Lecture 3: Processes

601.418/618 Operating Systems

David Hovemeyer

January 29, 2024

Agenda

- Processes: concepts
- Processes: kernel perspective
- Processes: user (program) perspective

Acknowledgments: These slides are based on Prof. Ryan Huang's Fall 2022 slides, which in turn are based on Prof. David Mazières's OS lecture notes. They also include some content developed by Prof. Phillipp Koehn for CSF.

Recap: architectural support

Architectural features to allow a robust process abstraction:

- Execution modes (kernel mode vs. user mode)
- Event handling (asynchronous vs. synchronous, hardware-generated vs. software-generated)
- Synchronization (disabling/enabling interrupts, atomic machine instructions)

Today: focus on how these hardware features are used to create the process abstraction.

Processes: concepts

Processes

Questions to think about:

- What data structures are needed to represent a process
- How to processes utilize the CPU?
  - What is the OS kernel's role in allowing processes to share the CPU?
- What is the lifecycle of a process?
  - As it runs, what are the important *states* it moves through?

Process abstraction

In a traditional OS kernel, processes are the basic unit of scheduling.

- I.e., when the OS kernel makes a scheduling decision, it chooses a runnable process

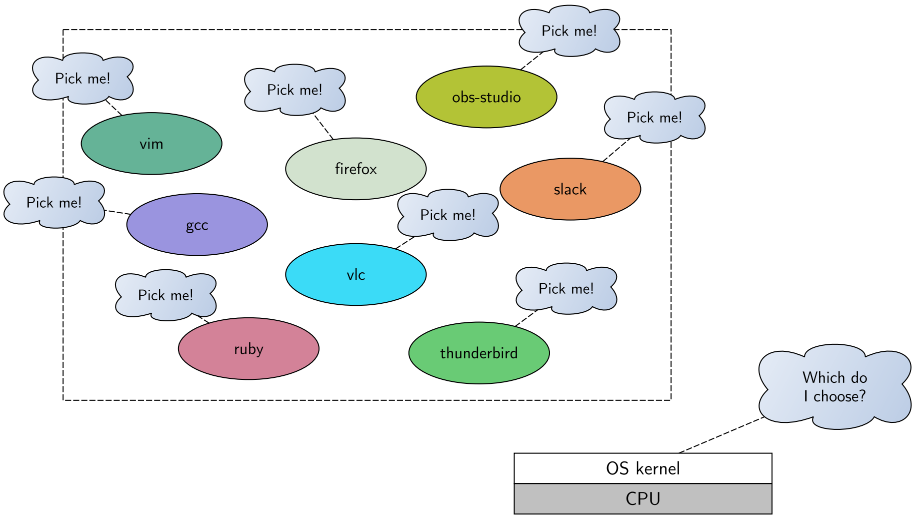
Traditionally, the term “process” is synonymous with “job” or “task”.

- Looking ahead: *threads* execute within a process, and this will change our view of scheduling

A *process* is a *running program*.

- A “program” is a static entity: instructions and data that have the potential to become a process.
- A “process” is a dynamic entity: the instructions of a program are actually being executed.

Process scheduling in a nutshell



The early days

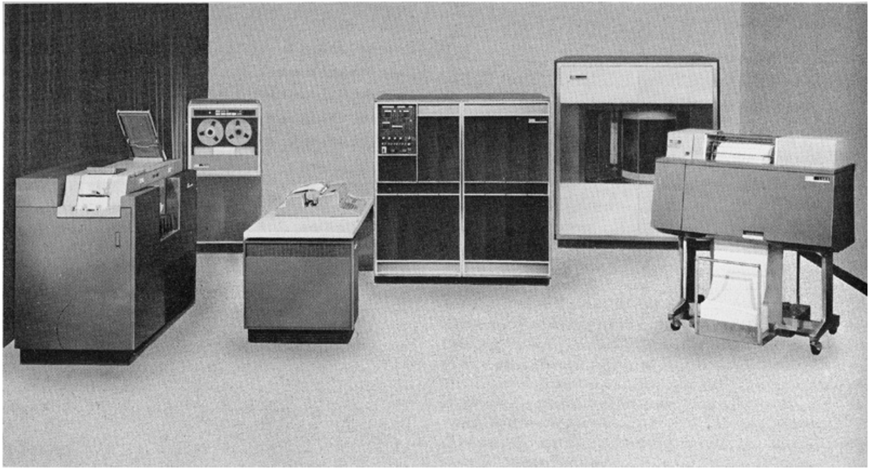
In the early days of computing, *batch processing* was the norm.

