CS214 Recitation Sec.7

Oct. 31, 2017

Topics

1. HW5: Implementing “ps” command in C
2. Differences between threads & processes
3. Thread creation & joining

HW5 - Implementing "ps"

0. Modify the *ls* we wrote this week except output */proc*
Then open the file '*status*', look for the uid section and extract the owner's uid.
Using *pwd.h*, determine the name of the user who owns the process and print it out as well.

1. Open *status* alongside/after cmdline, read through and parse the status file looking for 'uid'

HW5 - Implementing "ps" (Cont.)

2. Then open the file *schedstat* and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a run queue, # of times context switched

Like default “ps” only prints out info for YOUR procs by default - so can get current uid within your code using getuid and comb through /proc and only print out status and schedstat information for staff whose uid matches

PS - where is process information

/proc - “proc filesystem” is a pseudo-filesystem which provides *an interface to kernel data structures*. It is commonly mounted at /proc. It includes all the information about processes. The information for each process is stored in a folder name by its PID:

```
-sh-4.2$ ls /proc
1      1296   16406  21232  2473   27847  4263  6318  779   9449
10     12972  16442  21247  2475   27859  43    6320  7830  9452
1005   13     16446  21265  24870  27861  4329  6330  784   9453
1011   1307   16453  21267  24990  27883  4331  6331  7841  946
10251  1312   16456  21283  25040  27903  449   6335  785   9462
10353  13327  16465  21295  25088  2793   4551  6350  786   9471
1073   1334   16518  21315  2556   2794   4560  6358  7868  9476
```

PS - where is process information

<pwd.h> - provides a definition for **struct passwd**, which saves the information such user's login name and program to be used

NAME

pwd.h - password structure

SYNOPSIS

```
#include <pwd.h>
```

DESCRIPTION

The *<pwd.h>* header provides a definition for **struct passwd**, which includes at least the following members:

```
char    *pw_name   user's login name
uid_t    pw_uid    numerical user ID
gid_t    pw_gid    numerical group ID
char    *pw_dir    initial working directory
char    *pw_shell  program to use as shell
```

The **gid_t** and **uid_t** types are defined as described in *<sys/types.h>*.

The following are declared as functions and may also be defined as macros. Function prototypes must be provided for use with an ISO C compiler.

```
struct passwd *getpwnam(const char *);
struct passwd *getpwuid(uid_t);
int            getpwnam_r(const char *, struct passwd *, char *,
                   size_t, struct passwd **);
int            getpwuid_r(uid_t, struct passwd *, char *,
                   size_t, struct passwd **);
void           endpwent(void);
struct passwd *getpwent(void);
void           setpwent(void);
```

PS - where is process information (Cont.)

file '*stat*'

/proc/[pid]/stat

Status information about the process. This is used by ps(1). It is defined in the kernel source file *fs/proc/array.c*.

The fields, in order, with their proper scanf(3) format specifiers, are listed below. Whether or not certain of these fields display valid information is governed by a ptrace access mode **PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT** check (refer to ptrace(2)). If the check denies access, then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

(1) *pid* %d
The process ID.

(2) *comm* %s
The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

(3) *state* %c
One of the following characters, indicating process state:

R Running

S Sleeping in an interruptible wait

PS - where is process information (Cont.)

file '*status*'

/proc/[pid]/status

Provides much of the information in */proc/[pid]/stat* and */proc/[pid]/statm* in a format that's easier for humans to parse. Here's an example:

```
$ cat /proc/$$/status
Name:   bash
Umask:  0022
State:  S (sleeping)
Tgid:   17248
Ngid:   0
Pid:    17248
PPid:   17200
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    100     100     100     100
FDSize: 256
Groups: 16 33 100
NStgid: 17248
NSpid:  17248
NSpgid: 17248
NSsid:  17200
VmPeak:     131168 kB
VmSize:     131168 kB
VmLck:           0 kB
VmPin:           0 kB
VmHWM:       13484 kB
VmRSS:       13484 kB
```

PS - where is process information (Cont.)

file '*comm*'

/proc/[pid]/comm (since Linux 2.6.33)
This file exposes the process's *comm* value—that is, the command name associated with the process. Different threads in the same process may have different *comm* values, accessible via */proc/[pid]/task/[tid]/comm*. A thread may modify its *comm* value, or that of any of other thread in the same thread group (see the discussion of **CLONE_THREAD** in clone(2)), by writing to the file */proc/self/task/[tid]/comm*. Strings longer than **TASK_COMM_LEN** (16) characters are silently truncated.

This file provides a superset of the prctl(2) **PR_SET_NAME** and **PR_GET_NAME** operations, and is employed by pthread_setname_np(3) when used to rename threads other than the caller.

PS - where is process information (Cont.)

file '*schedstat*'

includes time spent running on CPU (in nanoseconds), time spent waiting on a run queue, # of times context switched.

Use the comand */proc/<pid>/schedstat* and get the following output:

```
-sh-4.2$ cat /proc/76/schedstat
17882 298 2
```

PS - when time is expressed by jeffies

The software clock, HZ, and jiffies

The accuracy of various system calls that set timeouts, (e.g., select(2), sigtimedwait(2)) and measure CPU time (e.g., getrusage(2)) is limited by the resolution of the *software clock*, a clock maintained by the kernel which measures time in *jiffies*. The size of a jiffy is determined by the value of the kernel constant *HZ*.

The value of *HZ* varies across kernel versions and hardware platforms. On i386 the situation is as follows: on kernels up to and including 2.4.x, HZ was 100, giving a jiffy value of 0.01 seconds; starting with 2.6.0, HZ was raised to 1000, giving a jiffy of 0.001 seconds. Since kernel 2.6.13, the HZ value is a kernel configuration parameter and can be 100, 250 (the default) or 1000, yielding a jiffies value of, respectively, 0.01, 0.004, or 0.001 seconds. Since kernel 2.6.20, a further frequency is available: 300, a number that divides evenly for the common video frame rates (PAL, 25 HZ; NTSC, 30 HZ).

