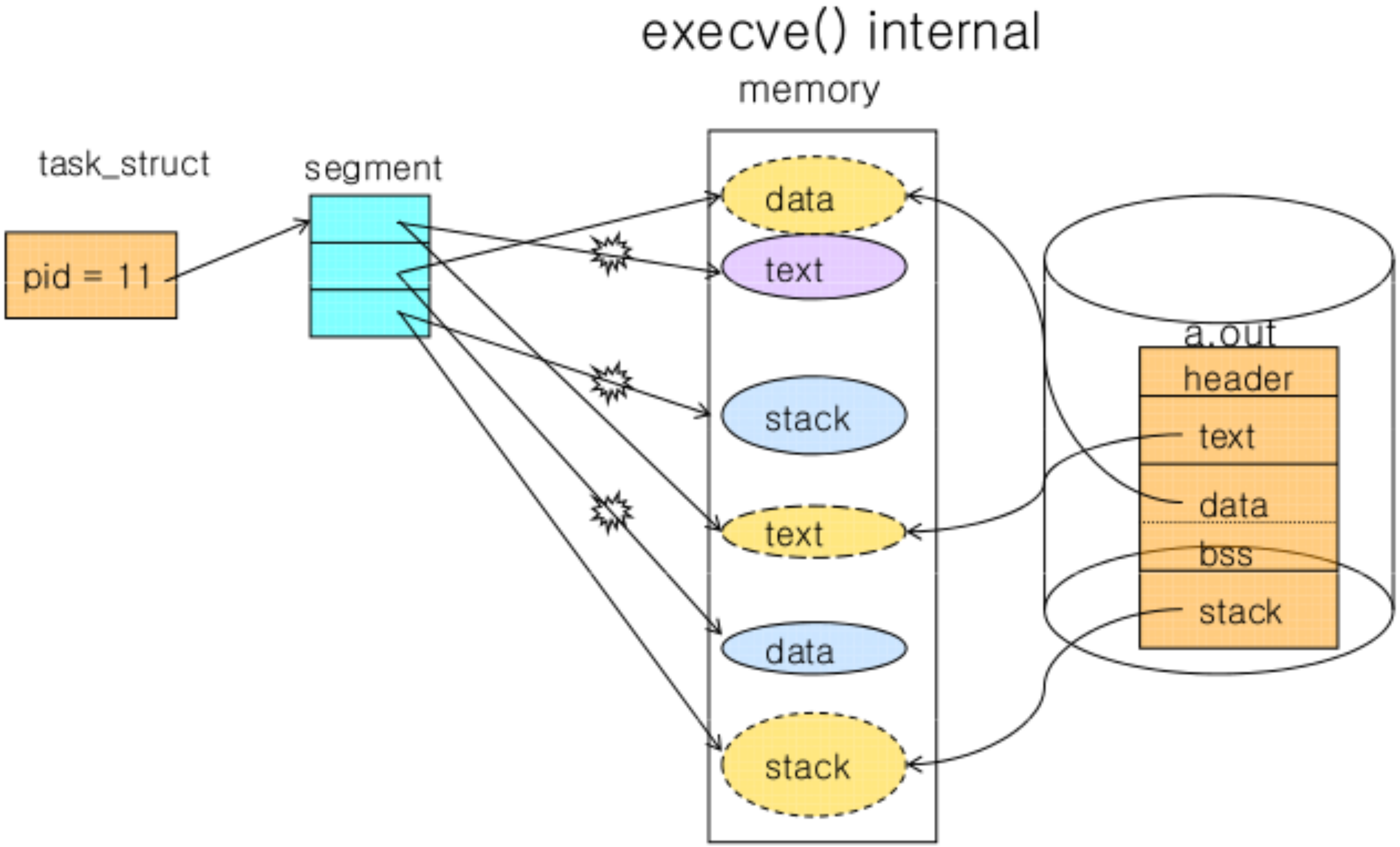

Physical Memory
00x H E L L
01x R L D !
02x O W O
03x H A V E
04x F U N
05x L O T
06x S O F
07x ; -)
Process A
Page Table
00x 00
01x 02
02x 01
03x n.a.
04x n.a.
05x 07
Virtual Memory
00x H E L L
01x O W O
02x R L D !
03x ########
04x ########
05x ; -)
Process B
Page Table
00x 03
01x 05
02x 06
03x 04
04x n.a.
05x 07
Virtual Memory
00x H A V E
01x L O T
02x S O F
03x F U N
04x ########
05x ; -)

execve

- fork internal : compile results



execve



execve() internal
memory
task_struct
segment
pid = 11
data
text
stack
text
data
stack
a.out
header
text
data
bss
stack

2. Process Identifiers

- Process ID = a nonnegative integer

PID	Process
0	swapper (scheduler)
1	init (/sbin/init)
2	pagedaemon (virtual memory paging)
3, 4, …	other processes

Identifier functions

- #include <sys/types.h>
- #include <unistd.h>
- pid_t getpid(void); return PID
- pid_t getppid(void); return parent PID
- uid_t getuid(void); return real UID
- uid_t geteuid(void); return effective UID
- gid_t getgid(void); return real GID
- gid_t getegid(void); return effective GID