Programs would be run one at a time:

1. Reset computer
2. Load program from media (punched cards, paper tape, magnetic tape, etc.)
3. Execute program (might involve reading data from media)
4. Wait for program to finish and store or print its output

A human *operator* is required to supervise and control the machine.

IBM 1401



Source: http://www.columbia.edu/cu/computinghistory/1401.html

Drawbacks of batch processing

Users of the system typically have to wait hours or days before their job is run and they can see the program output.

- Imagine the feeling if you found that your program had a bug!

The CPU spends much of its time idle (between jobs and when waiting for data to be read from or written to media.)

Modern OS

daveho@lobsang

File Edit View Search Terminal Help

```
top - 06:08:47 up 27 min,  1 user,  load average: 0.53, 0.37, 0.35
Tasks: 312 total,   5 running, 306 sleeping,   0 stopped,   1 zombie
%Cpu(s): 47.2 us,  3.7 sy,  0.0 ni, 49.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  32034.8 total,  27572.8 free,   2384.7 used,   2077.4 buff/cache
MiB Swap:  31250.0 total,  31250.0 free,      0.0 used.  29145.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5726 daveho    20   0  182588 152924  19224 R  24.3   0.5   0:00.73 cc1plus
 5729 daveho    20   0  162776 132624  17988 R  18.6   0.4   0:00.56 cc1plus
 5735 daveho    20   0  108204  79844  16620 R  10.3   0.2   0:00.31 cc1plus
 5738 daveho    20   0   75312  45176  14100 R   4.3   0.1   0:00.13 cc1plus
 1196 root      20   0 1263052 206304 158484 S   2.7   0.6   0:29.46 Xorg
 2025 daveho    20   0 1604344  68992  46116 S   2.0   0.2   0:06.47 mate-terminal
 2207 daveho    20   0 2352920 172648  78036 S   1.3   0.5   0:21.61 quodlibet
 1666 daveho    20   0  459764  41660  29560 S   0.7   0.1   0:04.92 marco
   14 root      20   0       0      0      0 I   0.3   0.0   0:00.78 rcu_sched
 1386 daveho     9 -11 1238620  31484  21644 S   0.3   0.1   0:03.04 pulseaudio
 1701 daveho    20   0  663532  83640  74428 S   0.3   0.3   0:02.41 wnck-applet
 2296 daveho    20   0 3508696 541108 231016 S   0.3   1.6   0:37.72 firefox-bin
 4408 root      20   0       0      0      0 I   0.3   0.0   0:00.15 kworker/u16:1-events_fr+
 4774 daveho    20   0   13348   4140   3284 R   0.3   0.0   0:01.87 top
 5547 daveho    20   0    8860   2904   2452 S   0.3   0.0   0:00.01 make
    1 root      20   0  166548  12024   8428 S   0.0   0.0   0:00.71 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
```

Modern OS

On a modern OS, we expect that multiple programs can run "at the same time."

- If a process is ready to execute instructions, and a CPU is available, we want the process to execute

If the system has multiple users, their processes can all run "at the same time."

- This idea is called "time sharing"

Batch processing still has its uses

Note that there are specialized situations where batch processing still make sense, especially in cases where we *want* a single program to have full access to the CPU(s).

Scheduling CPU-intensive scientific computations on high-performance computing clusters is a good example.

- When such a program runs, we don't want it to compete with other processes for access to the CPU(s)
- So, we run them one at a time

Multiprogramming

Multiprogramming: run multiple processes "at the same time"

- A.k.a. *multitasking*.

Many processes exist simultaneously.

When the OS kernel makes a scheduling decision, it chooses a process to switch to.