The times(2) system call is a special case. It reports times with a granularity defined by the kernel constant *USER_HZ*. User-space applications can determine the value of this constant using *sysconf(_SC_CLK_TCK)*.

Isdigit function in <ctype.h>

```c
#include <ctype.h>
main(){
    char str[] = "123@#FDsP[e?";
    int i;
    for(i = 0; str[i] != 0; i++)
        if(isdigit(str[i]))
            printf("%c is an digit character\n", str[i]);
}
```

Output:

```
1 is an digit character
2 is an digit character
3 is an digit character
```

Getuid function in <unistd.h>

NAME top

getuid, geteuid - get user identity

SYNOPSIS top

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

uid_t getuid(void);
uid_t geteuid(void);
```

DESCRIPTION top

getuid() returns the real user ID of the calling process.

geteuid() returns the effective user ID of the calling process.

HW5 - A Sample Code

#include <ctype.h>

```
int check_if_number (char *str)
{
  int i;
  for (i=0; str[i] != '\0'; i++)
  {
    if (!isdigit (str[i]))
    {
      return 0;
    }
  }
  return 1;
}
```

#include <pwd.h>

```
const char *getUserName(int uid)
{
  struct passwd *pw = getpwuid(uid);
  if (pw)
  {
    return pw->pw_name;
  }

  return "";
}
```

HW5 - A Sample Code (Cont.)

```
#define MAX_BUF 2048
#define INT_SIZE_BUF 8
#define COMM_SIZE_BUF 256


void pidaux (char *cuser)

int main (int argc, char *argv[])
{
  char current_user[256];
  strcpy (current_user, getUserName(getuid()));
  printf("UID\t STATE\t PID\t PPID\t NUM_TR\t COMM\t\t WAIT_T\t\t
      T_SWCH\t\t EXEC_T\n");
  pidaux(current_user);
  long Hertz=sysconf(_SC_CLK_TCK);
  printf("Hertz: %ld\n", Hertz);
  return 0;
}
```

HW5 - A Sample Code (Cont.)

```c
void pidaux (char *cuser)
{
  DIR *thingy;
  FILE *fp;
  struct dirent *entry;

  char path[MAX_BUF], comm_buf[COMM_SIZE_BUF];
  char uid_int_str[INT_SIZE_BUF]={0};
  char *line;
  char *user, *command;
  size_t len=0; // auto allocation
  thingy = opendir ("/proc/");


  if (thingy == NULL)
  {
        perror ("Fail");
        exit(0);
  }
```

```c
while ((entry = readdir (thingy)) != NULL)
{
  if (check_if_number (entry->d_name))
  {
    strcpy(path, "/proc/");
    strcat(path, entry->d_name);
    strcat(path, "/status");
    fp = fopen(path, "r");

    if(fp!=NULL)
    {
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      getline(&line, &len, fp);
      sscanf(line, "Uid:    %s ", uid_int_str);
    }
    else
    {
      fprintf(stdout, "FP is NULL\n");
    }
```

HW5 - A Sample Code (Cont.)

```
strcpy(path, "/proc/");
strcat(path, entry->d_name);
strcat(path, "/stat");

fp = fopen(path, "r");
getline(&line, &len, fp);

char comm[COMM_SIZE_BUF],state;
unsigned int flags;
int pid,ppid,pgrp,session,tty_nr,tpgid;
unsigned long minflt,cminflt,majflt,cmajflt,utime,stime;
unsigned long long starttime;
long cutime,cstime,priority,nice,num_threads,itreavalue;

sscanf(line,"%d %s %c %d %d %d %d %d %u %lu %lu %lu %lu %lu %lu %ld %ld
  %ld %ld %ld %ld %llu",&pid,comm,&state,&ppid,&pgrp,&session,&tty_nr,
  &tpgid,&flags,&minflt,&cminflt,&majflt,&cmajflt,&utime,&stime,&cutime,
  &cstime,&priority,&nice,&num_threads,&itreavalue,&starttime);
```

HW5 - A Sample Code (Cont.)

```
strcpy (path, "/proc/");
strcat (path, entry->d_name);
strcat (path, "/comm");

fp = fopen (path, "r");
if (fp != NULL)
{
  fscanf(fp, "%s", comm_buf);
  fclose(fp);
}
```

```
strcpy(path,"/proc/");
strcat (path, entry->d_name);
strcat(path,"/schedstat");

fp=fopen(path,"r");
char exec_runtime_str[256];
char waiting_on_queue_str[256];
char times_switched_str[256];
if(fp!=NULL)
{
  getline(&line,&len,fp);
  sscanf(line,"%s %s %s", exec_runtime_str, waiting_on_queue_str,
      times_switched_str);
}
unsigned long long exec_runtime = atol(exec_runtime_str);
unsigned long waiting_on_queue = atol(waiting_on_queue_str);
unsigned long times_switched = atol(times_switched_str);
```

HW5 - A Sample Code (Cont.)

