CS 214 Recitation(Sec. 6)

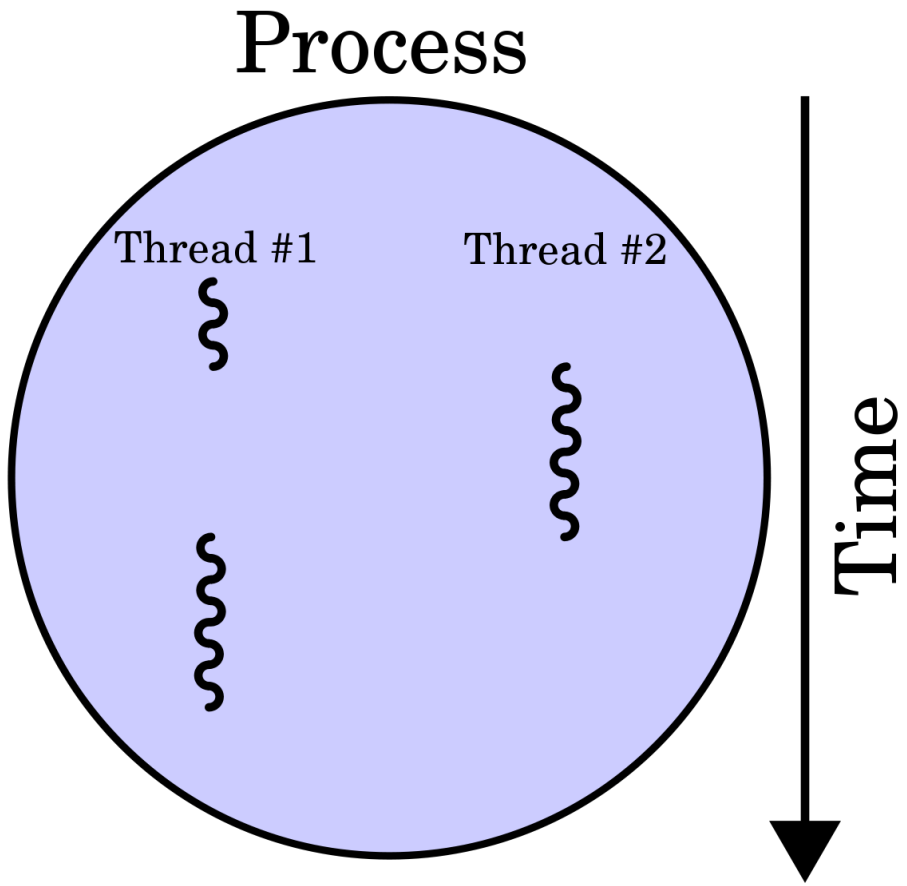
Zuohui Fu
Ph.D. Department of Computer Science
Office hour: Mon 2pm-3pm
Email: zf87 AT cs dot rutgers dot edu
10/19/2017

Topics

- Processes (ps/top)
- Forking (pid/ppid)
- HW4 & 5

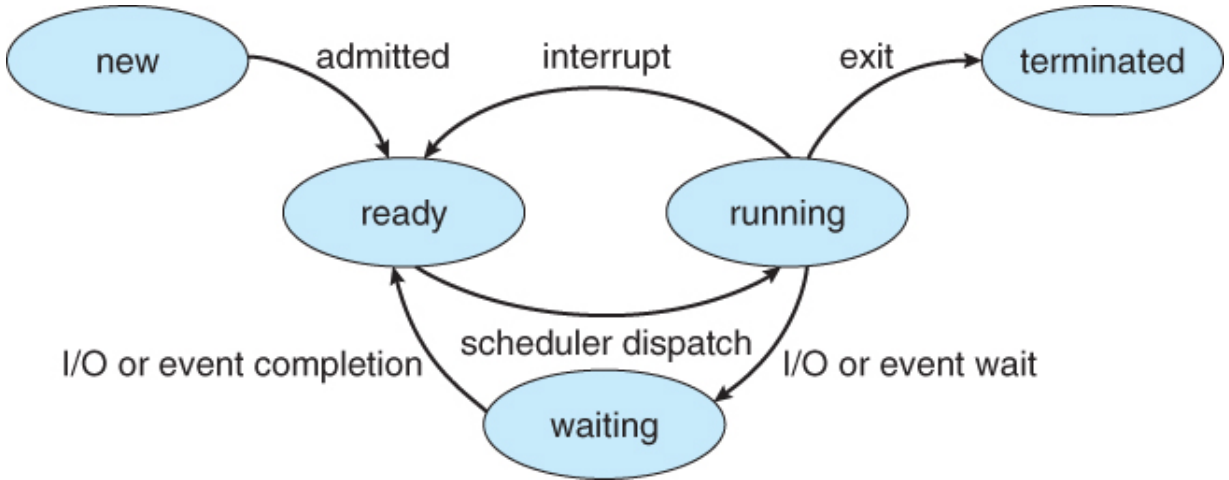
Process

- A process is basically a program in execution. It is fundamentally a container that holds all the resources needed to run a program.
- 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 memory space, while processes run in separate memory spaces.