When a process is forced to wait (e.g., for an I/O operation to complete), the OS kernel can switch to a process that is ready to execute.

System resources (CPU and I/O devices) can be more fully utilized than in batch processing.

- Because there are lots of processes to keep them busy!

Throughput and latency

Throughput: work finished per unit of time.

Latency: interval between when a task starts and when it finishes.

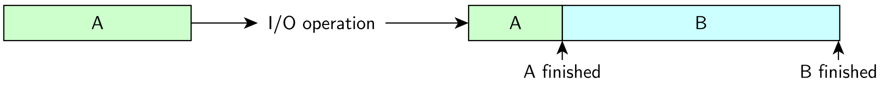
Multiprogramming can be advantageous for both throughput and latency.

- Although, there is no free lunch: "perfect" utilization of system resources would require knowledge of the future

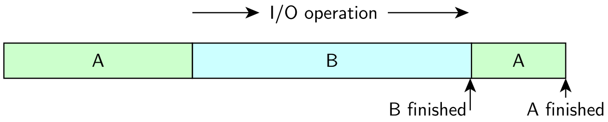
Throughput example

Having multiple processes available for scheduling allows the OS kernel to avoid leaving the CPU idle during I/O operations.

Leaving CPU idle during I/O:



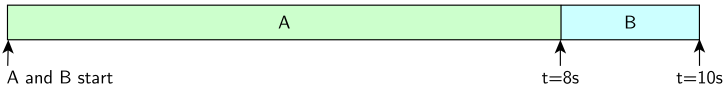
Overlapping process execution and I/O:



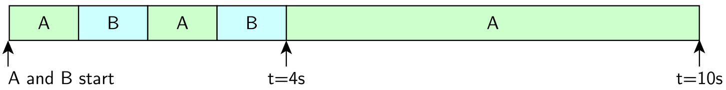
Latency example

Processes A and B start at about the same time. A will require 8 seconds and process B will require 2 seconds.

Option 1: run them one at a time. Average latency is 9s.



Option 2: switch between them. Average latency is 7s.



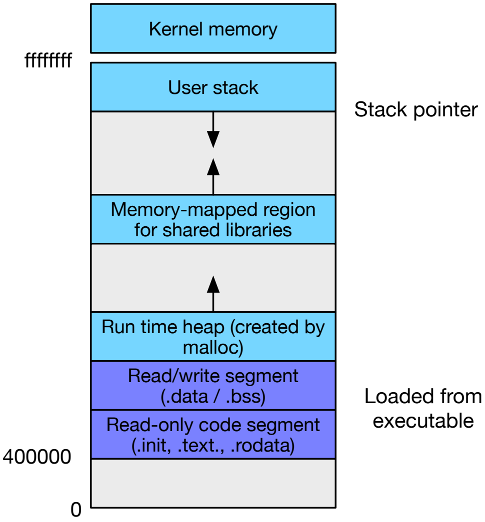
Processes: kernel perspective

Kernel's view of a process

State associated with a process

- Its *address space* (the memory it's using and the contents of that memory)
  - Code, data
- Its stack
  - This is just memory in the address space pointed to by the stack pointer
- Program counter
  - Which instruction will execute next
- Values of CPU registers
  - If the process is currently suspended (e.g., waiting for I/O to complete), these are needed when the OS kernel eventually resumes execution of the process
- Resources used (files, terminal, network connections, etc.)

Process address space



Process's view of itself

To code executing with in a process:

- It only sees memory in its own address space
- As far as it knows, it has its own CPU
  - The proceess doesn't know or care that the OS kernel is actually switching processes
- It has its own set of open files