```
        // only print for the current user
        if (strcmp(getUserName(atoi(uid_int_str)), cuser) == 0)
        {
          if (strlen(getUserName(atoi(uid_int_str))) < 9)
          {
              user = getUserName(atoi(uid_int_str));
          }
          else
          {
              user = uid_int_str;
          }
          printf("%s\t %c\t %d\t %d\t %ld\t %s\t\t %lu\t\t %lu\t\t %llu\n", user, state,
            pid, ppid, num_threads, comm_buf, waiting_on_queue, times_switched, exec_runtime);
        }
      }
    }
    closedir (thingy);
}
```

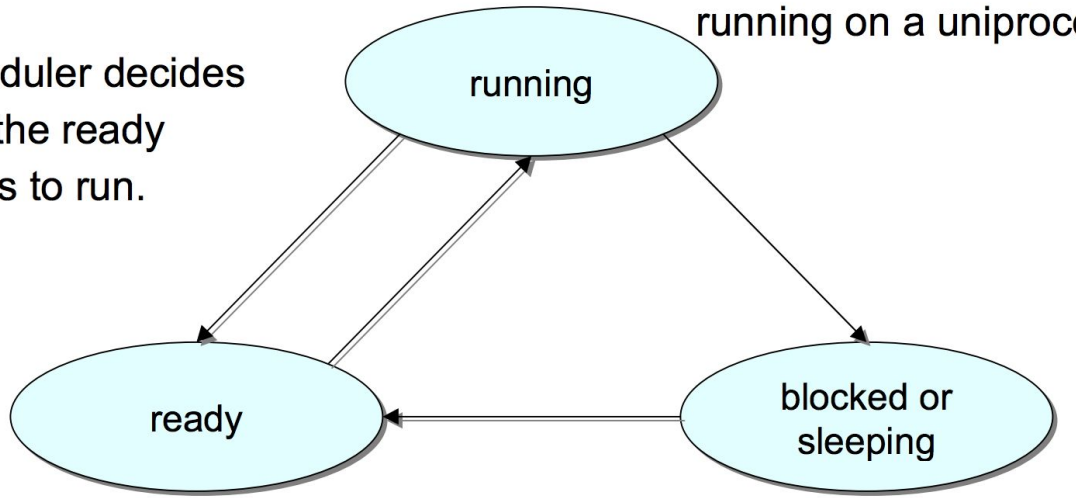
Recall: Process states

In a single CPU system what is the maximum number of processes that can be in the running state?

Recall: Process states (main memory)

Only one process can be running on a uniprocessor

The scheduler decides which of the ready processes to run.

