Programs and Processes

- Program: Executable binary (code and static data)
- Process: A program loaded into memory
  - Program (executable binary with data and text section)
  - Execution state (heap, stack, and processor registers)

Program

```
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```

Process

```
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```

Heap

Stack

Registers

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
- Uses of fork()
  - To create a parallel program with multiple processes (E.g. Web server forks a process on each HTTP request)
  - To launch a new program using exec() family of functions (E.g. Linux shell forks an ‘ls’ process)

fork() example

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  int seq = 0;
  if(fork()==0)
  {
    printf("Child! Seq=%d\n", ++seq);
  }
  else
  {
    printf("Parent! Seq=%d\n", ++seq);
  }
  printf("Both! Seq=%d\n", ++seq);
  return 0;
}
```

```
>> ./a.out
Parent! Seq=1
Both! Seq=2
Child! Seq=1
Both! Seq=2
```

fork() example

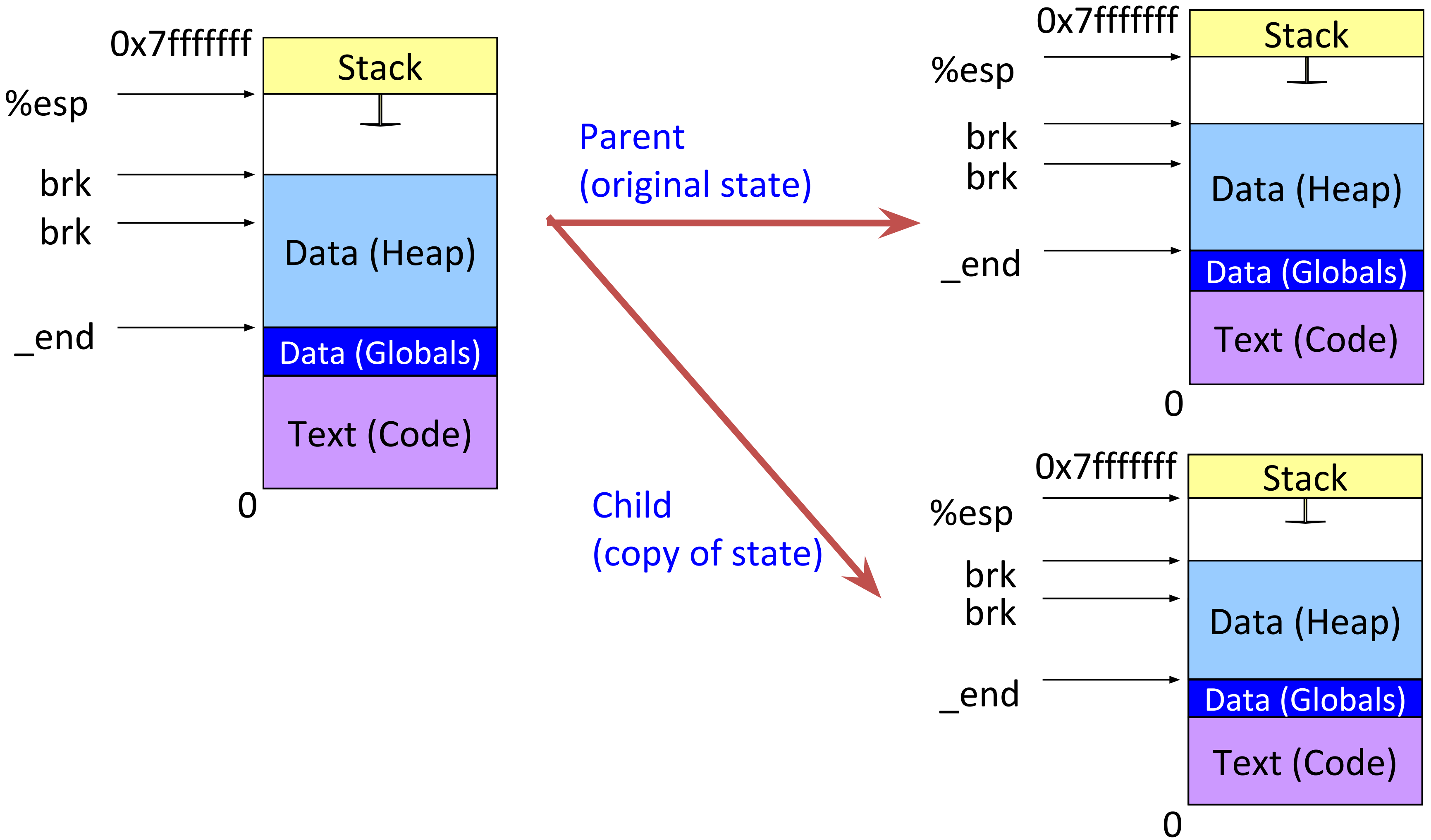
```c
#include <stdio.h>
#include <unistd.h>
int main()
{
  int seq = 0;
  if(fork()==0)
  {
    printf("Child! Seq=%d\n", ++seq);
  }
  else
  {
    printf("Parent! Seq=%d\n", ++seq);
  }
  printf("Both! Seq=%d\n", ++seq);
  return 0;
}
```

