Processes, Threads

CS 5007: Systems

Adrienne Slaughter, Joe Buck

Northeastern University

March 27, 2019

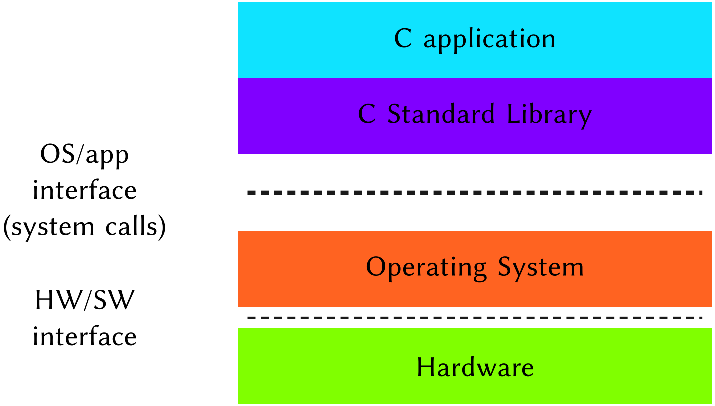
Acknowledgements

- Bryant and OH́alloran
- Helpful slides:
  - http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/14-ecf-procs.pdf
  - http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/23-concprog.pdf
- Justin Hsia
  - https://courses.cs.washington.edu/courses/cse333/18sp/lectures/27/CSE333-L27-processes_18sp-ink.pdf

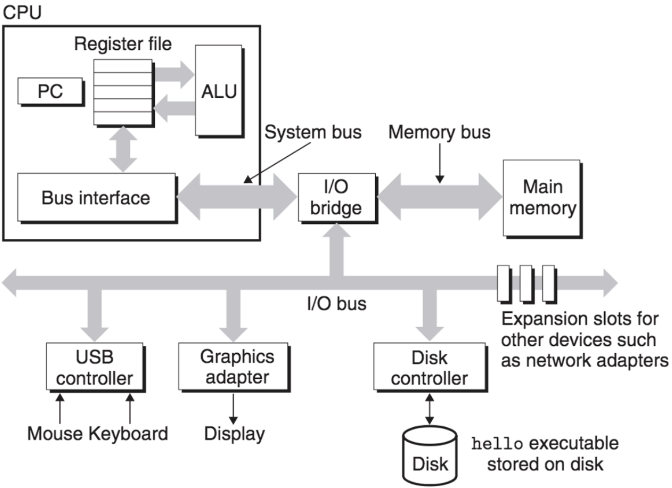
Section 1

Processes

The Big Picture: What is a System?



Hardware Organization



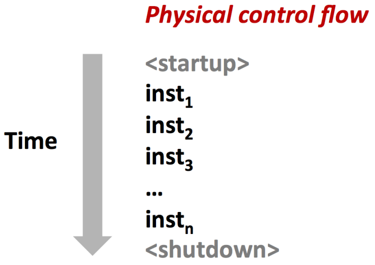
Outline

- Control flow
- Exceptions
- Processes
- Process Control
- Threads

Flow of control

Processors only do one thing:

- A processor simply reads a sequence of instructions from the start to the finish.



Flow of control

Processors only do one thing:

- A processor simply reads a sequence of instructions from the start to the finish.
- It's called the ***control flow***

Physical control flow

Time

<startup>
inst$_1$
inst$_2$
inst$_3$
...
inst$_n$
<shutdown>

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

But what happens when something changes in the ***system***?

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

But what happens when something changes in the ***system***?

- The program accidentally divides by 0

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

But what happens when something changes in the ***system***?

- The program accidentally divides by 0
- Someone enters Ctrl+C

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

But what happens when something changes in the ***system***?

- The program accidentally divides by 0
- Someone enters Ctrl+C
- ... Other system stuff we may not have talked about yet

Changing the Control Flow

We've seen ways to change the flow of control:

- Branches (if this, then that)
- Call/return (execute this other function)

These changes are part of the ***program***

But what happens when something changes in the ***system***?

- The program accidentally divides by 0
- Someone enters Ctrl+C
- ... Other system stuff we may not have talked about yet

Exceptional control flow

Processes

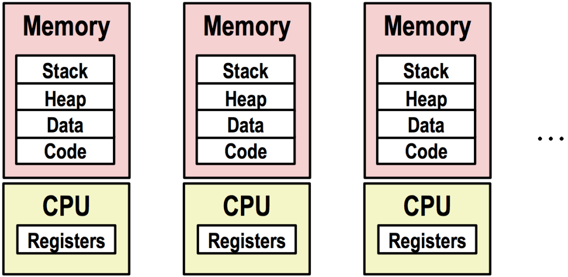
Definition: A ***process*** is an instance of a running program.