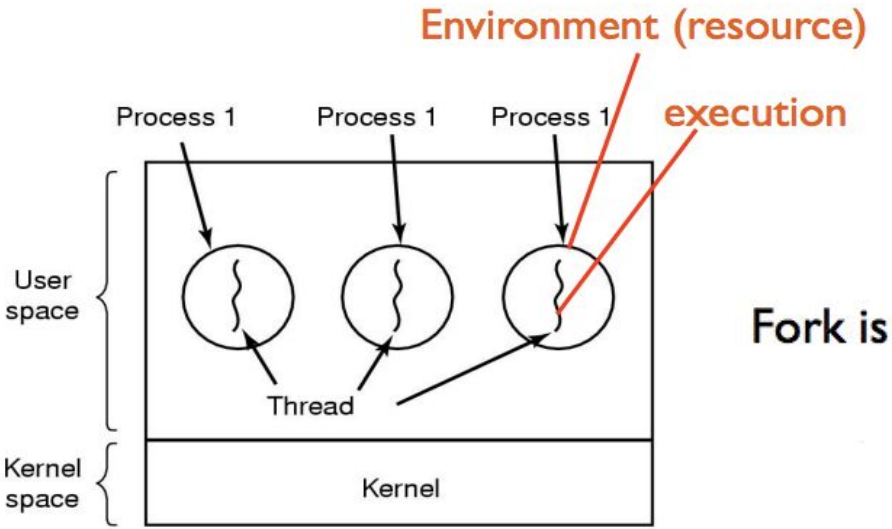


A process is ready if it could use the CPU immediately.

A process is blocked if it waiting for an event (I/O, signal)

Recall: Properties of processes

Processes do not share resources well



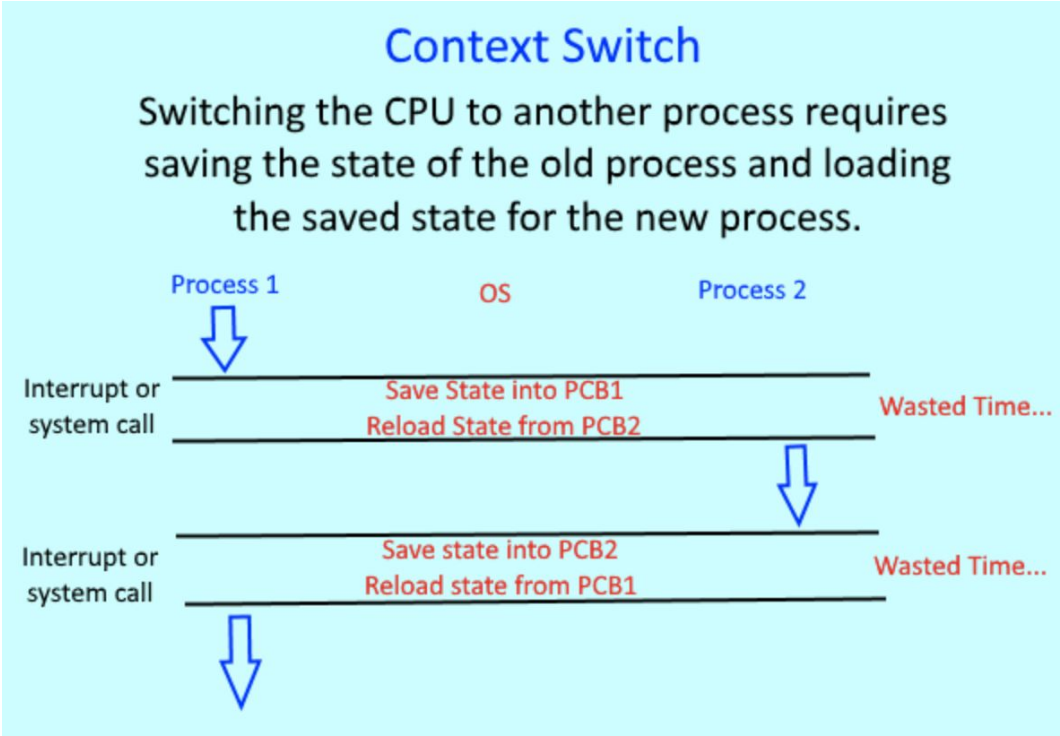
Fork is expensive (time & memory)

Three processes each with one thread

Recall: Properties of processes (Cont.)

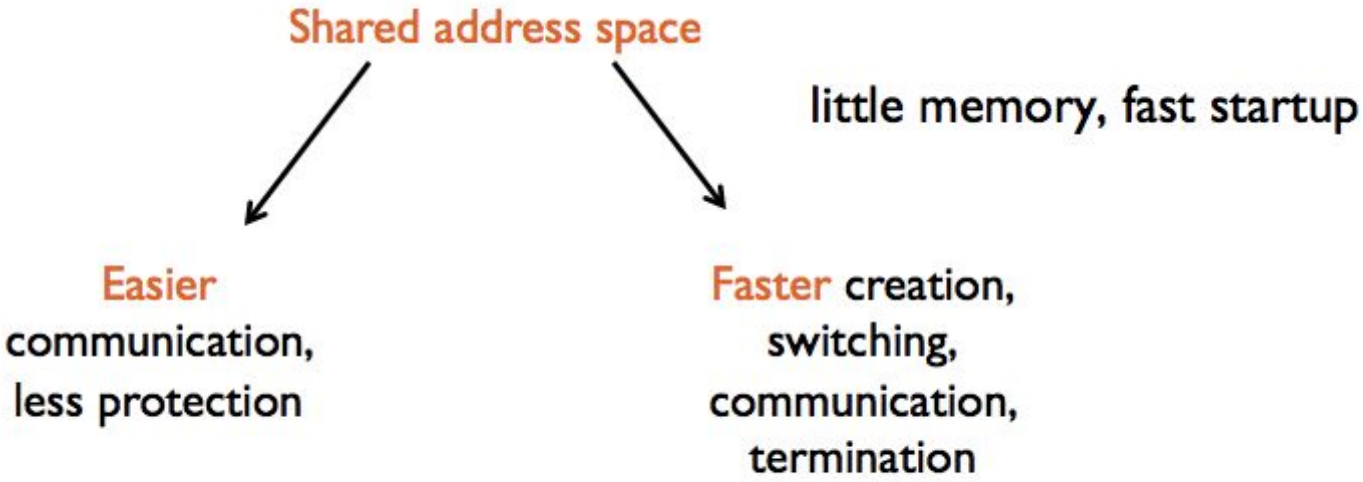
Process context switching cost is high

Context switch enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system



Recall: Properties of threads (Cont.)

Therefore ... Threads: light-weight processes



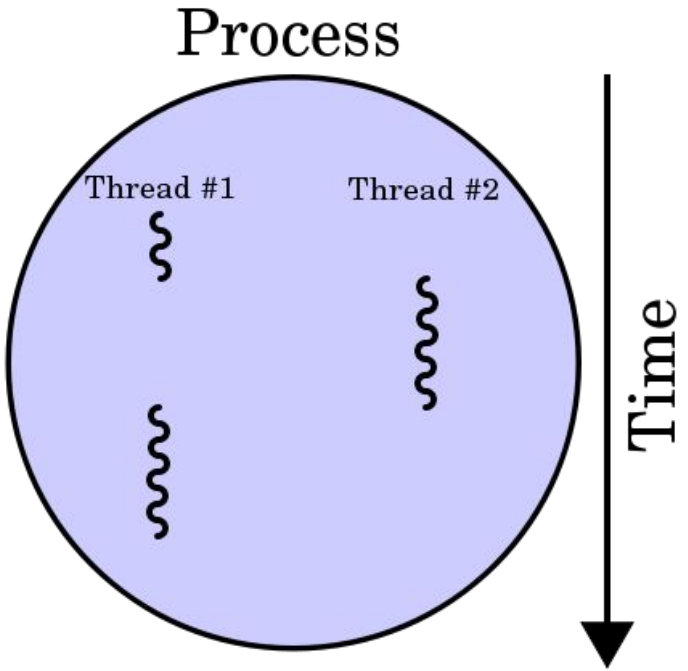
Threads

Each process can include many threads.

A ***thread*** is the smallest entity scheduled for execution on the CPU. *In general, a thread is a component of a process.*