3. fork Function

- fork() is the ONLY way to create a process in Unix kernel by user

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

- Returns: 0 in child, child PID in parent, -1 on error

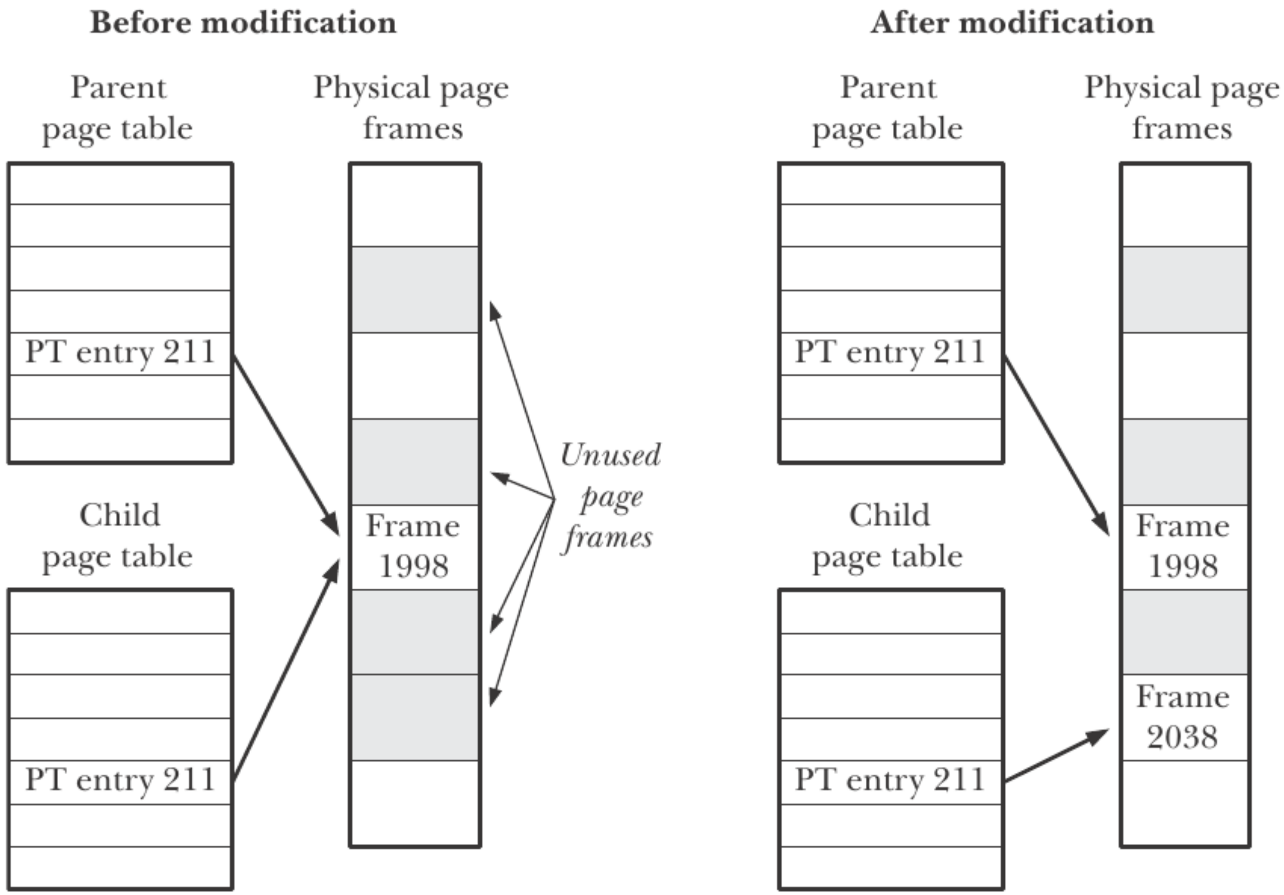
Parent / Child Processes

- Parent and child continue executing instructions following the fork() call
- Child gets a copy of parent's data space, heap, and stack
- Often, read-only text segment is shared
- Often, fork() is followed by exec()
- Waste of space and time for setting up child' s program space!!!

Copy-On-Write (COW)

- Memory regions are read-only and shared by parent and child
- If either process wants to write, kernel makes a copy of that memory only for that process.
- Saves space and time!

