CS 450: Operating Systems
Lecture 3: Processes

Spring 2014, J. Sasaki

Dept of Computer Science

Illinois Institute of Technology

Quick Review of Processes

Process: Program in Execution

- A ***process*** (a.k.a. job, task) is a program in action: has a program counter(s), owns resources. Fundamental unit of work.
- A process can create child processes, wait for them to die ….
- A program describes how a process acts
  - Something like a drawing of a person versus the actual person

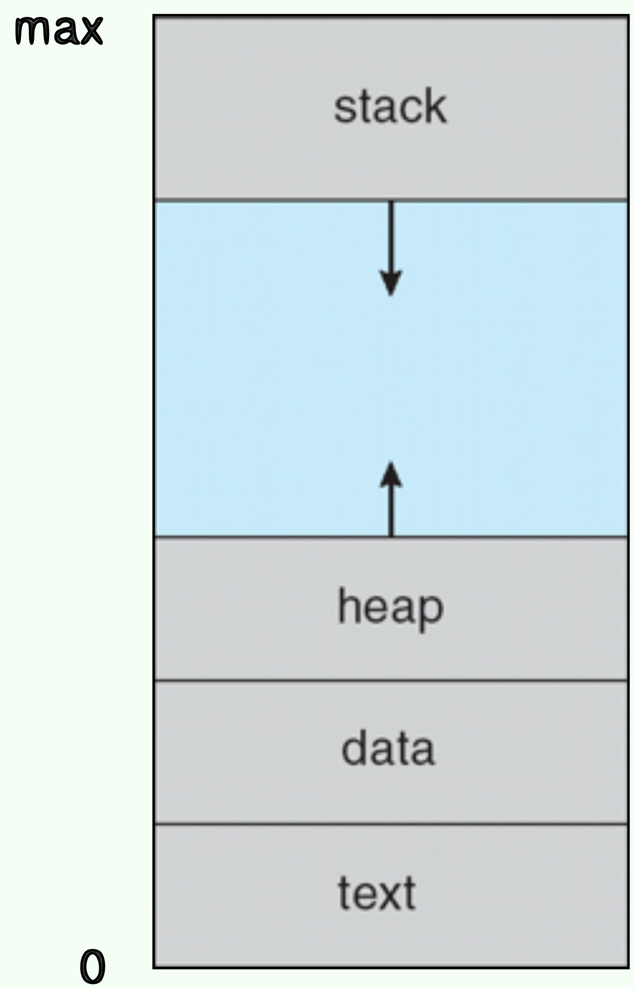
Play/Performance Analogy

- Another analogy for understanding processes vs programs:
  - A ***program*** is like a play ***script***.
  - A ***process*** is like a play ***performance***
- A performance is the activity of carrying out the instructions in the script.
- For a performance we need script + resources
  - Stage, actors, props // Memory, CPUs, data

Parts of a Process

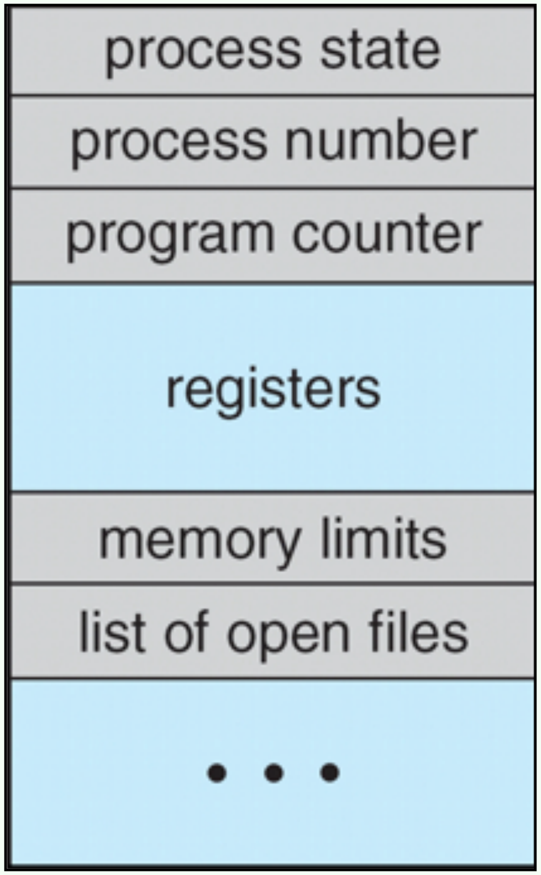
- Process is more than program code
  - Program code = text section
  - Runtime stack
  - Global variables = data section
  - Heap: Dynamically-allocated data
  - Processor info