The typical difference is that *threads (of the same process) run in a shared global memory space*, while processes run in separate memory spaces.

Suitable for dealing with *multiple parallel sub-tasks*



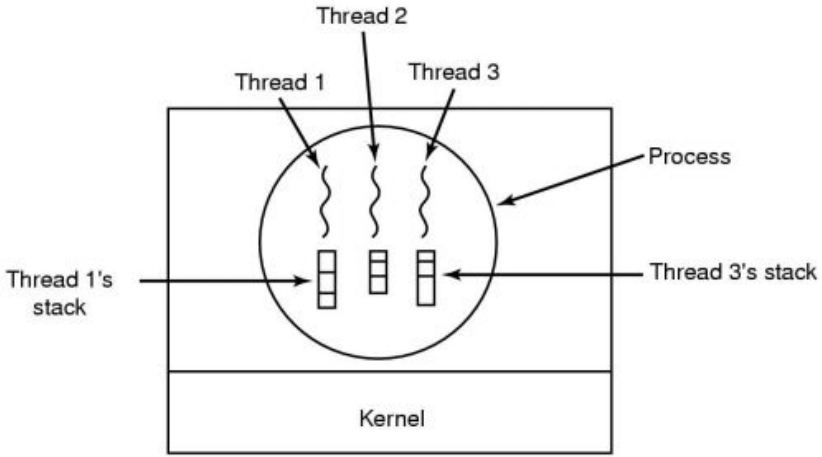
Threads (Cont.)

All threads of a process share:

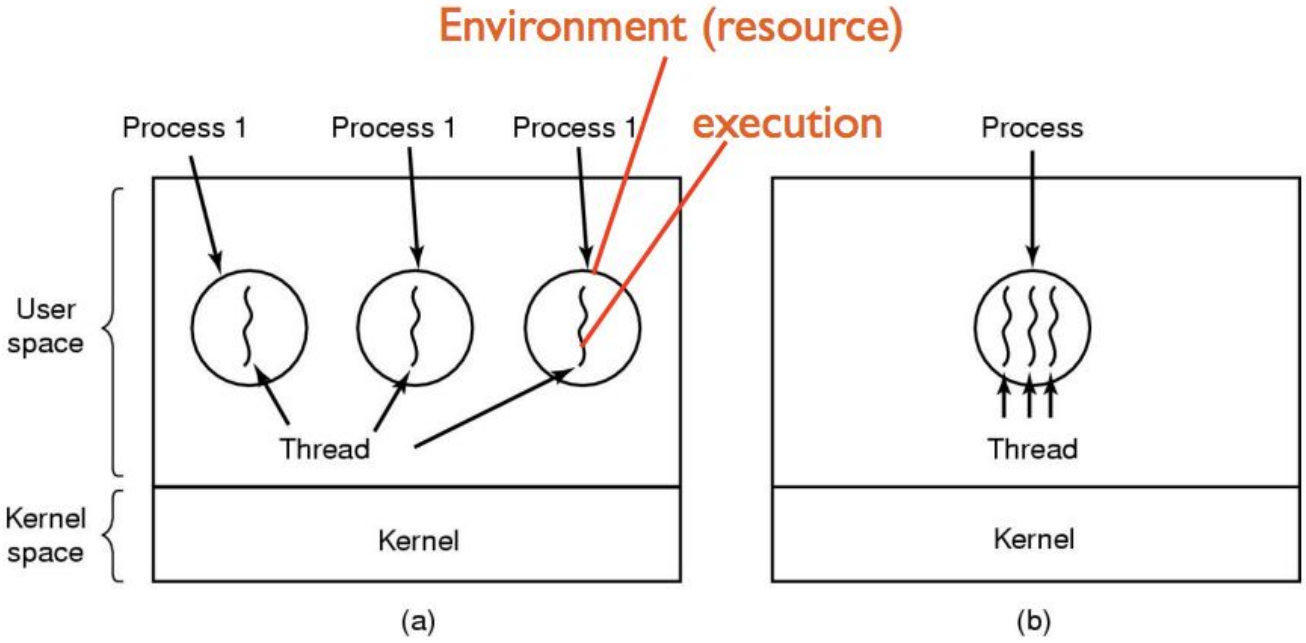
- Process ID
- Memory (program code and global data)
- Working environment (current directory, user ID, etc.)
- Signal handlers

Threads (Cont.)

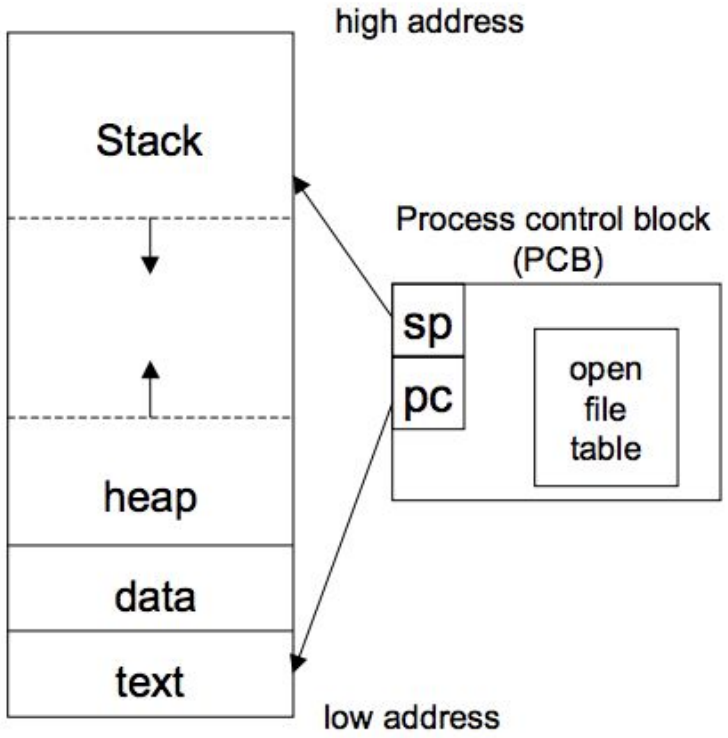
- Each has its own function calls & automatic (local) variables
- Need *program counter* and *stack* for each thread
- Each thread has its own *Thread ID* (integer)



Differences between threads & processes

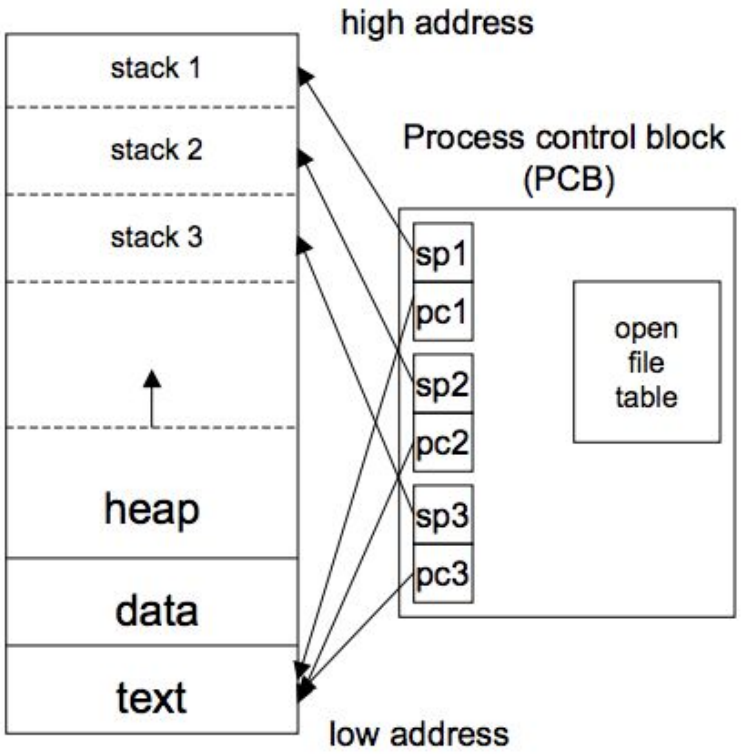


Memory for processes



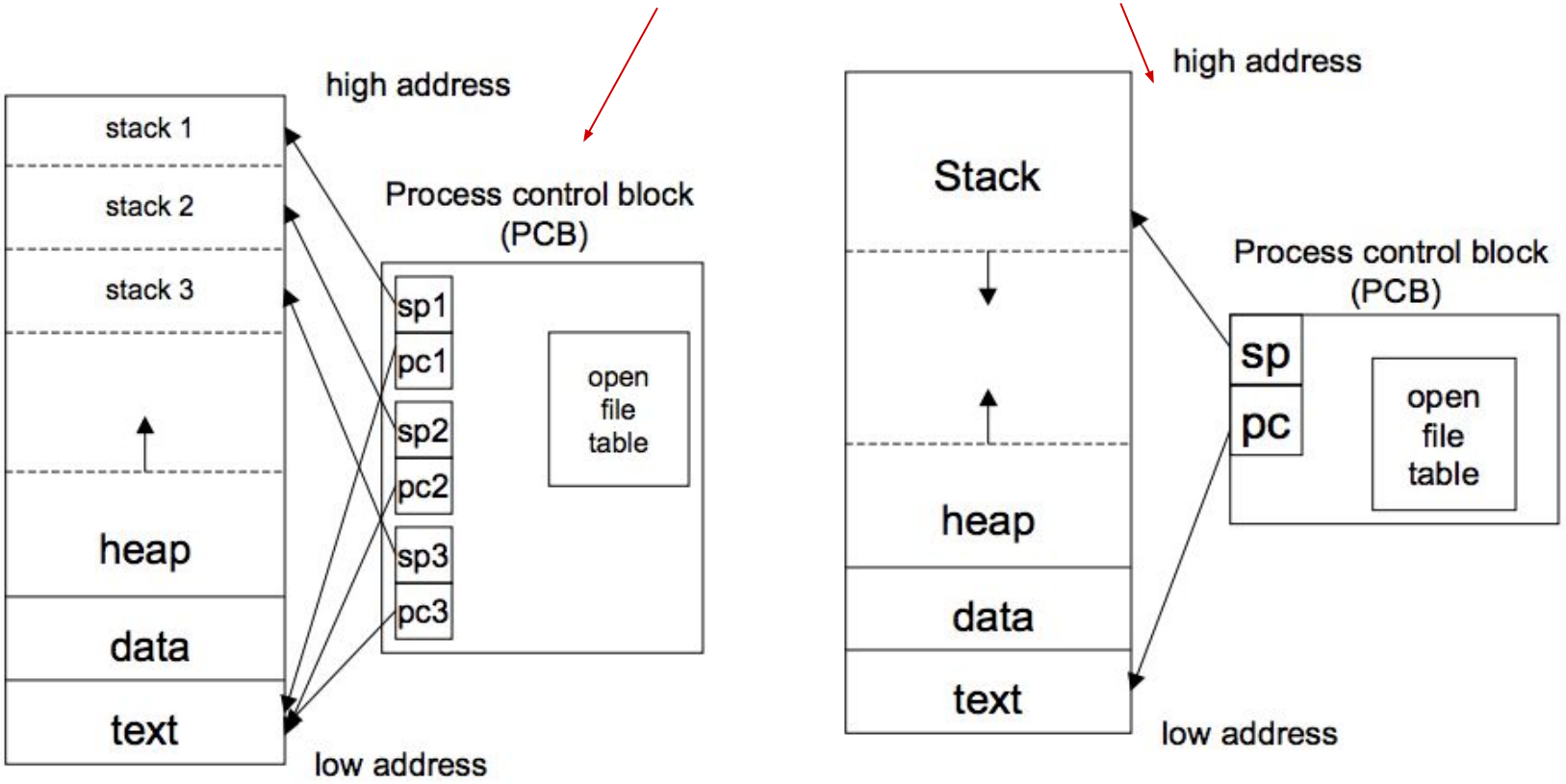
- Each process has its own
  - program counter
  - stack
  - stack pointer