Process state

- Five states in general, with specific operating systems applying their own terminology and some using a finer level of granularity:
  - New — process is being created
  - Running — CPU is executing the process's instructions
  - Waiting — process is, well, waiting for an event, typically I/O or signal
  - Ready — process is waiting for a processor
  - Terminated — process is done running



Command-ps

- The ps utility displays a header line, followed by lines containing information about all of your processes that have controlling terminals.
  - -a show all users' processes which have controlling terminals
  - -e show all users' processes
  - -x When displaying processes matched by other options, include processes which do not have a controlling terminal.
  - -f show detailed information u show user information for processes

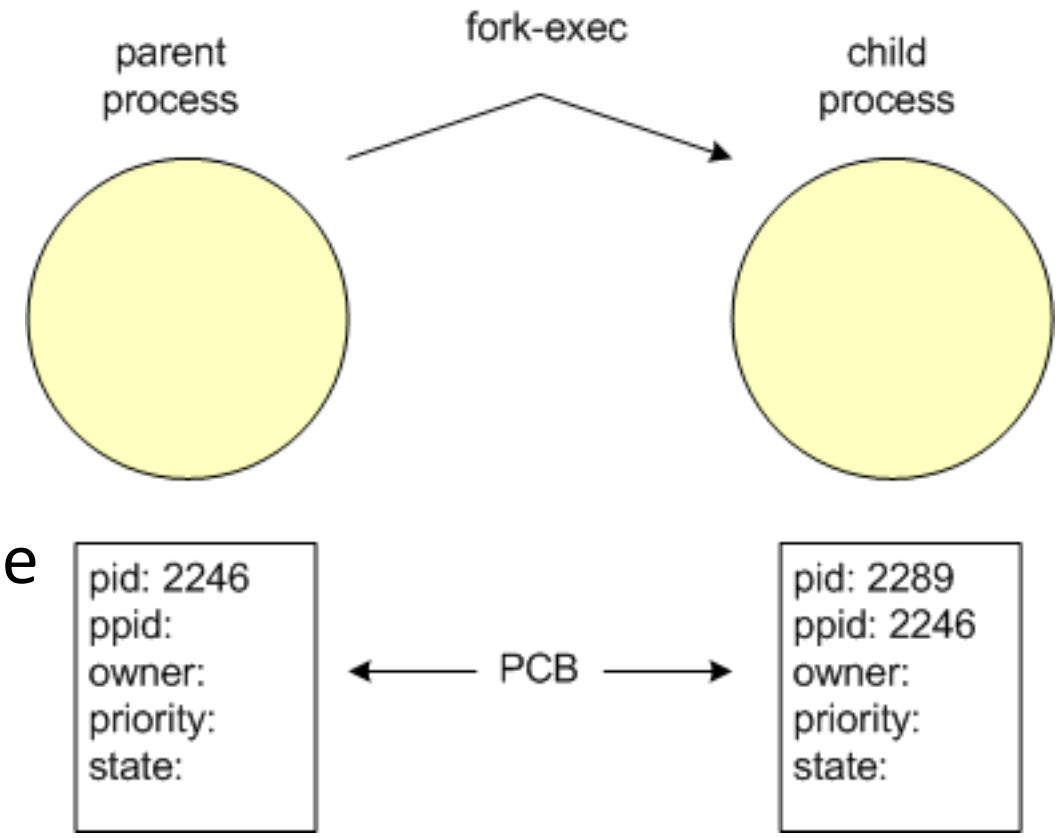
Command-top

- “top” command display and update sorted information about processes
  - -o key Order the process display by sorting on key in descending order. A + or - can be prefixed to the key name to specify ascending or descending order, respectively.
  - -O skey Use skey as a secondary key when ordering the process display
  - -s delay-secs Set the delay between updates to delay-secs seconds. The default delay between updates is 1 second
  - -n nprocs Only display up to nprocs process

https://www.tecmint.com/12-top-command-examples-in-linux/

Fork

- A system call that creates a new process identical to the calling one
  - Makes a copy of text, data, stack, and heap
  - Starts executing on that new copy
- The duplicate (child process) and the original (parent process) both proceed from the point of the fork with exactly the same data.
- The only difference is the return value from the fork call.