Process in Memory



Process Control Block

- Conceptually, all information for a process is combined into a structure.
- PCB: Information for a process



Process Representation in Linux

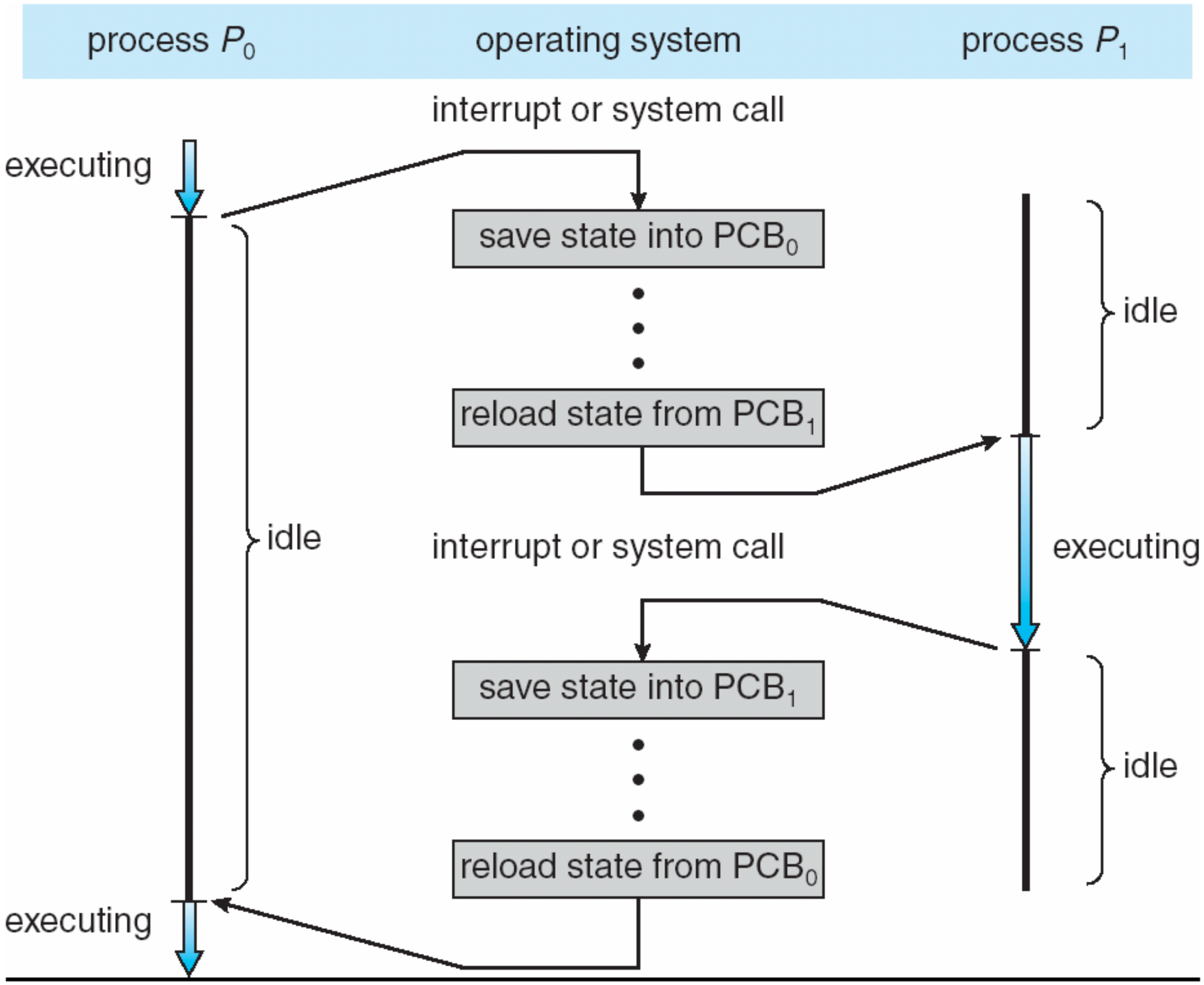
- C structure `task_struct`

```
pid_t pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children;  /* this process's children*/
struct files_struct *files; /* list of open files */
struct mm_struct *mm;       /* address space of this
                               process */
```