  - address space
- Processes may share
  - open files
  - pipes

Memory for threads



- Each thread has its own
  - program counter
  - stack
  - stack pointer
- Threads share
  - address space
    - variables
    - code
  - open files

Comparison of threads & processes



Creation time for threads & processes

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.3 GHz Opteron (16 cpus)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cpus)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus)	54.5	1.1	22.2	2.0	1.2	0.6

http://www.llnl.gov/computing/tutorials/pthreads.

Timings reflect 50,000 process/thread

Creations were performed with the `time` utility, and units are in seconds, no optimization flags.

Decision: processes or threads?

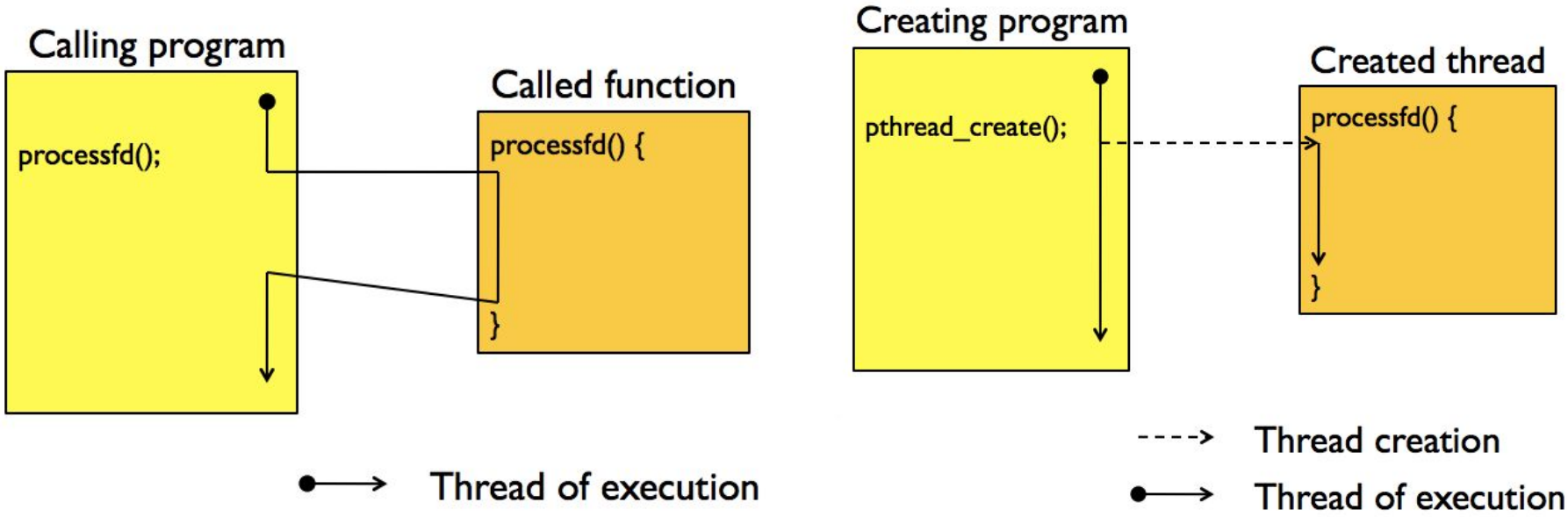
Processes

- Exploit parallelism successfully
- Separate memory space: good for protection

Threads

- Exploit parallelism successfully
- Shared memory space: good for working together

1 Thread vs Threaded function call



Pthread

POSIX threads (pthreads) is the most commonly used thread package on Unix/Linux

Frequently-used functions in <pthread.h>:

- *pthread_create* - create a new thread
- *pthread_join* - join with a terminated thread
- *pthread_exit* - terminate calling thread
- *pthread_cancel* - send a cancellation request to a thread

Thread Creation & Joining

pthread_create - create a new thread

thread identifies a thread within a process

attr sets attributes such as priority, *initial stack size* – can be specified as *NULL* to get defaults

start-routine - the function to call to start the thread

arg is the argument to **start-routine**

SYNOPSIS top

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

Compile and link with -pthread.
```