- One of the most profound ideas in computer science
- Not the same as "program" or "processor"

Process provides each program with two key abstractions:

- ***Logical control flow***
  - Each program seems to have exclusive use of the CPU
  - Provided by kernel mechanism called context switching
- ***Private address space***
  - Each program seems to have exclusive use of main memory.
  - Provided by kernel mechanism called virtual memory

The Illusion of Multiprocessing



- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network and I/O devices

```
ahslaughter
Processes: 418 total, 4 running, 414 sleeping, 3007 threads                                      14:47:33
Load Avg: 1.98, 1.94, 1.97  CPU usage: 15.72% user, 5.0% sys, 79.27% idle
SharedLibs: 148M resident, 36M data, 39M linkedit. MemRegions: 257721 total, 6052M resident, 99M private, 2232M shared.
PhysMem: 16G used (3947M wired), 40M unused.
VM: 1225G vsize, 627M framework vsize, 331473917(128) swapins, 335304932(0) swapouts.
Networks: packets: 15741002/13G in, 11739437/2992M out. Disks: 13747180/1392G read, 9813563/1406G written.

PID    COMMAND      %CPU TIME     #TH    #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS          %CPU_ME
96106  Java Updater 0.0  00:13.66 6      2    187   4888K  0B     7280K  96106 1     sleeping *0[1]           0.00000
95615  garcon       0.0  00:12.33 4      2    158   3692K  0B     3756K  95615 1     sleeping *0[2099]        0.00000
95608  com.apple.ap 0.0  01:10.69 4      2    370   14M    0B     30M    95608 1     sleeping *0[7611]        0.00000
94975  Google Chrom 0.0  09:23.28 27     2    275   77M+   0B     261M-  64743 64743 sleeping *0[1]           0.00000
94968  Google Chrom 0.0  04:06.68 18     2    142   18M    0B     93M    64743 64743 sleeping *0[1]           0.00000
94950  Google Chrom 0.0  00:47.81 17     2    130   10M    0B     49M    64743 64743 sleeping *0[1]           0.00000
94943  Google Chrom 0.0  00:24.97 17     2    139   10M    0B     57M    64743 64743 sleeping *0[1]           0.00000
92761  Google Chrom 0.0  08:49.28 20     1    155   29M    0B     58M    64743 64743 sleeping *0[1]           0.00000
92759  Google Chrom 0.0  77:21.83 19     2    157   69M    0B     128M   64743 64743 sleeping *0[1]           0.00000
92390  Google Chrom 0.1  09:35.03 18     2    141   16M    0B     95M    64743 64743 sleeping *0[1]           0.00000
92375  Google Chrom 0.2  10:06.90 18     2    140   13M    0B     75M    64743 64743 sleeping *0[1]           0.00000
91952  Microsoft Po 0.9  01:52:53 21/1   5    775   103M   60K    1504M  91952 1     running  *3[4174]        0.00000
91749  Microsoft AU 0.0  00:17.06 6      3    185   7404K  0B     5184K  91749 1     sleeping *0[2903]        0.00000
89134  syspolicyd   0.0  00:03.75 2      2    51    508K   0B     8900K  89134 1     sleeping  0[45]          0.00000
88314  DiskUnmountW 0.0  00:00.48 2      2    35    24K    0B     1148K  88314 1     sleeping  0[0]           0.00000
80927  com.apple.We 0.0  00:32.36 6      1    253   14M    0B     105M   80927 1     sleeping *0[2536]        0.00000
80926  com.apple.We 0.0  00:14.15 4      1    90    1032K  0B     12M    80926 1     sleeping  0[3077]        0.00000
80575  Google Chrom 0.0  16:24.06 19     2    156   53M    0B     177M   64743 64743 sleeping *0[1]           0.00000
78388  Google Chrom 0.5  68:17.96 23     2    177   84M    0B     158M   64743 64743 sleeping *0[1]           0.00000
64795  Google Chrom 0.5  02:48:06 24     2    182   116M   0B     212M   64743 64743 sleeping *0[1]           0.00000
64793  Google Chrom 0.0  04:54.37 19     2    146   81M    0B     168M   64743 64743 sleeping *0[1]           0.00000
64786  Google Chrom 0.1  11:39.68 19     2    161   247M   0B     218M   64743 64743 sleeping *0[1]           0.00000
64784  Google Chrom 9.3  26:10.87 20     2    185   679M   0B     2091M  64743 64743 sleeping *0[1]           0.00000
64782  Google Chrom 0.1  13:21.65 19     2    173   228M   0B     147M   64743 64743 sleeping *0[1]           0.00000
64781  Google Chrom 0.0  09:58.39 19     2    160   122M   0B     247M   64743 64743 sleeping *0[1]           0.00000
64780  Google Chrom 0.0  01:58.74 9      1    101   100M   0B     235M   64743 64743 sleeping *0[1]           0.00000
64779  Google Chrom 0.1  17:18.30 16     1    182   289M   0B     318M   64743 64743 sleeping *0[1]           0.00000
64776  Google Chrom 0.0  00:12.54 16     1    123   7868K  0B     16M    64743 64743 sleeping *0[1]           0.00000
64775  Google Chrom 0.0  00:12.67 16     1    123   7732K  0B     18M    64743 64743 sleeping *0[1]           0.00000
```