COW example



Page tables before and after modification of a shared copy-on-write page

Program 8.1: fork()

```
#include <sys/types.h>
#include "apue.h"
int glob = 6;          /* external variable in initialized data */
char     buf[] = "a write to stdout\n";
int main(void) {
    int    var;        /* automatic variable on the stack */
    pid_t pid;
    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n"); /* we don't flush stdout */
    if ( (pid = fork()) < 0)      err_sys("fork error");
    else if (pid == 0) {                    /* child */
     glob++;                                /* modify variables */
     var++;
    } else
     sleep(2);                              /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

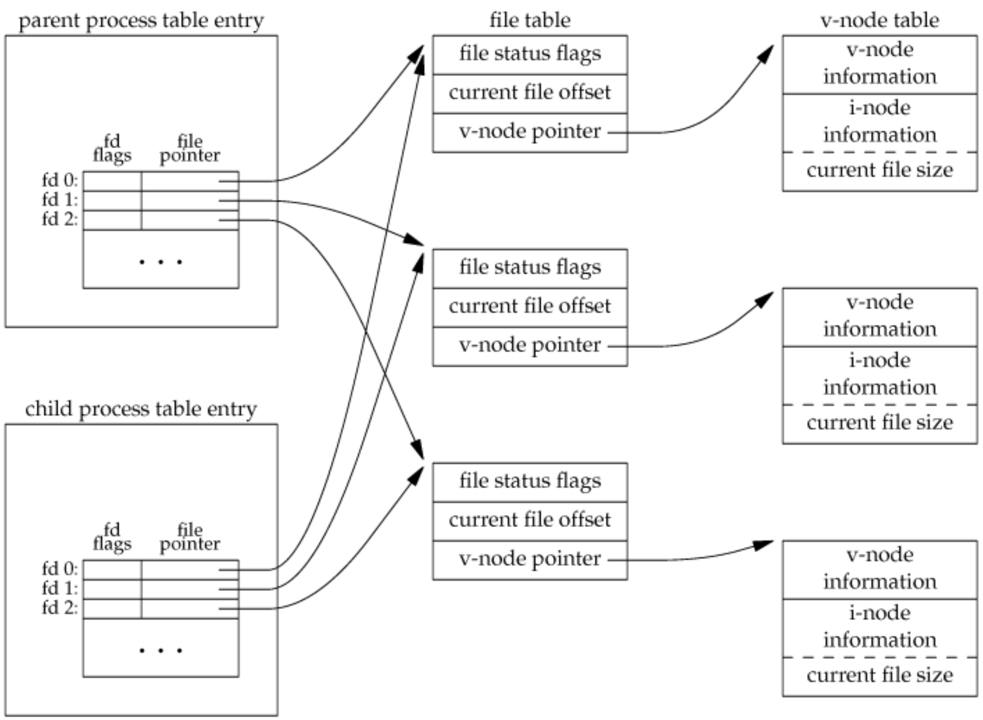
Program 8.1: results

$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88

$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88

File Sharing

- Parent and child share the same file descriptors
- Parent and child share the same file offset, otherwise overwrite
- Intermixed output from parent and child



4. vfork Function

- Creates a new process only to 'exec' a new program
- No copy of parent's address space for child (not needed!)
- Before exec, child runs in "address space of parent"
- Efficient in paged virtual memory
- Child runs first
- Parent waits until child 'exec' or 'exit'

Program 8.3: vfork()

```
#include <sys/types.h>
#include "apue.h"

int     glob = 6;        /* external variable in initialized data */

int main(void) {
    int          var;    /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    printf("before vfork\n");       /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    else if (pid == 0) {            /* child */
        glob++;                     /* modify parent's variables */
        var++;
        _exit(0);                   /* child terminates */
    }
    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Program 8.3: results

- $ a.out

  before vfork

  pid = 607, glob = 7, var = 89
- increments by child appear in parent address space

- Instead of _exit() → exit(), results in:

  $ a.out

  before vfork

stdio closed by child

5. Child Termination

Termination status:

- normal: exit status
- abnormal: kernel indicates reason

- What if child terminates before parent?
  - Child returns termination status to parent
- What if parent terminates before child?
  - Parent PID (of orphaned child) = 1 (init)

SIGCHLD

- Child terminates →
  Kernel sends SIGCHLD signal to parent
- Default action for SIGCHLD signal: ignore it
- Signal handlers can be defined by users

(Chapter 10)

6. wait(), waitpid()

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

#include <sys/wait.h>

pid_t wait(int *statloc);

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

block wait for any one child to terminate

place for storing termination status
NULL→no need!

- Return: PID if OK, 0, -1 on error

wait3 and wait4

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options,
                    struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options,
                    struct rusage *rusage);
```

resource information for terminated processes

- Return: PID if OK, 0 or -1 on error

Termination Status Macros

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Program 8.5: print exit status

```
#include <sys/types.h>
#include <sys/wait.h>
#include "apue.h"

void pr_exit(int status) {
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                                WTERMSIG(status),
#ifdef  WCOREDUMP
        WCOREDUMP(status) ? " (core file generated)" : "");
#else
        "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                                WSTOPSIG(status));
}
```

Program 8.6: demo exit status

```
#include      <sys/types.h>
#include      <sys/wait.h>
#include      "apue.h"

int main(void) {
   pid_t      pid;
   int        status;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0)          /* child */
        exit(7);
  if (wait(&status) != pid)   /* wait for child */
        err_sys("wait error");
   pr_exit(status);            /* and print its status */
```

Program 8.6 (II Part)

```
if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0)     /* child */
    abort();           /* generates SIGABRT */

if (wait(&status) != pid) /* wait for child */
    err_sys("wait error");
pr_exit(status);   /* and print its status */
```

Program 8.6 (III part)

```
if ( (pid = fork()) < 0)
     err_sys("fork error");
 else if (pid == 0)                /* child */
     status /= 0; /* divide by 0 generates SIGFPE */

 if (wait(&status) != pid)         /* wait for child */
     err_sys("wait error");
 pr_exit(status);          /* and print its status */

 exit(0);
}
```

Program 8.6: results

- **$ a.out**
- normal termination, exit status = 7
- abnormal termination, signal number = 6 (core file generated)

SIGABRT

- abnormal termination, signal number = 8 (core file generated)

SIGFPE

Zombie process

- Suppose child terminates first && parent don’t wait child
- Zombie: minimal info of dead child process (pid, termination status, CPU time)

Avoiding zombie processes

- A process forks a child
- It does not wait for the child to complete
- It does not want child to become zombie
- How to do this?
- Answer: fork twice! (Program 8.8)

Program 8.8: Avoid Zombie