DESCRIPTION top

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

Thread Creation & Joining

If I call pthread_create twice, how many stacks does my process have?

Your process will contain three stacks - one for each thread. The first thread is created when the process starts, and you created two more.

Thread Creation & Joining

pthread_join - join with a terminated thread

thread identifies a thread within a process

retval returned value by the thread when it terminates

Purpose of pthread_join():

1. Wait for a thread to finish
2. Clean up thread resources
3. Grabs the return value of the thread

SYNOPSIS top

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

Compile and link with -pthread.
```

DESCRIPTION top

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

Thread Creation & Joining

A thread can be terminated by

- Returning from the thread function
- the *main()* function exiting or *exit()* called or sending a *SIGTERM* signal
- *pthread_exit* - join with a terminated thread
- *pthread_cancel* - send a cancellation request to a thread

What is the difference between *exit* and *pthread_exit*?

- exits(): exits the entire process and sets the processes exit value. All threads inside the process are stopped
- pthread_exit(void*): only stops the calling thread. The pthread library will automatically finish the process if there are no other threads running.

Passing Arguments to Threads

```
MyArg *p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd;  /* assumes fd is defined */
strncpy(p->name, "CSC209", 7);
result = pthread_create(&threadID, NULL,
                        myThreadFcn, (void *)p);
void *myThreadFcn(void *p) {
    MyArg *theArg = (MyArg *) p;
    write(theArg->fd, theArg->name, 7);
    close(theArg->fd);
    free(theArg);
    return NULL;
}
```

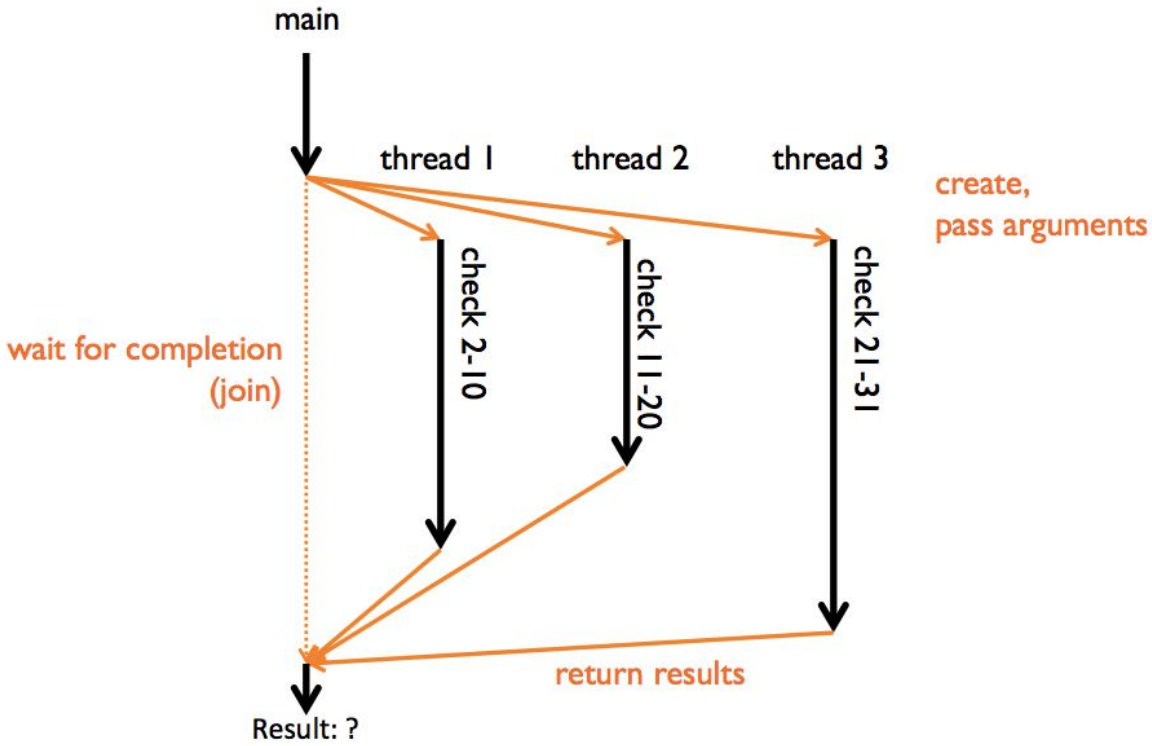
Thread example

Decide if an integer is prime

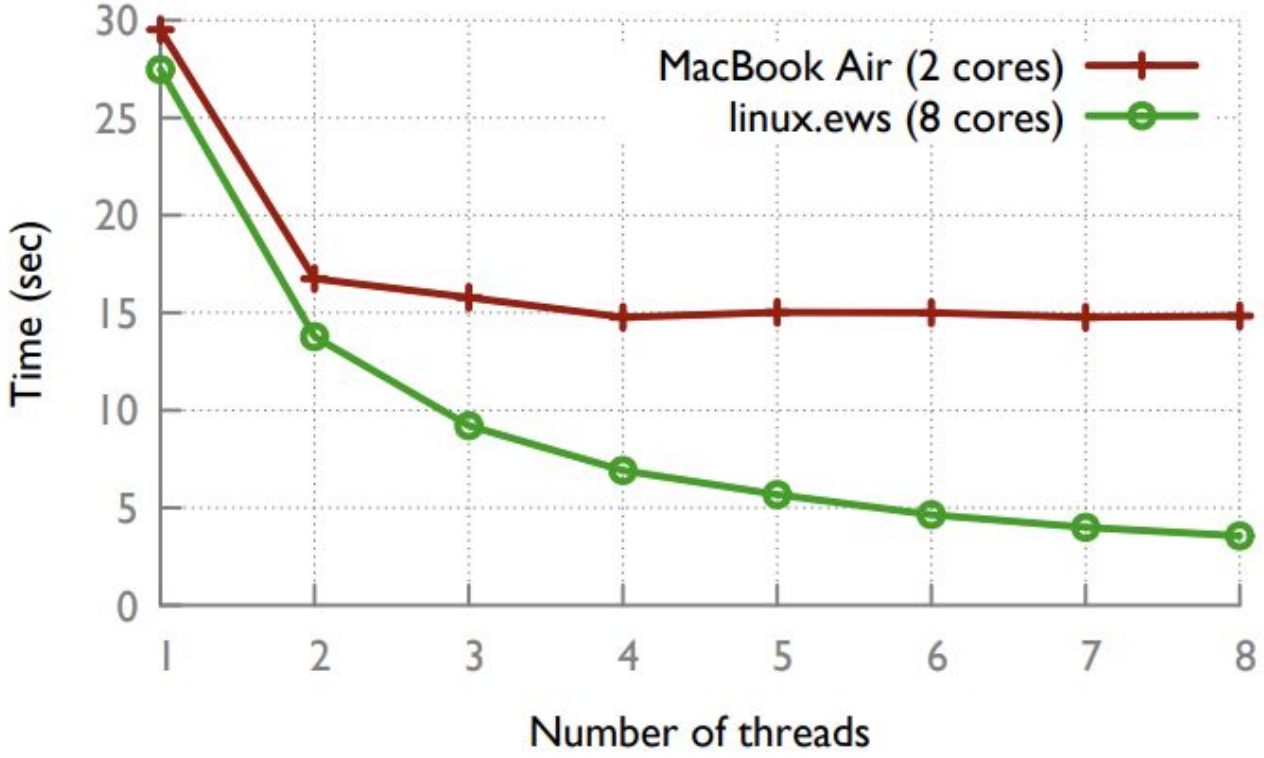
- Input: integer
- Output: prime, or composite with factor

Exploit parallelism

- Testing primality can be slow
- My laptop has multiple cores



Thread example




Happy Halloween