Fork() System Call and Processes

CS449 Fall 2016

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)
  - OS state (open file descriptors, current file offsets, etc.)

Program

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

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

Process

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

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

fork(): how processes are born

- **fork()**: A system call that creates a new process identical to the calling one
  - **Makes a copy** of text, data, stack, and heap
  - Starts execution on that new copy in parallel with old copy
  - Old copy: parent process, new copy: child process
- 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 (E.g. Linux shell forks an 'ls' process)

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
```

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

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

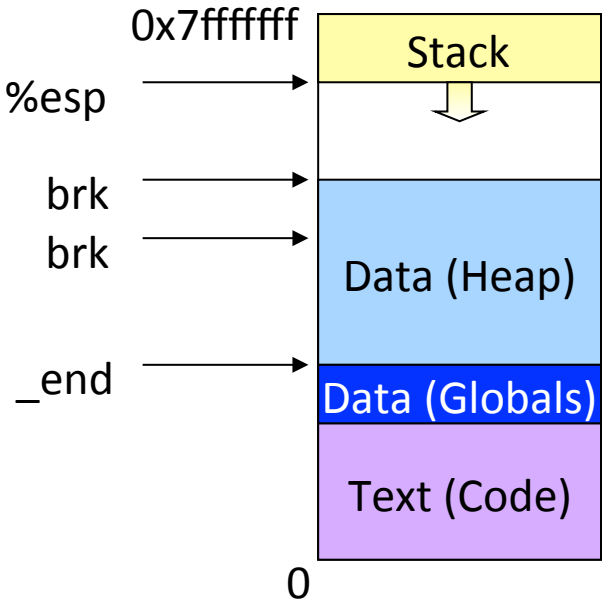
fork() example