```
int main(void) {
   pid_t      pid;
   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {               /* first child */
        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid > 0) /* parent from second fork */
                exit(0);   /*  == first child       */
        /* second child; parent becomes init */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
   }
   if (waitpid(pid, NULL, 0) != pid)/* wait for first child */
        err_sys("waitpid error");

   /* We're the parent (the original process) */
    …
   exit(0);
}
```

Program 8.8: results

- **$ a.out**
- second child, parent pid = 1

original process
terminates here

9. Race Conditions

- Multiple processes share some data
- Outcome depends on the order of their execution (i.e. RACE)
- After fork(), we cannot predict if the parent or the child runs first!
- The order of execution depends on:
  - system load
  - Kernel's scheduling algorithm

Program 8.12: Race Condition

```
#include <sys/types.h>
#include "apue.h"

static void charatatime(char *);

int main(void) {
    pid_t pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}
```

Program 8.12 (continued)

```
static void
charatatime(char *str){
    char          *ptr;
    int           c;

    setbuf(stdout, NULL);     /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

Program 8.12: results

- **$ a.out**
- output from child
- output from parent

- **$ a.out**
- ooutput from parent
- utput from child

Race Conditions

- Race condition problems are hard to detect because they work "most of the time"!

1. For parent to wait for child
   - call wait, waitpid, wait3, wait4
2. For child to wait for parent
   - **while (getppid() != 1) sleep(1);**

polling!
wastes CPU
time!

use signals or
other IPC methods

Race Conditions

- After fork
  - parent and child both need to do something on its own
  - For example, parent: write a record in a log file and child: creates a log file
- Parent and child need to:
  - **TELL** each other when its initial set of operations are done, and
  - **WAIT** for each other to complete

Program 8.13: No race condition

```
#include   <sys/types.h>
#include   "apue.h"
static void charatatime(char *);
int main(void) {
      pid_t  pid;
      TELL_WAIT();
      if ( (pid = fork()) < 0)
            err_sys("fork error");
      else if (pid == 0) {
            WAIT_PARENT();                /* parent goes first */
            charatatime("output from child\n");
      } else {
            charatatime("output from parent\n");
            TELL_CHILD(pid);
      }
      exit(0);
}
```

10. exec Functions

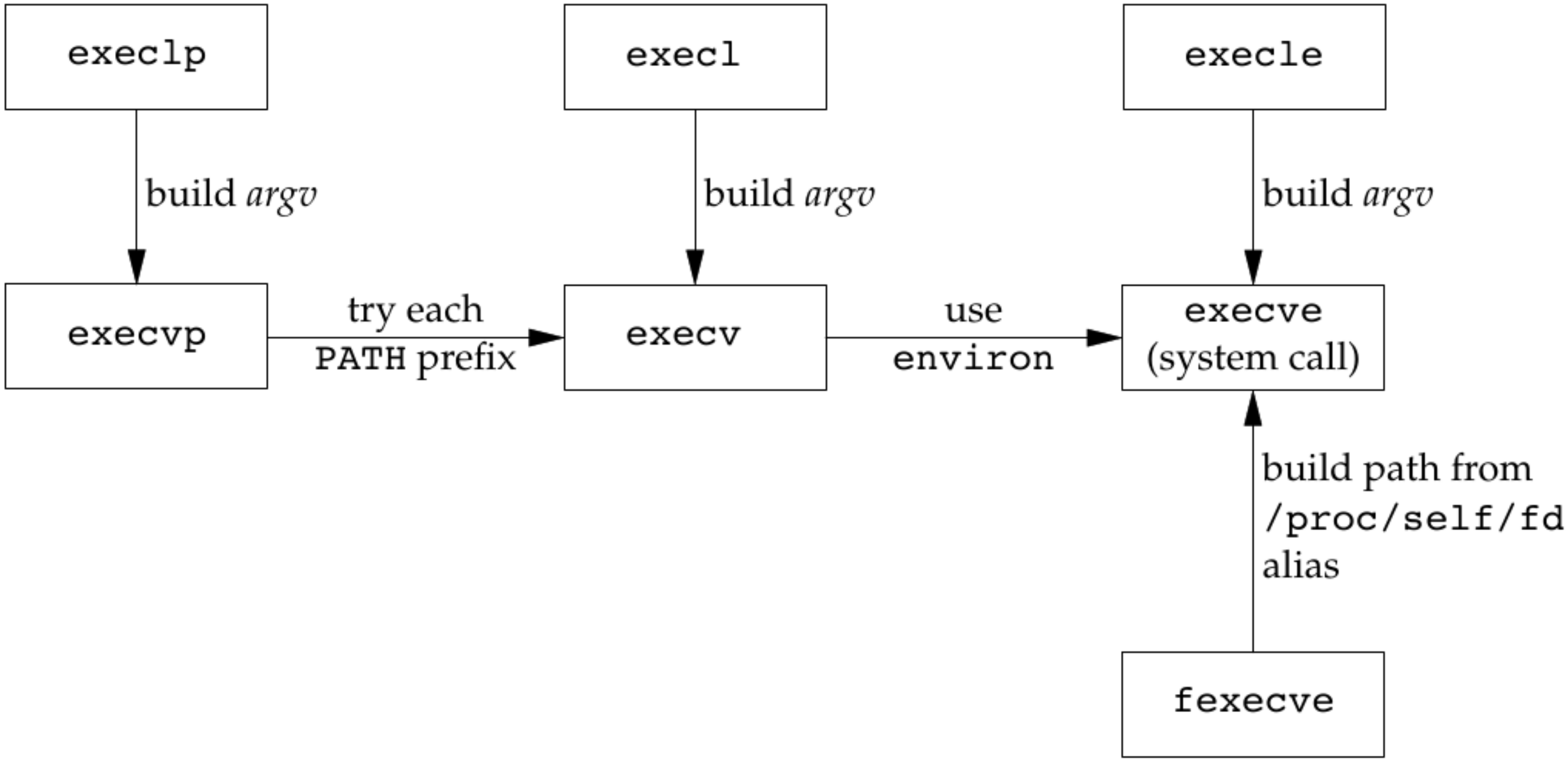
```
#include <unistd.h>
int execl(const char *pathname,
        const char *arg0, … /* (char *)0 */);
int execv(const char *pathname,
        char *const argv[]);
int execle(const char *pathname, const char *arg0, … /*
   (char *)0, char *const envp[] */);
int execve(const char *pathname,
        char *const argv[], char *const envp[]);
int execlp(const char *filename,
        const char *arg0, … /* (char *)0 */);
int execvp(const char *filename,
        char *const argv[]);
```