Context Switches

- To change running processes, save current PCB, load new PCB
  - Overhead
  - PCB complexity ↑ — Switch time ↑
  - Hardware support?

Context Switching



C Process Examples

In C: fork(), exec(), wait()

- C library; use ***fork()*** to create process
  - Child process is a copy of the parent.
  - Parent gets pid of child; child gets pid 0.
  - Process can change program via ***exec()***
  - Parent can ***wait()*** for child to terminate

C Example: Lec03_proc1.c

```
#include <stdio.h>
#include <unistd.h> // for fork, execlp

int main()
{
    pid_t pid;       // Process id
    pid = fork();    // Fork child process

    // If pid < 0, an error occurred
    //
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
```

```
// If pid < 0, an error occurred
//
if (pid < 0) {
    fprintf(stderr, "Fork Failed\n");
    return 1;
}

// If pid > 0, we're the parent.
// Wait for the child to finish
//
else if (pid > 0) {
    wait(NULL);
    fprintf(stderr,
      "Parent says: Child process %d complete\n",
       pid );
}
```

```
    // If pid = 0, we're the child.
    // Execute an ls command and quit
    //
    else  {
      fprintf(stderr,
          "Child says: I'm %d\n", getpid());
        int error
            = execl("/bin/ls", "ls", "-l", NULL);
        // execl only returns if an error occurs
        fprintf(stderr,
          "Child says: ",
          "exec returns with %d\n", error);
    }
    return 0;  // parent finishes
}
```

Sample Output

```
Child says: I'm 1310
total 40
-rw-r--r--@ 1 jts  jts   734 Jan 21 15:52 Lec03_1_proc.py
-rw-r--r--@ 1 jts  jts   820 Jan 22  2013 Lec03_2_proc.c
-rwxr-xr-x  1 jts  jts  9036 Jan 21 15:55 a.out
Parent says: Child process 1310 complete
```

Timed Wait: Lec03_proc2.c

```
#include <stdio.h>
#include <unistd.h>    // fork
#include <stdlib.h>    // exit
#include <sys/errno.h> // global error number
#include <sys/wait.h>  // wait

// Prototype for child processes
   pid_t child(int child_nbr, int sleeptime);

int main(int argc, char *argv[]) {
    pid_t pid_child1, pid_child2, exited_child;
    int i, child_nbr, status;
```

```
// Child 1 should sleep 3 sec; child 2 7 sec
 //
 pid_child1 = child(1, 3);
 pid_child2 = child(2, 7);
 printf("This is the parent\n");

 // Wait twice: Each time, wait for a
 // child and print out its nbr & status
 for (i = 0 ; i < 2 ; i++) {
   exited_child = wait(&status);
   child_nbr
       = (exited_child == pid_child1 ? 1 : 2);
   printf("Child %d pid %d exited with status %d\n",
       child_nbr, exited_child, status );
 }
 return 0;
}
```

```
// Child prints its pid and sleeps for a number
// of seconds.
pid_t child(int child_nbr, int sleeptime) {
    pid_t pid = fork();

    if (pid > 0) {  // Parent returns
        return pid;
    }
    else if (pid == -1) {  // Fork failed !?
        fprintf(stderr, "Fork %d failed with error %d\n",
            child_nbr, errno );
        exit(1);
    }

    // We're the child process
    printf("Child %d, pid %d\n", child_nbr, getpid());
    sleep(sleeptime);
    exit(0);
}
```