PID & PPID

- https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

System call: `int fork(void)`

- If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child;
- If `fork()` fails, it returns -1 to the parent (no child is created) and sets `errno`

Related system calls:

- `int getpid()` – returns the PID of current process
- `int getppid()` – returns the PID of parent process (ppid of 1 is 1)

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

Different return values

- http://www.cs.toronto.edu/~krueger/csc209h/lectures/Week7-Processes.pdf

Original process (parent)

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);
/* prints 5 */
pid = fork();
/* pid == 677 */
if (pid > 0)
    i = 6;
else (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 6 */
```

Child process

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);

pid = fork();
/* pid == 0 */
if (pid > 0)
    i = 6;
else if (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 4 */
```

HW4.0

- Using opendir and readdir, open the current directory and output all filenames until there are no more

```
char * base = "./";
DIR * thingy = opendir(base);
dirent * newfile = readdir(thingy);
```

4.1

- Parse the dirent struct to see if an entry is a directory or a file. If it is a file, prepend "./" to the filename, if it is a directory, don't.

..

if newfile != NULL //check type field of newfile dirent struct to determine the type of this file endpoint

newfile->d_type // compare with system defines for different endpoint types (3rd paragraph under 'NOTES' in man 3 readdir).

...

if == DT_REG //regular file

elseif == DT_DIR //directory ....

4.2

- Open a file handle to each file and use lseek to determine the file's size in bytes, and print out the file's size next to its name.

//assemble name of file using base directory and current path/name // concatenate all path up until now...

strcat(newpath, base)

// add currend name if it is a file...

newerpath = realloc(newpath, strlen(newpath)+strlen(newfile->d_name)); // d_name is REQUIRED to have a terminating null byte by standard ... yippee!

strcat(newerpath, newfile->d_name);

int checkFD = open(newerpath, RD_ONLY);

... if no error...

int len = lseek(checkFD, 0, SEEK_END);

close(checkFD);

printf(filename with full path, either color to indicate file/dir or put a "/" at the end to indicate dir, and number of bytes of size, if a file) //be sure to closedir() when done with dir descriptor

4.3

- Add a recursive element. If you find a directory, recursively call your code on that directory and prepend that directory name to each filename and directory name outputted.

```
elseif newfile->d_type == DT_DIR
strcat(newpath, base) // add currend name if it is a file...
newerpath = realloc(newpath, strlen(newpath)+strlen(newfile->d_name));
... recursively call my_LS on newerpath
```

HW5.0--Implementing "ps"

- 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.

Reference:

- char * base = "/proc"
  DIR * stuff = opendir(base, RD_ONLY)
  dirent * pidnumber = NULL;

  char * newCmdline = NULL;
  char * workingName = NULL;

  int fd = -1;

  while
  {
  pidnumber = readdir(stuff);
  if(pidnumber != NULL && pidnumber->d_type == DT_DIR) { newCmdline = ... malloc strlen(base)+strlen(pidnumber->d_name)+9;
  newCmdline[0] = '\0';
  strcat(newCmdline, base);
  strcat(newCmdline, "/");
  strcat(newCmdline, pidnumber->d_name);
  strcat(newCmdline, "/cmdline"

  fd = open(newCmdline, RD_ONLY);

  ... read loop ...
  .. .while read from fd != 0, printf it out ...

  close(fd);

  }

  }
  do(pidnumber != NULL);

  ... this gets you all command lines run for all pids for all procs on the system

  printf(command run and its pid(i.e. directory name of cmdline))

5.1

- 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.

  open status alongside/after cmdline (but before clocking your readdir loop! readdir is destructive!) read through and parse the status file looking for 'uid'

reference:

- while reading in status file ...
  if buffer[i] == 'u'
  if bufferLength - i >= 2
    if (buffer[i+1] == 'i' && buffer[i+2] == 'd')
    ... can start reading in uid that called this code ... w00t!

  printf(command run, its pid and the uid that called it) .. boring .. want userNAME, not UID .. :^P .. but only place in system this information is together is in passwd file ... very well, then...

  struct passwd * getUname = getpwuid(UID parsed out of status above)

  new can print out:
    command run (from cmdline file)
    username that called it (passwd->pw_name)

  .. for every current PID

  Congrats .. you wrote basic ps

5.2

- Then open the file schedstat and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a runqueue, # of times context switched(be careful of decimals! you may want to check the status file to be sure your degree is correct)

  Then, for some fun times, print out some stats.

  Maybe also add command line options! 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 stuff whose uid matches