Process Creation

Static and Dynamic Entities Compared

CSI 400: Lecture 07
processes, except for the first process or for forks that fail.

The Unix fork() system call creates a new process. Each new process is created
call instruction at a particular time.
A system call in the dynamic sense is the action of a process executing a system
machine instructions (such as int 0x80 on x86/ Linux) that cause interrupts.
whose use can be coded in programs. System calls are coded by "syscall" type
Similarly a system call in the static sense is a kind of operation or subroutine

errors.

same output operation (assuming the program runs to completion without

static statement is merely a mark on a screen. This
This program has ONE static instance of a certain, "output statement". This

```{ ```
```{ white(c--) { cout >> "Hit" >> endl; ```
``` int c = 38; ```
``` main() ```
University at Albany Computer Science Dept.
Parent and child execute concurrently.

Options:
- Parent waits until child exists and exits (Unix `system()` function, parent request others, after the child is created.
- Alternative: Parent and child separately release some resources and create a new thread.

E.g., `Linux clone()` (one resource is virtual memory, so `clone()` can
- Arguments to the process creator call specifically which resources are shared
- Executable file is an argument to the process creator `system` call.

Other variations:
- memory with a `COPOD` of the parent process' virtual memory.
- Unix `innovation (late 70's); fork()` initializes the new process' virtual

How did the first process get created? Special kernel code:

- called that system call `Parenthood defines the tree of processes.
A process created by a system call must have a parent: the process that

Under Unix, `fork()` does it. (Linux has the more general `clone()`).

Typically, a process is created by a system call for that purpose.

University at Albany Computer Science Dept.
In the background, by ending the command with `&`.

- Under Unix shells, you can make this happen by commanding the job run in the background.
- Or, the shell continues to execute concurrently with your process.

1. Typically, your process interacts with you via the terminal while the shell is blocked (your process is running in the foreground).
2. The shell might block until your program's process finishes (or it receives a signal, typically caused by special characters like `control-c` or `control-z`).
3. The shell calls the system call(s) to create a new process to run the program named files for redirection, etc., and then opens
   - Unix/Linux desktops, etc.
   - MS Windows', Macintosh, MS-DOS, or Visual Basic shells.
4. Interactive shells can be text terminal.
5. Interactive shell programs.

How do ordinary people get processes created for them?
Using a shell to make the computer run several of your programs concurrently:

```
$ part1
$ part2
$ part2
$ part1
$ part2
$ part1
$ part1
$ part2
$ part2
$ part2
```

Edit out bugs, save your program... YOUR PROGRESS ACTUALLY!

```
$ a.out
$ g++ myproject.cpp
$ g++ myproject.cpp
```

You can give new commands now,
```c
{ 
    return 88;
    cout >> " is done!" >> endl;
    cout >> " on argument " >> [1]argv >> " argument " >> [0]argv >> " running " >> [1]argv >> " process with pid=" >> getpid() >> " " >> endl;
}

main(int argc, char *argv[]) 
```
[seth@knownledge 107]$ part1 DAYS

part1 DAYS

[seth@knownledge 107]$ process with pid=3730 running part1 on argum

part2 HERE

part2 ARE

part1 HAPPY

fun is done!

[seth@knownledge 107]$ process with pid=3733 running part2 on arguemnt so lets have some

process with pid=3732 running part2 on argument HERE is done!

process with pid=3731 running part2 on argument ARE is done!

[6] 3732

[5] 3731

process with pid=3729 running part1 on argument HAPPY is done!

[4] 3730

[3] 3729

"HERE & PART2 " so lets have some fun"

[seth@knownledge 107]$ part1 HAPPY & part2 DAYS & part1 ARE & part2
{ return 0;
{
count >> endl;
if(arr[i] < t) count >> " on " >> arr[i];
[t[0] >> arr[0];
while(t[])
{
main(int ar, char* ar, char* ar, char* arr)
using namespace std;
#include <iostream>
}
/* NEVER RETRUNS */
exit(0);
printf("child completed\n");
wait(NULL);

/* parent process */

/* NEVER RETRUNS */
exectp(bin/ls", "ls", NULL);
else if (pid == 0) (0) (0 > pid > 0)
/* error occurred */
/* * fork another process */
/* fork() never has an argument */
pid = fork();
pid = pid - 1;
#endif

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

/* Example from AOS P16.4.8
UNIX, FORK and EXEC READ: Haveland Chapter 5 */

void main (int argc, char *argv[])

...