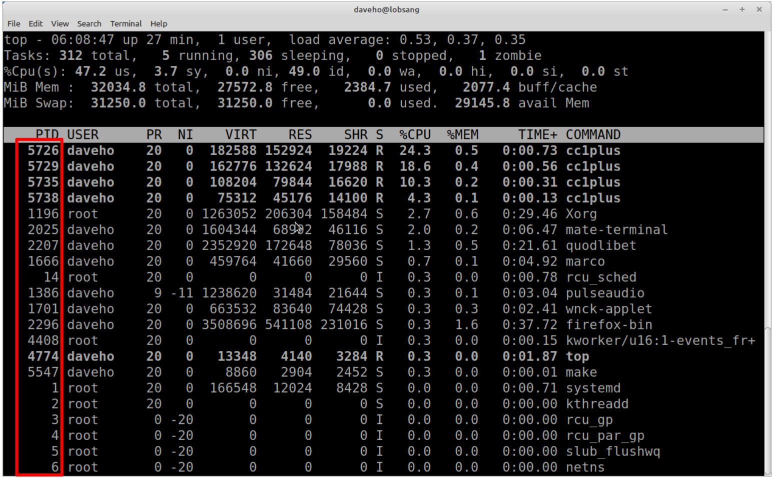
Virtual memory: an address within a process has *no meaning* to other processes.

- Address `0xdfba260` in process A is a different memory location that address `0xdfba260` in process B

Each process has its own isolated collection of resources. It's as though each process has its own virtual computer system.

Identifying processes

The OS kernel will maintain a unique *identifier* naming the process. On Unix/Linux, each process has a *process id* ("PID".)



Interprocess communication

Processes are isolated from each other (by default, their resources are private.)

Sometimes we would like one process to be able to interact with another. Many mechanisms exist for this purpose:

- files (one process writes, another reads)
- shared memory (memory regions in two processes map to the same physical memory)
- communication channels (pipes, sockets)
- signals (one process sends a signal to another)

Some of these mechanisms require some form of synchronization to use correctly. (E.g., shared memory, files.)

Some mechanisms are inherently sequential (pipes, sockets).

Process Control Block (PCB)

In the kernel, each process is represented by an instance of a data type known as the "Process Control Block" (PCB).

On Linux, this is the `struct task_struct` data type.

Linux task_struct data type

```
struct task_struct {
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info                  thread_info;
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long                        state;
    void                                *stack;
    refcount_t                          usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int                        flags;
    unsigned int                        ptrace;
#ifdef CONFIG_SMP
    struct llist_node                   wake_entry;
    int                                 on_cpu;
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* Current CPU: */
    unsigned int                        cpu;
#endif
```

Linux task_struct data type

```
    unsigned int                     wakee_flips;
    unsigned long                    wakee_flip_decay_ts;
    struct task_struct              *last_wakee;
    /*
     * recent_used_cpu is initially set as the last CPU used by a task
     * that wakes affine another task. Waker/wakee relationships can
     * push tasks around a CPU where each wakeup moves to the next one.
     * Tracking a recently used CPU allows a quick search for a recently
     * used CPU that may be idle.
     */
    int                             recent_used_cpu;
    int                             wake_cpu;
#endif
    int                             on_rq;
    int                             prio;
    int                             static_prio;
    int                             normal_prio;
    unsigned int                    rt_priority;
    const struct sched_class       *sched_class;

    /* ...etc, lots more fields... */
```

Some interesting things in Linux task_struct

```
struct mm_struct                *mm;
```

Data structure representing the virtual address space (which memory regions exist, what is mapped in each region, etc.)

```
struct files_struct             *files;
```

Table of open files.

```
volatile long                   state;
```

Process state.

Process states

At any given point, a process is in one of the following states:

Running The process is currently executing instructions (i.e., it's scheduled on a CPU.)

Ready The process is ready to run, but isn't currently executing on a CPU. When the OS kernel makes a scheduling decision, it chooses a Ready process.)

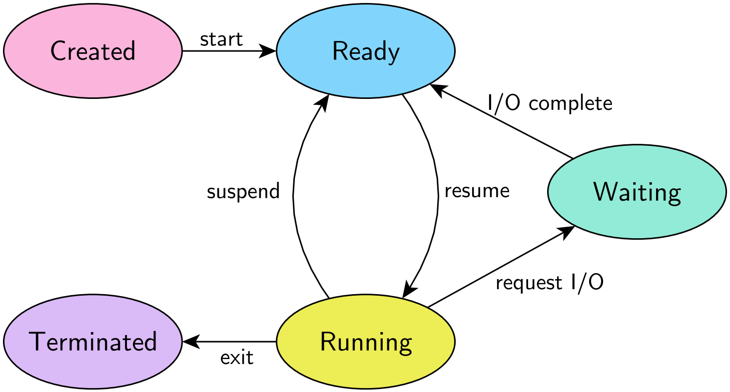
Waiting The process is suspended while it waits for an event to occur (usually the completion of an I/O request.)