```
#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
```

- Differentiate child and parent using return value of fork():
  - For Child: 0
  - For Parent: child's process id
- **Copies** execution state (not shares)
  - Child has its own stack with own copy of seq variable
  - Updates to seq in child not reflected on parent (vice versa)

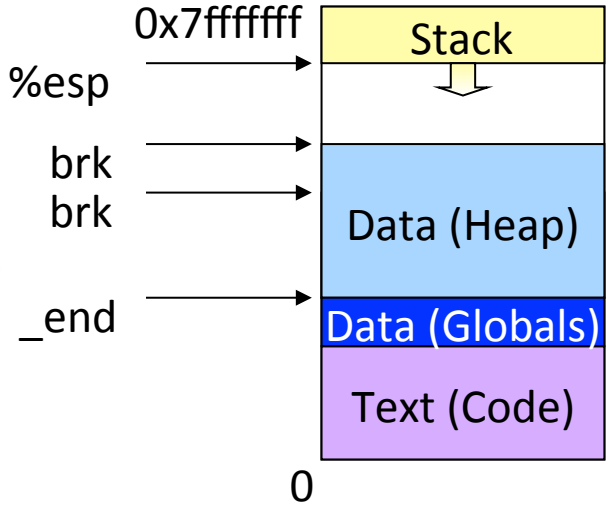
Before fork()

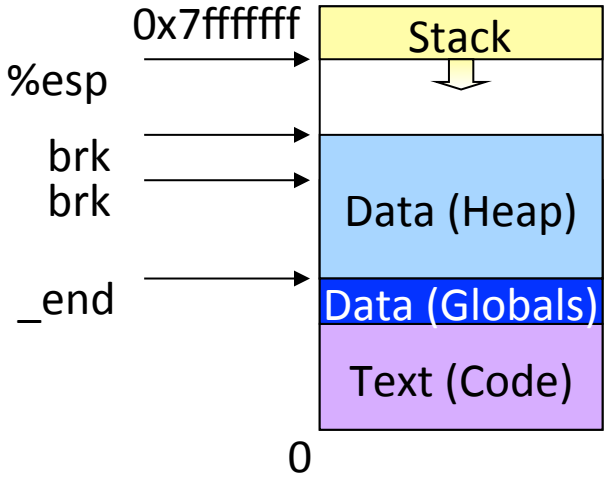


Parent
(original state)

Child
(copy of state)

After fork()





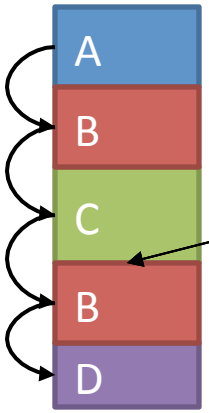
Copy-On-Write

- Inefficient to immediately copy physical 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

  1. Initially map same physical page to child virtual memory space (but setting the write protection bit in the page table)
  2. Write to child virtual page triggers page protection violation (in other words, a page fault exception)
  3. OS handles exception by making physical copy of page and remapping the written virtual page to that page

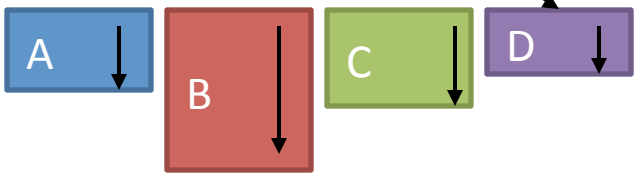
Multitasking: Concurrently Running Processes

Single $EIP
(CPU's point of view)

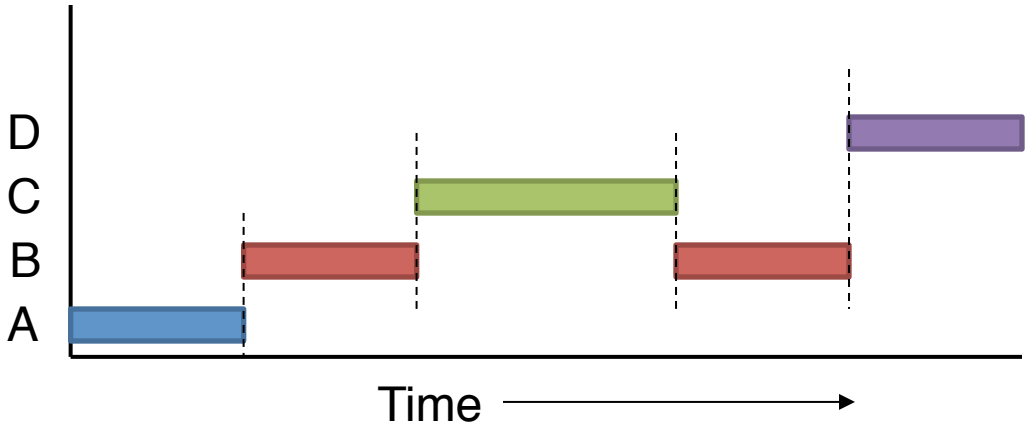