```
>> ./a.out
Parent! Seq=1
Both! Seq=2
Child! Seq=1
Both! Seq=2
```

- Differentiate child and parent using return value of fork()
  - "Child" process return value is 0
  - "Parent" process gets child's process id number
- **Copies** execution state (not shares)
  - Child copies stack variable seq onto its own stack
  - Child / parent has own copy of seq

Before fork()

After fork()



Copy-On-Write

- Inefficient to physically copy memory from parent to child
  - Code (text section) remains identical after fork
  - Even portions of data section, heap, and stack may remain identical after fork
- Copy-On-Write
  - OS memory management policy to **lazily** copy pages only when they are modified
  - Initially map same physical page to child virtual memory space (but in read mode)
  - Write to child virtual page triggers page protection violation (exception)
  - OS handles exception by making physical copy of page and remapping child virtual page to that page

How do Parent / Child Run in Parallel?

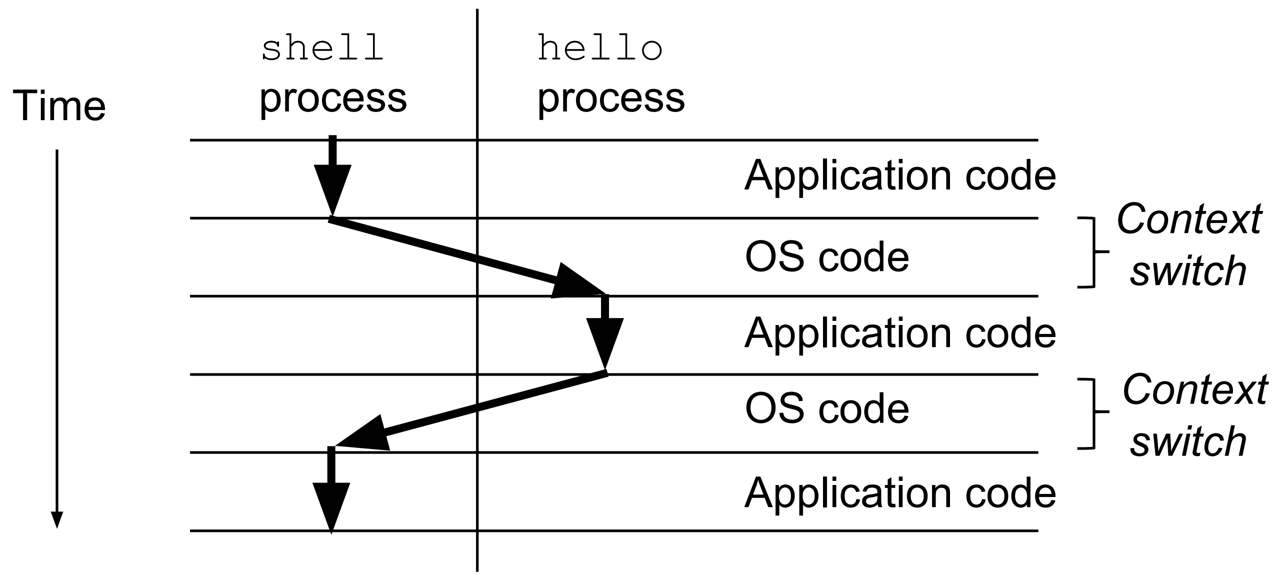
- Obvious answer: by running on different processors on a multiprocessor system
- What if it’s a uniprocessor system?
- What if other processors are already busy?
- Answer: by context switching (running each process in short time slots)

Process Context

- Context: set of data that must be saved by a task on interruption and restored on continuation
- Process Context
  - Process registers (including program counter, stack pointer etc.)
  - Memory management info (pointer to page table etc.)
  - I/O status info (pointer to file descriptor table etc.)
  - Process scheduling info (scheduling priority etc.)
  - Process ID, or PID (unique identifier for process)
- Data structure in OS containing process context information is called **process control block (PCB)**

Context Switching

- OS provides the illusion of a dedicated machine per process
- Context switch
  - Saving context of one process, restoring that of another one
  - Processor alternates between executing different processes



Dispatch Mechanism

- OS maintains list of all processes (PCBs)
- Each process has a mode
  - Running: Executing on the CPU
  - Ready: Waiting to execute on CPU
  - Blocked: Waiting for I/O or synchronization with another thread
- Dispatch Loop