Return -1 on error, no return on success.

Differences among exec functions

- filename (execlp, execvp: uses PATH) v/s pathname (others: does not use PATH)
- list (l) v/s vector (v)
  - list of arguments (execl, execle, execlp)
  - array of pointers to arguments (execv, execve, execvp)
- pointer to an array of pointers to environment strings (execle, execve) v/s environ (others)

Relationship of 6 exec functions



Differences among exec functions

Function	*pathname*	*filename*	*fd*	Arg list	*argv*`[ ]`	`environ`	*envp*`[ ]`
`execl`	•			•		•	
`execlp`		•		•		•	
`execle`	•			•			•
`execv`	•				•	•	
`execvp`		•			•	•	
`execve`	•				•		•
`fexecve`			•		•		•
(letter in name)		`p`	`f`	`l`	`v`		`e`

Program 8.16: exec functions

```
#include <sys/types.h>
#include <sys/wait.h>
#include "apue.h"

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

int main(void) {
   pid_t pid;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {   /* specify pathname, specify environment */
        if (execle("/home/stevens/bin/echoall",
                           "echoall", "myarg1", "MY ARG2", (char *) 0,
                           env_init) < 0)
                err_sys("execle error");
   }
   if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {   /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
                err_sys("execlp error");  }
   exit(0); }
```

Inheritance by child from parent after exec

- PID, Parent PID
- Real UID, Real GID
- Supplementary GIDs
- Process GID
- Session ID
- Controlling Terminal
- Time Left Until Alarm Clock
- Current Working Directory
- Root Directory
- File Mode Creation Mask
- File Locks
- Process Signal Mask
- Pending Signals
- Resource Limits
- Nice value
- tms_utime, tms_stime, tms_cutime, tms_ustime values

11. Changing UIDs and GIDs

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Return: 0 if OK, -1 on error

- 1) If Superuser

  real, effective, saved set-UID := uid

- 2) If real Or saved set-UID = = uid

  effective := uid

  else

  errno := EPERM; return error;

seteuid(), setegid()

```
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

- Return: 0 if OK, -1 on error
- Only effective UID or GID is changed

“saved set-UID” Example: at

- Example: at, executes commands at a specified time.

```
➜  ~ ps -ef | grep atd
daemon     695     1  0 14:29 ?        00:00:00 /usr/sbin/atd -f
```

```
➜  ~ ls -l /usr/bin/at /usr/sbin/atd
-rwsr-sr-x 1 daemon daemon 51464 1月  15  2016 /usr/bin/at
-rwxr-xr-x 1 root   root   26632 1月  15  2016 /usr/sbin/atd
```

```
→ ~ at 5:00
warning: commands will be executed using /bin/sh
at> date >tmp.txt
at> <EOT>
job 22 at Tue Oct 10 05:00:00 2017
```

```
→ ~ sudo ls -l /var/spool/cron
total 12
drwxrwx--T 2 daemon daemon  4096 10月  9 16:40 atjobs
```

```
→ ~ sudo ls -l /var/spool/cron/atjobs
total 16
-rwx------ 1 zhu daemon 5979 10月  9 16:34 a00015017f65d8
-rwx------ 1 zhu daemon 5978 10月  9 16:41 a00016017f65ec
```

```
→ ~ sudo cat /var/spool/cron/atjobs/a00016017f65ec
#!/bin/sh
# atrun uid=1000 gid=1000
# mail zhu 0
umask 2
XDG_SEAT=seat0; export XDG_SEAT
XDG_SESSION_ID=c3; export XDG_SESSION_ID
ROS_ROOT=/opt/ros/kinetic/share/ros; export ROS_ROOT
ROS_MASTER_URI=http://localhost:11311; export ROS_MA
ROSLISP_PACKAGE_DIRECTORIES=; export ROSLISP_PACKAGE
cd /home/zhu || {
         echo 'Execution directory inaccessible' >&2
         exit 1
}
date >tmp.txt
```

"saved set-UID" Example: At

- Example: at, executes commands at a specified time.

1. assuming owned by daemon, set-UID bit is SET, after exec:
   - Real UID = our own UID
   - Effective UID = **daemon**
   - Saved set-UID = daemon

```
→ ~ ls -l /usr/bin/at
-rwsr-sr-x 1 daemon daemon 51464 1月  15  2016 /usr/bin/at
```

```
→ ~ at 5:00
warning: commands will be executed using /bin/sh
at> date >tmp.txt
at> <EOT>
job 22 at Tue Oct 10 05:00:00 2017
```

In general, we try to use the *least-privilege* model when we design our applications.

2. reduce privileges. it calls seteuid(), only e-UID is changed:

- Real UID = our own UID (unchanged!)
- Effective UID = our own UID
- Saved set-UID = **daemon** (unchanged!)

Runs with our own UID as effective UID, can access only our own normally accessed files.

No additional permissions.