One stream of instructions

Multiple $EIPs
(process point of view)

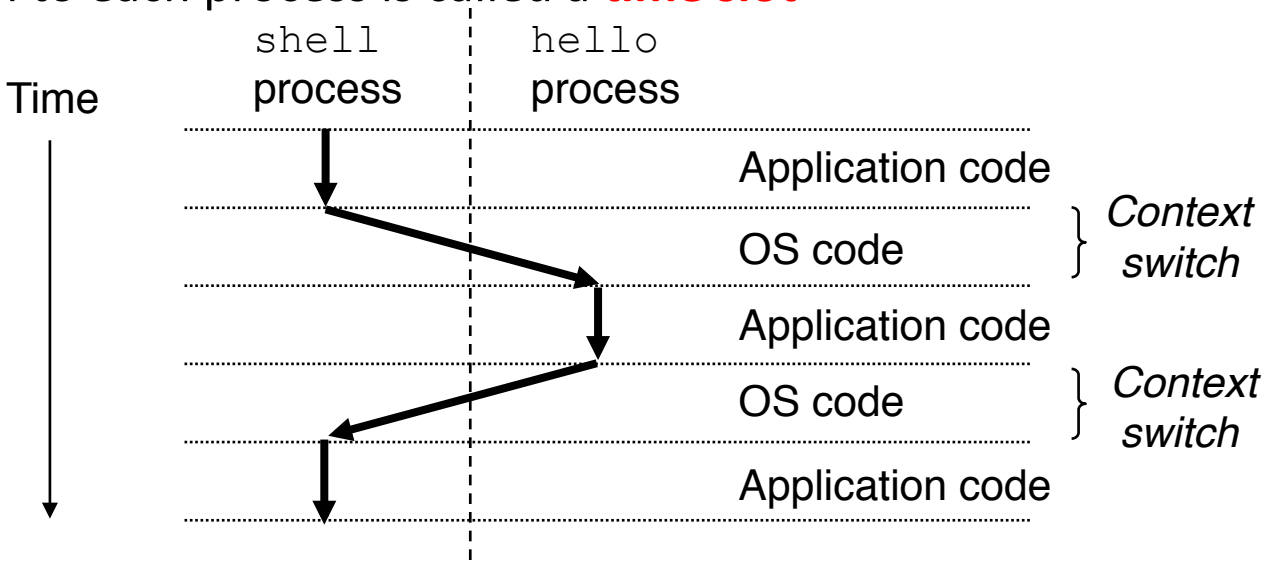


Isolated address spaces
that are not interrupted



Context Switch: Switching Tasks

- **Context switch**
  - Saving context of one process, restoring that of another one
  - Processor alternates between executing different processes
  - Period given to each process is called a **time slot**



- OS provides the illusion of a dedicated processor per process

Process Context

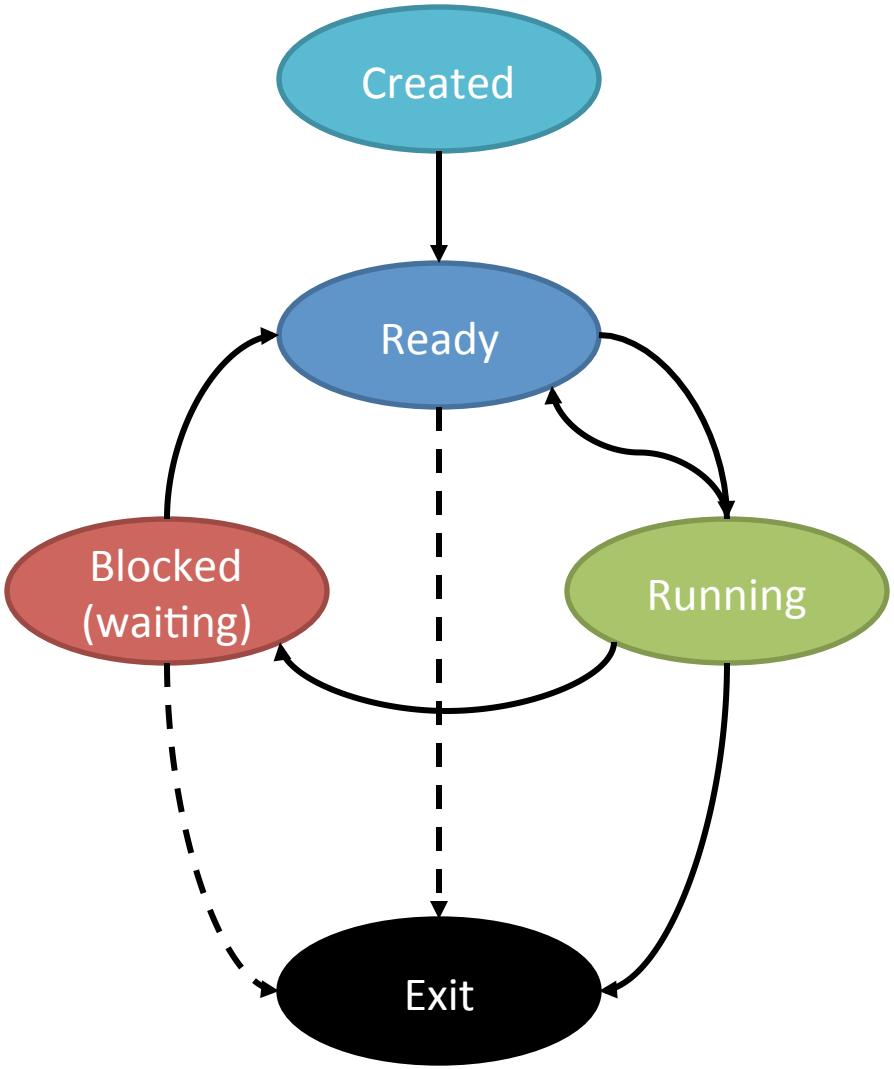
- Context: state that must be saved by a task on interruption and restored on continuation
- Process Context: Data in registers + OS state
  - Processor register values ($EAX, $EIP etc…)
  - Process ID, or PID (unique identifier for process)
  - Memory management info (pointer to page table etc.)
  - I/O status info (pointer to file descriptor table etc.)
  - Process scheduling info (scheduling priority etc.)
- What is *not* a part of context: Data in memory
- Data structure in OS containing process context information is called **process control block (PCB**)

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 ready process;
}
```