```
while (1) {
    run process for a while;
    stop process and save its state;
    load state of another process;
}
```

How does dispatcher gain control?

- Must change from user to system mode
  - Problem: Only one CPU, and CPU can only do one thing at a time
  - A user process running means the dispatcher isn't
- Two ways OS gains control
- Exceptions: Events caused by process execution
  - System calls, page faults, segfault, etc
- Hardware interrupts: Events external to process
  - Typing at keyboard, network packet arrivals
  - Control switch to OS via Interrupt Service Routine (ISR)
- How does OS guarantee it will regain control?
  - By setting a timer interrupt (allows fair scheduling of processes)

Spawning a New Program

- Combination of fork() and exec(…)
  - fork(): Clone current process
  - exec(…): copy new program on top of current process
- Exec(...) family of functions
  - execl, execlp, execle, execv, execve, execvp
  - User space wrappers for the **execve** system call
  - Also called the “program loader”
    - Loads in text and data sections of a binary executable
    - Links in any shared objects and perform relocations
    - Sets up stack and starts executing
  - What Linux shell calls when launching a program

execvp() example

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  if(fork()==0)
  {
    char *args[3] = {"ls", "-al", NULL};
    execvp(args[0], args);
    // DOES NOT GET HERE
  }
  else
  {
    printf("Parent!\n");
  }
  printf("Only parent!\n");
  return 0;
}
```

```
>> ./a.out
Parent!
Only parent!
drwx------  4 wahn UNKNOWN1    4096
Oct 21 08:13 .
drwxr-xr-x 10 wahn UNKNOWN1    2048
Oct 21 08:13 ..
-rwxr-xr-x  1 wahn UNKNOWN1    6743
Oct 21 08:12 a.out
```

- execvp never returns since memory is overwritten using another program image

Managing processes (Unix)

- Finding processes
  - ‘ps’
- Monitoring Processes
  - ‘top’
- Stopping processes
  - ‘kill <pid>’ (for a soft kill using SIGTERM)
  - ‘kill -9 <pid>’ (for a hard kill using SIGKILL)
- Procfs (/proc/)

Using ‘ps’

- Listing processes associated with this terminal

```
(84) thot $ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
wahn     11848 11847  0 06:27 pts/1    00:00:00 -bash
wahn     15754 11848  0 11:24 pts/1    00:00:00 ps -f
```

- Listing all processes

```
(85) thot $ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Aug26 ?        00:00:11 /sbin/init
root         2     0  0 Aug26 ?        00:00:00 [kthreadd]
root         3     2  0 Aug26 ?        00:00:03 [migration/0]
root         4     2  0 Aug26 ?        00:00:04 [ksoftirqd/0]
root         5     2  0 Aug26 ?        00:00:00 [migration/0]
```

Using ‘kill’

- Killing your own shell

```
(51) thot $ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
wahn     11848 11847  0 06:27 pts/1    00:00:00 -bash
wahn     15904 11848  0 11:36 pts/1    00:00:00 ps -f
(52) thot $ kill 11848
(53) thot $ kill -9 11848
Connection to thot.cs.pitt.edu closed.
```

Procfs

- File system based export of process information
  - Mounted on /proc/
  - Contains information on every process on the system
  - Organized by PID
  - **/proc/self** exports information for the running process

- Information available
  - Resources in use
  - Resources allowed to use
  - Virtual memory map
  - Cmdline
  - Etc…

Procfs usage examples

- Listing open file descriptors for process

```
(84) thot $ ls -l /proc/self/fd
total 0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 0 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 1 -> /dev/pts/0
lrwx------ 1 wahn UNKNOWN1 64 Sep 11 13:37 2 -> /dev/pts/0
lr-x------ 1 wahn UNKNOWN1 64 Sep 11 13:37 3 -> /proc/16970/fd
```

- Showing virtual memory map

```
(85) thot $ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 fd:00 2681          /bin/cat
0080a000-0080b000 rw-p 0000a000 fd:00 2681          /bin/cat
0080b000-0082c000 rw-p 00000000 00:00 0             [heap]
34ff600000-34ff78b000 r-xp 00000000 fd:00 131       /lib64/libc-2.12.so
34ff98e000-34ff98f000 rw-p 0018e000 fd:00 131       /lib64/libc-2.12.so
7ffffffea000-7ffffffff000 rw-p 00000000 00:00 0     [stack]
```