Viewing processes and their states

The Unix/Linux `ps` and `top` programs list processes, and have a `STAT` or `S` column to indicate the state of each process.

```
$ ps --sort=-pcpu auxw
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
daveho     13036 98.0  0.5 219628 190116 pts/3   R+   10:55   0:00 /usr/lib/gcc/x86_64-linux
root        1196  1.2  0.6 1303036 224240 tty7   Ssl+ 05:41   3:52 /usr/lib/xorg/Xorg -core
daveho      2296  1.2  1.6 3617808 534216 ?      Sl   05:43   3:51 /usr/lib/firefox/firefox
daveho     13025  1.0  0.0  10192  4092 pts/3    S+   10:55   0:00 make -j 4
daveho      7818  0.8  2.6 13001900 855908 ?     Sl   06:50   2:05 java -jar /home/daveho/So
daveho     12243  0.6  0.7 2574280 232020 ?      SLl  10:49   0:02 gimp-2.10 /home/daveho/gi
daveho      2207  0.5  0.5 2404980 176868 ?      SLl  05:42   1:50 /usr/bin/python3 /usr/bin
daveho      2554  0.4  0.4 2688348 158436 ?      Sl   05:43   1:30 /usr/lib/firefox/firefox-
daveho      1505  0.3  0.0 387636  9656 ?        Ssl  05:41   1:01 /usr/bin/ibus-daemon --da
daveho     12827  0.3  0.0 117844  7928 ?        Ssl  10:53   0:00 /usr/bin/speech-dispatche
daveho      2140  0.2  0.1 485068 52432 ?        Sl   05:42   0:45 gvim -f
daveho      2285  0.2  0.1 408724 52240 ?        Sl   05:42   0:39 mintreport-tray
daveho     11885  0.2  0.4 2505204 157196 ?      Sl   10:39   0:02 /usr/lib/firefox/firefox-
daveho      1593  0.1  0.0 163608  7260 ?        Sl   05:41   0:18 /usr/libexec/ibus-engine-
```

Process lifecycle



Tracking process states

The OS kernel will often need to find a process that is in a particular state. For example, find a process that is in the Ready state when making a scheduling decision.

Could search list of all processes

- Inefficient, especially if there is a large number of processes

Better idea: different lists for processes in different states

- Ready queue: list of all processes in the Ready state

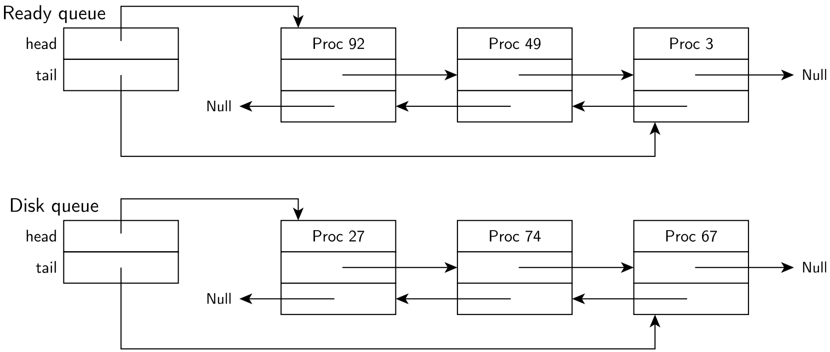
For processes in Waiting state, have a separate queue per event type

- E.g., a queue of processes waiting on disk I/O

Process PCBs are moved from queue to queue when the state changes.

Running state: don't really need a queue, since (at least on a single-processor system) there can be only one process in this state.

Process queues



Scheduling

When a scheduling decision is made, which Ready process should the kernel choose?