Life Cycle of a Process



When does the dispatcher run?

- Remember: Kernel is a **reactive** program
  - Only runs in response to user process request or HW event
  - Cannot rely on user process to request dispatcher to run
  - Processes can be greedy or worse malicious

- How does the kernel wrest control from a process?
  - Opportunistic method: Run dispatcher while handling…
    - Exceptions: System calls, page faults, etc
    - Interrupts: Keyboard strokes, network packet arrivals, etc.
  - Guaranteed method: Set HW timer for length of time slot
    - When timer fires, a hardware interrupt will be generated

Process / Processor Abstraction

- **Process** is to **processor** what **virtual memory** is to **physical memory**

Virtual Memory	Process
Abstracts away physical memory	Abstracts away processors
Application oblivious of the physical memory underneath it	Application oblivious of the physical processors underneath it
Allocates multiple virtual memory spaces on one physical memory space	Runs multiple processes on one processor
Swaps virtual memory in & out of hard disk when physical memory overflows	Swaps out processes in & out of processors when too many processes
OS controls mapping from virtual memory to physical memory	OS controls mapping from processes to processors

Spawning a New Program

- Sequence of fork() and exec(…)
  1. fork(): Clone current process
  2. exec(…): copy new program on top of current process
- Exec(...) family of C Library functions
  - execl, execlp, execle, execv, execvp
  - Wrappers for the execve system call
  - This is the **link loader** mentioned with dynamic linking
    1. Loads in text and data sections of a binary executable
    2. Loads in any shared objects and performs dynamic linking
    3. Starts executing from entry point given by executable header
  - What Linux shell calls when launching a program after forking

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
    printf(”Child!\n");
  }
  else
  {
    printf("Parent!\n");
  }
  return 0;
}
```

```
>> ./a.out
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 (ls) by link loader

Managing processes (Unix)

- Finding processes
  - ‘ps’, ‘pstree’
- 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

```
thoth $ 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

```
thoth $ 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 ‘pstree’

- Displays all processes in the form of a tree

```
thoth $ pstree -p
init(1)─┬─NetworkManager(1493)
        ├─abrt-dump-oops(2011)
        ├─abrtd(2001)
        ├─acpid(1681)
        .
        .
        .
        ├─sshd(20847)─┬─sshd(10995)──sshd(11007)──bash(11008)──nano(11144)
        |             ├─sshd(11125)──sshd(11847)──bash(11848)──pstree(15369)
```

- Note how all process are forked off the ‘init’ process
  - Init process: First process started after booting. Launches all other OS services (e.g. sshd) by executing an init script.

Using ‘kill’

- Killing your own shell

```
thoth $ 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
thoth $ kill 11848
thoth $ 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
  - Open file descriptors
  - Virtual memory map
  - Cmdline
  - State of process (running / ready / blocked)
  - Etc…

Procfs usage examples

- Listing open file descriptors for process

```
thoth $ 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

```
thoth $ 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
7ffffffea000-7fffffff000 rw-p 00000000 00:00 0             [stack]
```