- Running program “top” on Mac
  - System has 418 processes, 4 of which are active
  - Identified by Process ID (PID)

`htop`

- Similar to `top`, but with some extra features

`lsof`

- Used to find open files and which processes are using them.

The Reality: Traditionally

- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

The Reality: In Modern systems

- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Concurrent Processes

- Each process is a logical control flow.
- Two processes run ***concurrently*** (are concurrent) if their flows overlap in time
  - Otherwise, they are ***sequential***
  - Examples (running on single core):
    - Concurrent: A & B, A & C
    - Sequential: B & C

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other

Context Switching

- Processes are managed by a shared chunk of memory resident OS code called the kernel
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a context switch

An Aside: Errors when using System Calls

- On error, Linux system-level functions typically return -1 and set global variable errno to indicate cause.
  - Hard and fast rule:
    - You must check the return status of every system-level function
    - Only exception is for the few functions that return void
  - Example:

```
if ((pid = fork()) < 0) {
Ifprintf(stderr, "fork error: %s\n", strerror(errno));
Iexit(0);
 }
```

Error Handling

- Can simplify somewhat using an error-reporting function:

```
void unix_error(char *msg) /* Unix-style error */
{
Ifprintf(stderr, "%s: %s\n", msg, strerror(errno));
Iexit(0);
}
```

```
if ((pid = fork()) < 0)
Iunix_error("fork error");
```

Error Handling

- Can simplify more by wrapping system calls with an error-handling function:

```
pid_t Fork(void)
{
 pid_t pid;
 if ((pid = fork()) < 0)
 unix_error("Fork error");
 return pid;
}
```

```
pid = Fork();
```

Things we want to do with processes

- Get process IDs

Things we want to do with processes

- Get process IDs
- Create and terminate processes

Things we want to do with processes

- Get process IDs
- Create and terminate processes
- Reap child processes

Things we want to do with processes

- Get process IDs
- Create and terminate processes
- Reap child processes
- Kill the Zombies

Things we want to do with processes

- Get process IDs
- Create and terminate processes
- Reap child processes
- Kill the Zombies
- Synchronizing with child processes

Getting Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process
- `pid_t getppid(void)`
  - Returns PID of parent process

Process States

From a programmer's perspective, we can think of a process as being in one of three states:

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execu;on is suspended and will not be scheduled until further notice (next lecture when we study signals)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- Process becomes terminated for one of three reasons:

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function
- `void exit(int status)`

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function
- `void exit(int status)`
  - Terminates with an exit status of `status`

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function
- `void exit(int status)`
  - Terminates with an exit status of `status`
  - Convention: normal return status is 0, nonzero on error

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function
- `void exit(int status)`
  - Terminates with an exit status of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function
- `void exit(int status)`
  - Terminates with an exit status of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine
- `exit` is called once but never returns.

Creating Processes

- Parent process creates a new running child process by calling `fork`

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

Creating Processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called once but returns twice

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
    childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
    parentprint
    exit(0);
}
I
```

- Call once, return twice

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
     childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
     parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
     childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
     parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child

fork example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
      childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
      parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space

`fork` example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
      childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
      parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
     childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
     parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files

`fork` example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork(); //line:ecf:forkreturn
    if (pid == 0) {  /* Child */
IIprintf("child : x=%d\n", ++x); //line:ecf:
     childprint
IIexit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x); //line:ecf:
     parentprint
    exit(0);
}
I
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`
  - Edges can be labeled with current value of variables

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - printf vertices can be labeled with output

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - printf vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - printf vertices can be labeled with output
  - Each graph begins with a vertex with no inedges
- Any topological sort of the graph corresponds to a feasible total ordering.

Using Process Graphs (maybe)

- A ***process graph*** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - `a -> b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - printf vertices can be labeled with output
  - Each graph begins with a vertex with no inedges
- Any topological sort of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