Goals:

- fairly allocate CPU time to processes
- avoid any expensive operations (e.g., don't search the entire ready queue)

Pintos default scheduler: FIFO

- Running $\rightarrow$ Ready (suspend process): put process at tail of ready queue
- Ready $\rightarrow$ Running (resume process): remove process at head of ready queue

We'll have a lot more to discuss regarding scheduling in the near future.

Note that if there is a special "idle" process that never waits, then there is always a process available to schedule.

- The idle process should only be scheduled if there is not another process in the Ready state

Preemption

Preemption means that every time the OS kernel regains control of the CPU, it could make a scheduling decision.

The *interval timer* device ensures that the OS kernel has the opportunity to make scheduling decisions at regular intervals.

Any exception (hardware interrupt, fault, system call, etc.) is an opportunity for the kernel to make a scheduling decision.

A process (while running in kernel mode) could also voluntarily offer to find a different process to run (`sched_yield()`.)

Context switch

Switching from one process to another is called a *context switch*.

General idea:

- save context of process being suspended
- restore context of process being resumed

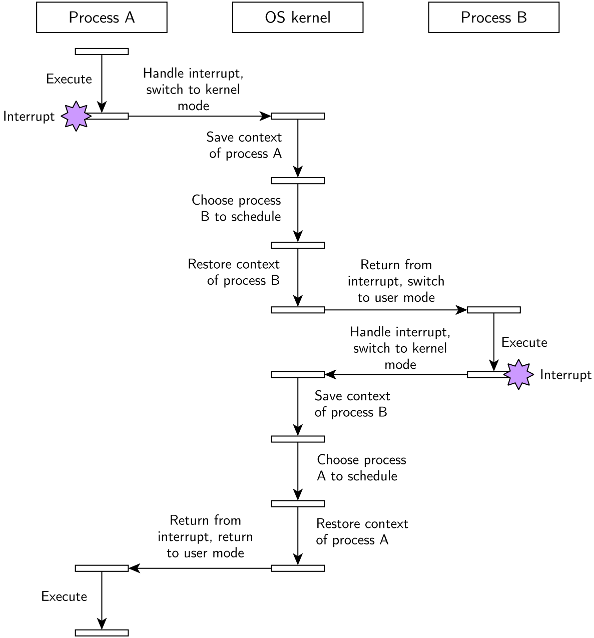
“Context” primarily means the values of CPU registers, but also includes

- the process’s virtual address space
- floating point registers (and other special registers such as the condition code register)

Context is often saved in the Process Control Block.

- It could also be saved on the process’s *kernel stack*

Context switch



Context switch cost

A context switch has a non-zero cost.

Changing address spaces will generally require the TLB to be flushed. (We don't want "stale" mappings of the old process's virtual pages.)

Changing address spaces will also usually incur a significant number of cache misses (since process's generally don't share memory, so the switched-to process's data will need to be loaded into the cache.)

Saving and restoring floating point registers can be expensive.

Trade-off: making scheduling decisions more frequently means that processes wait less to use the CPU, but means more overhead due to context switches.

Processes: user (program) perspective

Creating a process

Important system call: allow the creation of a *child* process

Each process has a parent

- Child process will run with user identity of parent, so has same privileges, can access same resources
- Point to ponder: how is the first process created?
  - On Unix/Linux, process ID 1 is the “init” process
  - Is ultimately responsible for creating all other processes

Creating a process: Windows

On Windows, the `CreateProcess` system call creates a child process:

```
BOOL CreateProcess(char *prog, char *args, /* various optional arguments */);
```

1. Create PCB for new process
2. Create new address space
3. Arrange for code and data of specified executable to be mapped into the new address space
4. Copy `args` (command line arguments) into the new address space
5. Prepare initial context
   - Make it look like the process was interrupted just before executing the first instruction at the program's entry point
6. Put PCB on the ready queue

Creating a process: Unix/Linux

In Unix/Linux, the `fork()` system call creates a child process:

```
pid_t fork(void);
```

1. Create PCB for new process
2. Create a *copy* of the parent's address space
3. Duplicate parent process's open files
4. Copy parent process's context
5. Put PCB on the ready queue

In the parent process, `fork()` returns the child's process ID.