```
→ ~ sudo ls -l /var/spool/cron
total 12
drwxrwx--T 2 daemon daemon   4096 10月   9 16:40 atjobs
```

3. Increase privileges to to access the configuration files that control which commands are to be run and the time at which they need to run. These files are owned by the daemon that will run the. The at command calls seteuid to set the effective user ID to daemon. This call is allowed because the argument to seteuid equals the saved set-user-ID. (This is why we need the saved set-user-ID.) After this, we have:

- Real UID = our own UID (unchanged!)
- Effective UID = **daemon**
- Saved set-UID = daemon (unchanged!)

4. After the files are modified to record the commands to be run and the time at which they are to be run, the at command lowers its privileges by calling seteuid to set its effective user ID to our user ID. This prevents any accidental misuse of privilege. At this point, we have

- Real UID = our own UID (unchanged!)
- Effective UID = **our own UID**
- Saved set-UID = daemon (unchanged!)

```
➜  ~ ls -l /usr/sbin/atd
-rwxr-xr-x 1 root root 26632 1月  15  2016 /usr/sbin/atd
```

```
➜  ~ ps -U root | grep atd
  695 ?        00:00:00 atd
➜  ~ ps -u daemon | grep atd
  695 ?        00:00:00 atd
```

Real uid = root

Effective uid = daemon

5. The daemon atd starts out running with root privileges. To run commands on our behalf, the daemon calls fork and the child calls setuid to change its user ID to our user ID. Because the child is running with root privileges, this changes all of the IDs. We have

- Real UID = our own UID
- Effective UID = **our own UID**
- Saved set-UID = our own UID

Now the daemon can safely execute commands.

Changing 3 UIDs

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

```
→ ~ ls -l /usr/bin/at
-rwsr-sr-x 1 daemon daemon 51464 1月  15  2016 /usr/bin/at
```

setreuid(), setregid()

```c
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

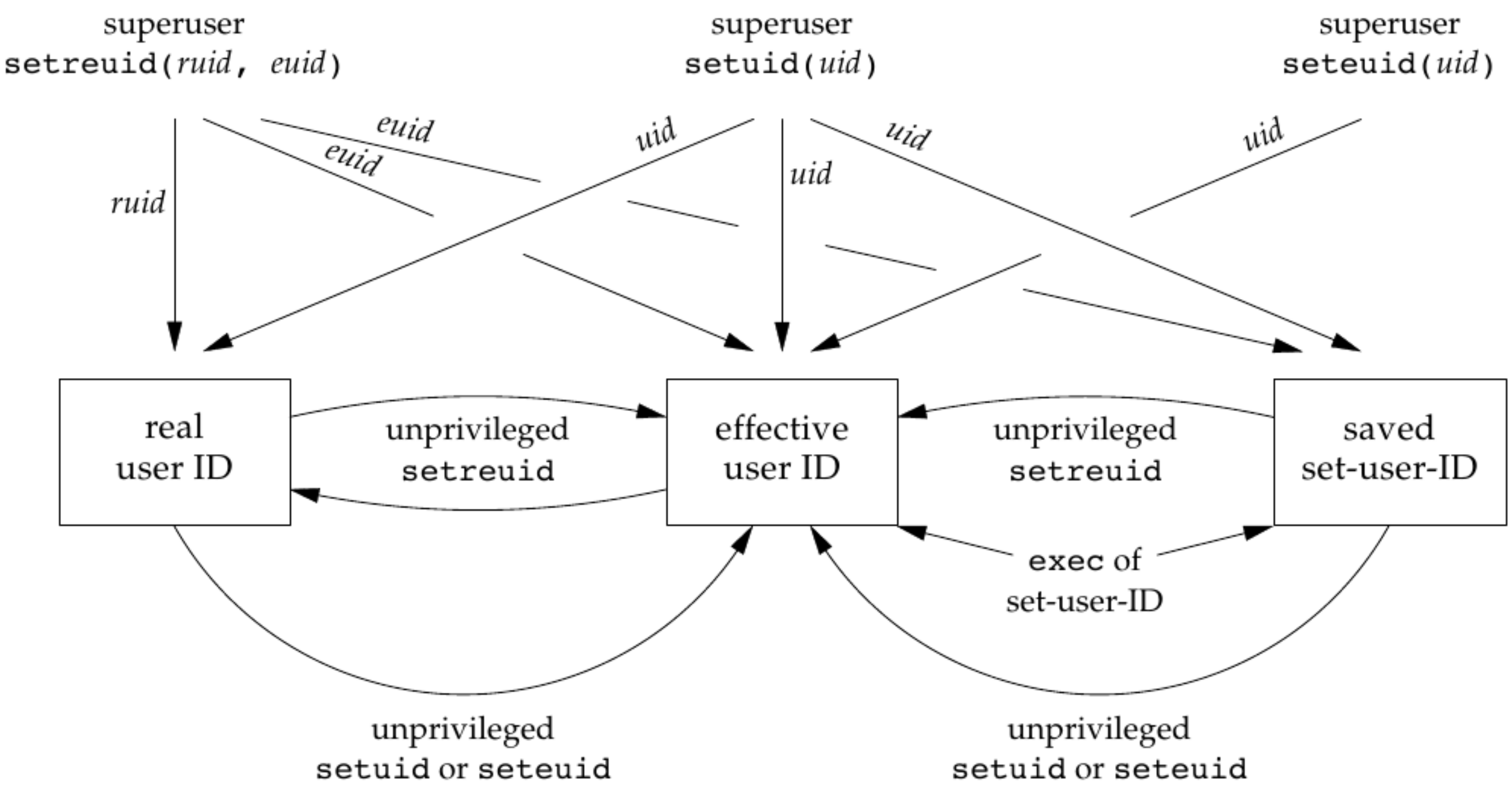
- Return: 0 if OK, -1 on error
- Sets the real user ID of the process to ruid and the effective user ID to euid.
- If argument is -1, leave that ID unchanged.