Child is Copy of Parent

- The address space of the parent is duplicated in the child.
- Each process sees its data at the same locations, but the spaces are not shared.
- Changes to the child's address space aren't reflected in the parent

Example: Lec03_proc3.c

```
#include <stdio.h>
#include <unistd.h>    // fork
#include <stdlib.h>    // exit
#include <sys/wait.h>  // wait

// Child will get duplicate of parent's address
// space, so its global variable will be at the
// same location, but in its own space, not the
// parent's
//
int glovar = 1;
```

```
int main(int argc, char *argv[]) {
   pid_t pid = fork();

   // Parent prints global var, waits for
   // child to finish, then reprints global var
   //
   if (pid > 0) {
      fprintf(stderr,
           "Parent: &glovar: %p, glovar: %d\n",
          &glovar, glovar );
      fprintf(stderr,
           "Wait for child with pid %d\n", pid );
      wait(NULL);
      fprintf(stderr,
           "Parent: glovar: %d\n", glovar );
      return pid;
   }
```

```
   // Child changes global variable then returns
   //
   else if (pid == 0) {
      glovar = 1234;
      fprintf(stderr,
           "Child:  &glovar: %p, glovar: %d\n",
           &glovar, glovar );
      exit(0);
   }

   // Complain if fork failed
   //
   else if (pid == -1) {
      fprintf(stderr, "Fork failed\n");
      exit(1);
   }
}
```

Sample Output

```
Parent: &glovar: 0x10b681068, glovar: 1
Wait for child with pid 2457
Child:  &glovar: 0x10b681068, glovar: 1234
Parent: glovar: 1
```

Python Process Examples

Python Example: Lec03_proc4.py

```
from multiprocessing import Process

def go_proc():
    for i in range(5):
        # Create each process, have it run say_hello(i)
        # then print child's process id
        #
        p = Process(target=say_hello, args=([i]))
            # (Note list of argument values)
        p.start()
        print('started process {}'.format(p.pid))

def say_hello(id):
    print('hello from child {}'.format(id))

go_proc() # run the main program
```

Sample Output

```
> python3 Lec03_proc4.py
started process 1628
started process 1629
hello from child 0
started process 1630
hello from child 1
started process 1631
hello from child 2
started process 1632
hello from child 3
hello from child 4
>
```

Python Example: Lec03_proc5.py (Address Space)

```
from multiprocessing import Process

glovar = 1

def go_proc():
   for i in range(5):
      # Create each process, have it run say_hello(i)
      # then print child's process id & our global var
      #
      p = Process(target=say_hello, args=([i]))
          # (Note list of argument values)
      p.start()
      print('Started process {}'.format(p.pid))
      print('glovar = {}'.format(glovar))
```

```
# Each child prints global var before and after
# setting it to 2 * its process id
#
def say_hello(id):
   global glovar
   glovar_init = glovar
   glovar = id * 2 # twice our process id
   print('Child {}, glovar was {}, setting it to
{}'.format(id, glovar_init, glovar))

go_proc()   # run the main program
```

Sample Output

```
Started process 2886
glovar = 1
Started process 2887
glovar = 1
Child 0, glovar was 1, setting it to 0
Started process 2888
glovar = 1
Child 1, glovar was 1, setting it to 2
Started process 2889
glovar = 1
Child 2, glovar was 1, setting it to 4
Started process 2890
glovar = 1
Child 3, glovar was 1, setting it to 6
Child 4, glovar was 1, setting it to 8
```