```c
int main() {
Ipid_t pid;
Iint x = 1;
Ipid = Fork();
Iif (pid == 0) {
II/* Child */
IIprintf("child : x=%d\n", ++x);
IIexit(0);
I}
I/* Parent */
Iprintf("parent: x=%d\n", --x);
Iexit(0);
}
I
```

Interpreting Process Graphs

- Original Graph:

Interpreting Process Graphs

- Original Graph:
- Relabeled Graph:

Interpreting Process Graphs

- Original Graph:
- Relabeled Graph:

Nested forks

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn't reap?

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a “zombie”
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid == 1)

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a “zombie”
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid == 1)
  - So, only need explicit reaping in long-running processes

Reaping Child Processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

- ps shows child process as "defunct" (i.e., a zombie)
  - Killing parent allows child to be reaped by init

Non-Terminating Child Example

- `ps` shows child process as “defunct” (i.e., a zombie)
  - Killing parent allows child to be reaped by init

`wait` : Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the pid of the child process that terminated
  - If `child_status != NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
    - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
    - Details…look them up if you're interested :)

Synchronizing with children

```
void fork9() {
Iint child_status;
Iif (fork() == 0) {
IIprintf("HC: hello from child\n");
IIexit(0);
I} else {
IIprintf("HP: hello from parent\n");
IIwait(&child_status);
IIprintf("CT: child has terminated\n");
I}
Iprintf("Bye\n");
}
I
```

Feasible Output:

- HC
- HP
- CT
- Bye

Infeasible Output:

- HP
- CT
- Bye
- HC

Another wait example

```
void fork10() {
Ipid_t pid[N];
Iint i, child_status;
Ifor (i = 0; i < N; i++)
IIif ((pid[i] = fork()) == 0) {
IIexit(100+i); /* Child */
I}
Ifor (i = 0; i < N; i++) { /* Parent */
IIpid_t wpid = wait(&child_status);
IIif (WIFEXITED(child_status))
IIIprintf("Child %d terminated with exit status %d\n",
IIIIIwpid, WEXITSTATUS(child_status));
IIelse
IIIprintf("Child %d terminate abnormally\n", wpid);
I}
}
I
```

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

`waitpid`: Waiting for a specific process id

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates

```
void fork11() {
Ipid_t pid[N];
Iint i;
Iint child_status;
Ifor (i = 0; i < N; i++)
IIif ((pid[i] = fork()) == 0)
IIIexit(100+i); /* Child */
IIfor (i = N-1; i >= 0; i--) {
IIIpid_t wpid = waitpid(pid[i], &child_status, 0);
IIIif (WIFEXITED(child_status))
IIIIprintf("Child %d terminated with exit status %d\n",
IIIIIIwpid, WEXITSTATUS(child_status));
IIIelse
IIIIprintf("Child %d terminate abnormally\n", wpid);
II}
I
```

Processes: Summary

- Exceptions
- Processes
  - At any given time, a system has multiple processes running
  - Only one can run at a time on a single processor
  - Each process *appears* to have complete control over the processor, memory, etc.
- Spawning processes
  - Call `fork`
  - Call once, returns twice
- Process completion
  - Call `exit`
  - Call once, returns 0 times
- Reaping and waiting for processes
  - Call `wait` or `waitpid`

Threads vs Process

- A thread is a single sequence stream within in a process.
  - Threads are sometimes called *"lightweight processes"*

Threads vs Process

- A thread is a single sequence stream within in a process.
  - Threads are sometimes called *“lightweight processes”*
- Both are considered ***independent sequences of execution***

Threads vs Process

- A thread is a single sequence stream within in a process.
  - Threads are sometimes called *"lightweight processes"*
- Both are considered ***independent sequences of execution***
- Threads run in a shared memory space

Threads vs Process

- A thread is a single sequence stream within in a process.
  - Threads are sometimes called *"lightweight processes"*
- Both are considered ***independent sequences of execution***
- Threads run in a shared memory space
- All processes start out with a single thread, and can start a new thread

1

[1] https://randu.org/tutorials/threads/

Why have threads?

- When the process has a lot to do, threads can run simultaneously, potentially speeding up computation
- Threads start faster than processes
- It's easier & more efficient for threads to communicate and share memory