setreuid

- **int setreuid(uid_t *ruid*, uid_t *euid*);**

- The **setreuid**() function has been used to swap the real and effective user IDs in set-user-ID programs to temporarily relinquish the set-user-ID value.
- This purpose is now better served by the use of the **seteuid**() function.

Summary of set ID functions



13. system Function

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

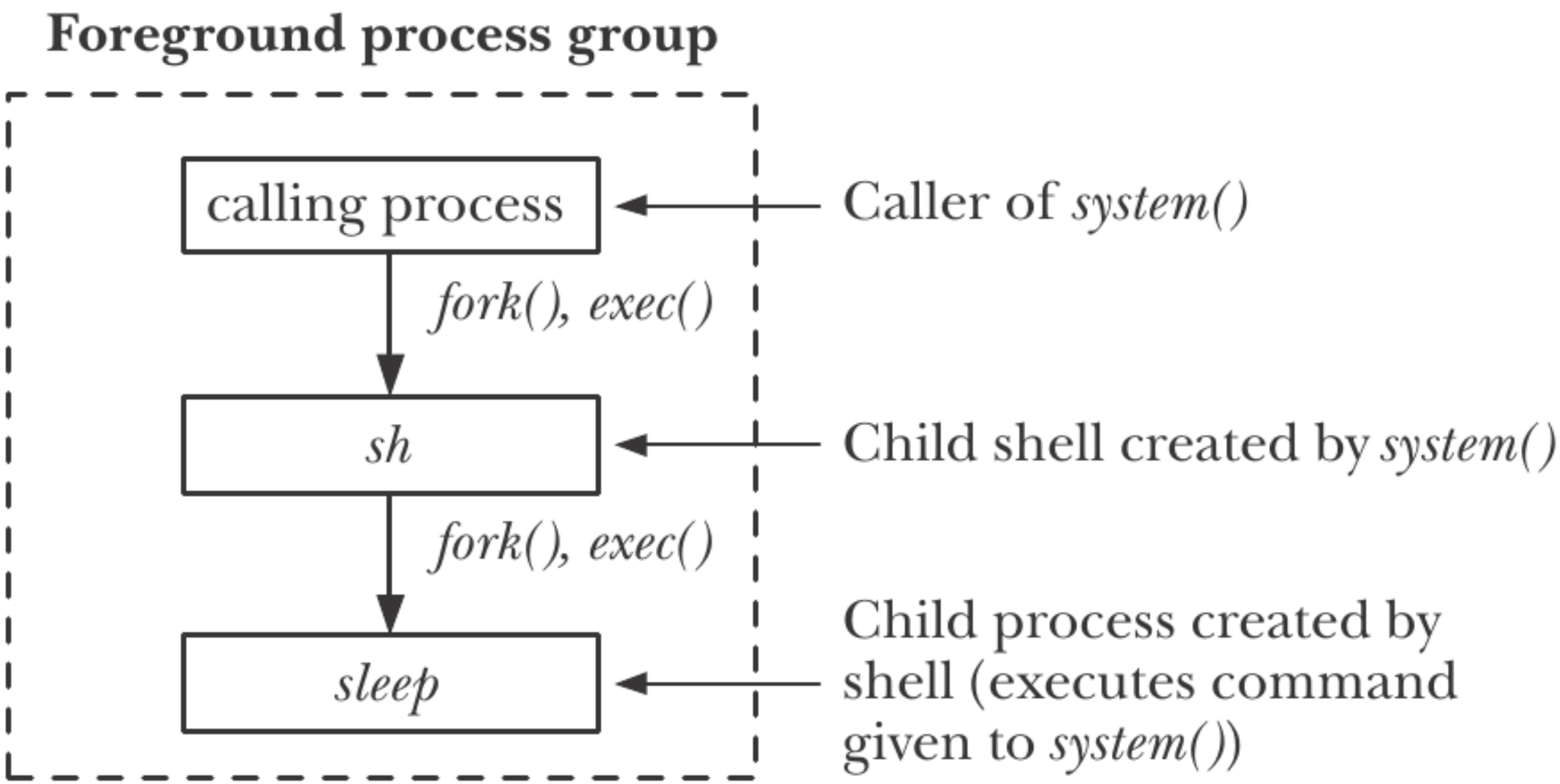
uses fork to create a child process that executes the shell command using execl :

```
execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

system() returns after the command has been completed.

System example

Arrangement of processes during execution of system("sleep 20")



Program 8.22: system implement

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        <errno.h>
#include        <unistd.h>

int system(const char *cmdstring)
                /* version without signal handling */
{
   pid_t        pid;
   int          status;

   if (cmdstring == NULL)
        return(1);  /* always a command processor with Unix */

   if ( (pid = fork()) < 0) {
        status = -1;   /* probably out of processes */
   }
```