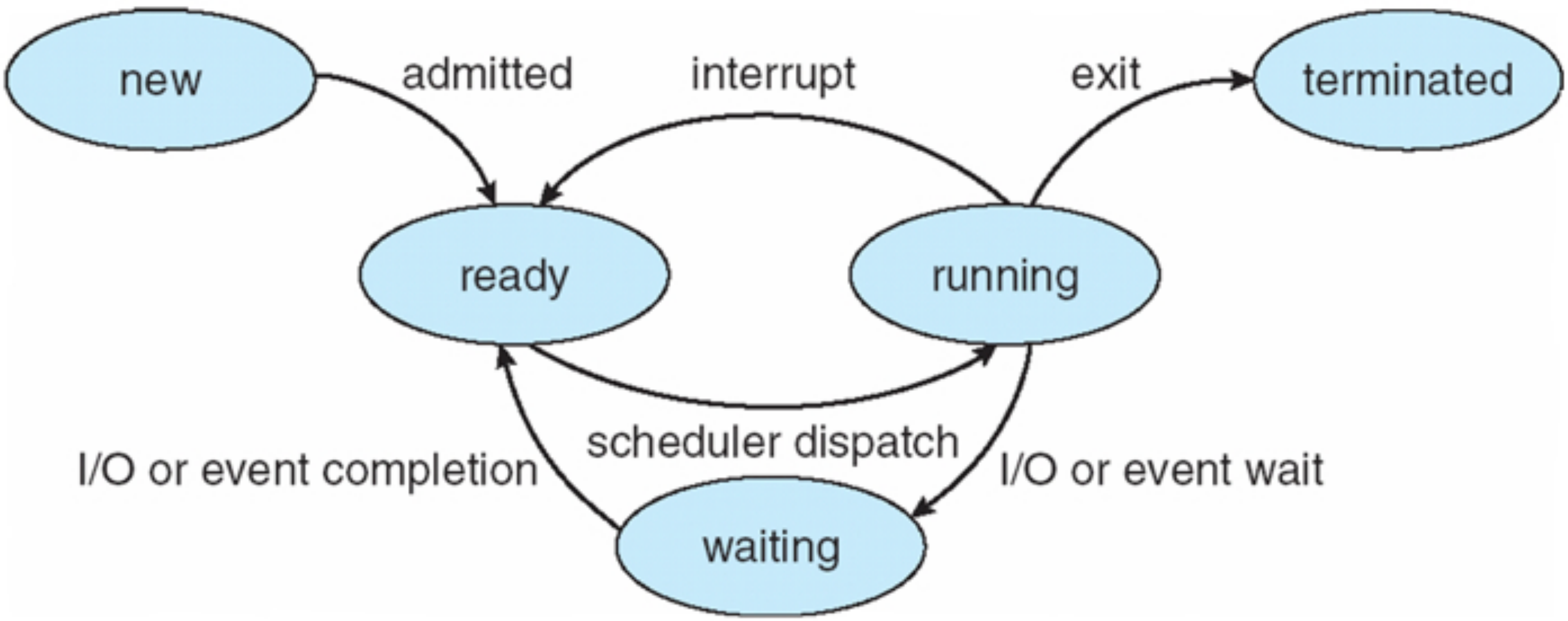
Process States and Transitions

Process States

- A process has a life cycle, various states during that life cycle.
  - New
  - Running
  - Waiting
  - Ready
  - Terminated

Process State Transitions

(Without virtual memory)



Process States with Virtual Memory

- Ready/**in-memory** & Waiting/**in-memory**
  - Add **Ready/swapped-out** and **Waiting/swapped-out**.
  - Add swap-in/swap-out transitions
- Waiting/in ↔ Waiting/out transition: Why? How?
- Ready/in ↔ Ready/out transition: Good? Bad? Ugly?