In the child process, `fork()` returns 0.

fork() example program

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s is %d\n", name, getpid());
    return 0;
  } else {
    printf("My child is %d\n", child_pid);
    return 0;
  }
}
```

Running the example program

```
$ make fork_example
cc     fork_example.c   -o fork_example
$ ./fork_example
My child is 6784
Child of ./fork_example is 6784
```

Returning a different return value in each process

Because a child process created by `fork()` runs in an address space identical to its parent, the address of the instruction `fork()` returns to is the same in each process.

The only difference in the context for returning to parent vs. returning to child is what value is stored in the CPU register storing the return value (`%eax` for 32-bit x86):

- In the parent process, the return value register stores the child's process ID
- In the child process, the return value register stores 0

The child runs asynchronously

Once the child process is added to the ready queue, it competes for CPU time the same as any other process. So, the execution of the child process is not synchronized with the execution of the parent process.

For the example program, the outputs

```
My child is 6784
Child of ./fork_example is 6784
```

and

```
Child of ./fork_example is 6784
My child is 6784
```

are both possible.

Starting a different program

Having the child run the same program as the parent is sometimes useful.

However, most of the time the goal is for the child process to run a different program.

```
int execv(char *prog, char *argv[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

1. Create new address space (replacing the current one)
2. Map code and data of executable program into the new address space
3. Set up context to begin execution at the new program's entry point
4. Place PCB back in the ready queue

Note: these functions only return if an error occurs and the new program can't be executed.

Also note: the Pintos `exec` is like `fork` followed by Unix/Linux `exec`.

When `fork()` is useful

Even though the common case of following `fork()` with `exec()` is useful, there are important situations where `fork()` alone is quite useful.

The child process

1. Inherits resources from the parent (including open files, network connections, etc.)
2. The child can make any adjustments to the state it inherited from the parent that it needs to
   - No special arguments for `fork()` are needed to do things like set the child process's standard input, standard output, etc.
   - Contrast: Windows `CreateProcess()`, lots of complicated optional arguments

Network server

```c
for (;;) {
  int fd = accept(ssock, NULL, NULL);
  pid_t child = fork();
  if (child == 0) {
    chat_with_client(fd);
    exit(0);
  }
  close(fd);
}
```

Shell program

`minish.c` (excerpt):

```
pid_t pid; char **av;
void doexec () {
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

/* ... main loop: */
for (;;) {
  parse_next_line_of_input (&av, stdin);
  switch (pid = fork ()) {
  case -1:
    perror ("fork"); break;
  case 0:
    doexec ();
  default:
    waitpid (pid, NULL, 0); break;
  }
}
```

Shell program

```
bash$ make minish
cc     minish.c   -o minish
bash$ ./minish
$ /bin/echo Hello
Hello
$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Shell program with I/O redirection

redirsh.c (excerpt):

```
void doexec (void) {
  int fd;
  if (infile) {/* non-NULL for "command < infile" */
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0); // <---- change standard input in child
      close (fd);
    }
  }
  /*...do same for outfile→fd 1, errfile→fd 2...*/
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```

Shell program with I/O redirection

```
bash$ make redirsh
cc     redirsh.c   -o redirsh
bash$ ./redirsh
$ ls > files
$ grep sh < files
minish
minish.c
redirsh
redirsh.c
```

Ending a process

Windows:

```
void ExitProcess(UINT uExitCode);
```

Linux/Unix:

```
void _exit(int status);
```

1. Terminate all threads (much more to say about this next time)
2. Close all open files
3. Tear down address space
   - Release all memory used by process
4. Notify parent process of child's exit status
5. (Eventually) destroy PCB

The OS kernel is responsible for deallocating all resources used by a process.

Wait for child to finish

Windows:

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

Linux/Unix:

```
pid_t wait(int *wstatus);
```

Parent process is suspended until the child process exits.

Zombie process: child process that has exited, but the parent hasn't attempted to `wait` for the child. The OS kernel is obligated to preserve a record of the child process (and not reuse its PID) until the parent has waited for it.

Next time

Threads and thread scheduling!