Program 8.22

```
else if (pid == 0) {              /* child */
    execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
    _exit(127);               /* execl error */

} else {                          /* parent */
    while (waitpid(pid, &status, 0) < 0)
        if (errno != EINTR) {
            status = -1; /* error other than EINTR */
            break;
        }
}
return(status);
}
```

to prevent child flushing buffer

Program 8.23: calling system

```
#include      <sys/types.h>
#include      <sys/wait.h>
#include      "apue.h"
int main(void) {
   int         status;
   if ( (status = system("date")) < 0)
        err_sys("system() error");
   pr_exit(status);
   if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
   pr_exit(status);
   if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
   pr_exit(status);
   exit(0);
}
```

Program 8.23: results

$ **a.out**

Thu Aug 29 14:24:19 MST 1991
normal termination, exit status = 0 for date

sh: nosuchcommand: not found
normal termination, exit status = 1 for nosuchcommand

stevens console	Aug 25 11:49
stevens ttyp0	Aug 29 05:56
stevens ttyp1	Aug 29 05:56
stevens ttyp2	Aug 29 05:56

normal termination, exit status = 44 for exit

Problem

- What happens if we call system from a set-user-ID program?
- A security hole!
- Should never be done!

Compile into program tsys

Program 8.24: system from cmd

```
#include    "apue.h"
int main(int argc, char *argv[]) {
  int       status;
  if (argc < 2)
      err_quit("command-line argument required");
  if ( (status = system(argv[1])) < 0)
      err_sys("system() error");
  pr_exit(status);
  exit(0);
}
```

Compile into program **printuids**

Program 8.25: print UIDs

```
#include "apue.h"

int
main(void)
{
  printf("real uid = %d, effective
  uid = %d\n", getuid(), geteuid());
  exit(0);
}
```

Program 8.24, 8.25: results

$ **tsys printuids**
real uid = 224, effective uid = 224
normal termination, exit status = 0

make tsys set-user-ID

$ **su**
Password:
chown root tsys
chmod u+s tsys
ls –l tsys
-rwsrwxr-x 1 root 105737 Aug 18 11:21 tsys
exit

$ **tsys printuids**
real uid = 224, effective uid = 0 this is a security hole
normal termination, exit status = 0

17. Process Times

#include <sys/times.h>

clock_t times(struct tms *buf);

- Returns: elapsed wall clock time in clock ticks if OK, -1 on error

```
struct tms {
 clock_t tms_utime;  /* user CPU time */
 clock_t tms_stime;  /* system CPU time */
 clock_t tms_cutime; /* sum of user time for terminated children */
 clock_t tms_cstime; /* sum of system time for terminated children*/
};
```

Program 8.30 (main())

```
#include        <sys/times.h>
#include        "apue.h"

static void     pr_times(clock_t, struct tms *, struct tms *);
static void     do_cmd(char *);

int main(int argc, char *argv[]) {
    int             i;

    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);        /*once each command-line arg */
    exit(0);
}
```

Program 8.30 do_cmd()

```
static void do_cmd(char *cmd)     /*execute and time the "cmd" */ {
     struct tms      tmsstart, tmsend;
     clock_t         start, end;
     int             status;
     fprintf(stderr, “\ncommand: %s\n” , cmd);
     if ( (start = times(&tmsstart)) == -1)     /* starting values */
          err_sys("times error");
     if ( (status = system(cmd)) < 0) /* execute command */
          err_sys( “system() error” );
     if ( (end = times(&tmsend)) == -1)         /* ending values */
          err_sys( “times error” );
     pr_times(end-start, &tmsstart, &tmsend);
     pr_exit(status);
}
```

Program 8.30 pr_times()

```
static void pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
  static long                 clktck = 0;
  if (clktck == 0)  /* fetch clock ticks per second first time */
          if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
                  err_sys("sysconf error");
  fprintf(stderr, "  real:  %7.2f\n", real / (double) clktck);
  fprintf(stderr, "  user:  %7.2f\n",
          (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
  fprintf(stderr, "  sys:   %7.2f\n",
          (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
  fprintf(stderr, "  child user:  %7.2f\n",
          (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
  fprintf(stderr, "  child sys:   %7.2f\n",
          (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Program 8.30: results

```
→  proc git:(master) ✗ ./times1 "sleep 3"

command: sleep 3
  real:      3.00
  user:      0.00
  sys:       0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0
```

```
→  proc git:(master) ✗ ./times1 "find /usr/include -type f -exec wc {} \; >/dev/null"

command: find /usr/include -type f -exec wc {} \; >/dev/null
  real:     36.68
  user:      0.00
  sys:       0.00
  child user:     0.74
  child sys:      2.69
normal termination, exit status = 0
```