Operating System Concepts

Parallel Programming

2024-11-18

Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

Programming with OpenMP

- How many threads run in this example?
    1. Twelve (nproc output)
    2. Four
    3. Three
    4. Two
    5. One

```c
int main(void) {
    omp_set_num_threads(3);

    #pragma omp parallel num_threads(4) if(1)
    printf("Hello world from "
           "thread %02d/%02d.\n",
           omp_get_thread_num(),
           omp_get_num_threads());

    return 0;
}
```

- What happens in this example?
  1. The same as with `parallel for`
  2. Compiler exits with an error
  3. Both threads calculate the whole loop
  4. Undefined behavior due to race condition

```c
int main(void) {
    int i;

    omp_set_num_threads(2);

    #pragma omp parallel
    for (i = 0; i < 10; i++) {
        printf("i=%d, id=%d\n",
            i, omp_get_thread_num()
        );
    }

    return 0;
}
```

- What is the fastest synchronization construct for incrementing a variable?
  1. `critical`
  2. `atomic`
  3. `reduction`
  4. `omp_lock_t`

Outline

Motivation

- Parallel programming often requires low-level knowledge
  - Hardware architecture (NUMA), scheduling, affinity etc.
- The operating system is involved in many decisions
  - Having a basic understanding of operating system concepts is necessary
- We will take a look at some of those concepts
  - Applications, processes and threads
  - Privileges, kernel/user mode and thread-safety
  - Inter-process communication (IPC) via shared memory

Applications, Processes and Threads

- Application
  - Executable binary, usually compiled from source code
  - Applications can be started as processes
- Process
  - Operating system object to manage application instances
  - Isolated address spaces due to security reasons
  - Files, allocated memory etc. are managed per-process
- Thread
  - Lightweight process or sub-process
  - Shared address space for all threads within a process

Outline

Operating System

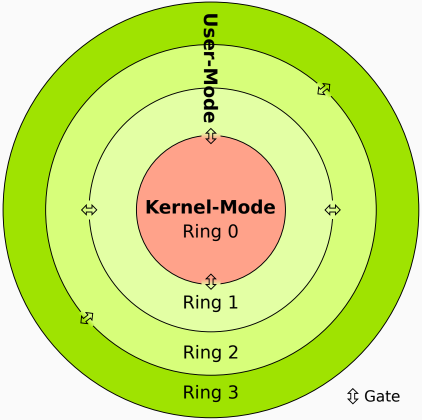
- Operating system manages applications, processes and threads
  - Provides functionality to start new processes etc.
- Uses similar concepts internally
  - Operating system is often used synonymously with kernel
  - Kernel is an application that runs directly on top of the hardware
  - Uses threads for performance improvements and separation of concerns
- Operating system is responsible for much more
  - File system, I/O, network, user/group management etc.

Operating System…

- Operating system schedules tasks
  - Tasks can be processes, threads, kernel threads etc.
  - Available cores are used to execute tasks
- Achievable performance depends on scheduling policy
  - Cooperating threads should be scheduled at the same time
  - There are usually more tasks to be scheduled than cores available
- Processes and threads can be mapped differently to tasks
  - Most operating systems use a 1:1 mapping

Privileges

- Privileges are separated into rings
  - Processors usually support four rings
  - Possible to transition between rings
  - OSs often only use rings 0 and 3
- Kernel mode allows full hardware access
  - Privileged operations and physical memory
  - Also called supervisor mode
- User mode is restricted
  - Reduced privileges, virtual memory
- Newer processors have a ring -1
  - Used for hypervisor mode



[Sven, 2006]

Privileges…

- Transitions between rings are called mode switches
  - Can be caused by system calls or interrupts
- Supervisor mode allows processes full access
  - Physical address space, memory management, peripherals etc.
  - Allowed for the kernel but not for user applications
- User applications have to perform system calls into kernel mode
  - Kernel can then use supervisor mode to make privileged changes
  - Kernel returns execution to user space afterwards

Privileges…

- System calls can be quite expensive
  - Sometimes more than 1,000 processor cycles
- Earlier attempts at putting performance-critical software into the kernel
  - For instance, a web server for reduced access latency
  - Problematic from a security point of view due to privileges
- Linux injects vDSO (virtual dynamic shared object) into processes
  - Allows avoiding system calls in some cases
  - `malloc` does not perform system calls for each allocation
- Newer approaches use kernel bypass to reduce overhead
  - For instance, applications talk directly to the network card

Privileges…

- x86 uses privilege levels for instructions
  - From level 0 to 3, with 0 being the most privileged
  - Trying to execute a higher privileged instruction triggers a general protection fault
- Anatomy of a system call
  - User process sets up registers and memory, triggers system call
  - Software interrupt or special instruction causes switch to kernel mode
  - Kernel stores process state and checks user space request
  - Either context switch to different process or mode switch back to process
  - Usually handled by wrappers in the standard library (`libc`)

Thread-Safety

- Earlier operating systems and libraries were not thread-safe
  - Thread-safe code is a bit more complicated and has more overhead
- Example: Functions like `strerror` from the standard library
  - "… returns a pointer to a string that describes the error code …" [Linux man-pages project, 2021]
  - "This string must not be modified by the application, but may be modified by a subsequent call to `strerror()` …" [Linux man-pages project, 2021]
  - Not thread-safe since parallel invocations might modify the string
- Thread-safe versions of these functions have an _r suffix
  - Stands for reentrant, which means that a function can be safely used concurrently

Thread-Safety...

- Strictly speaking, reentrancy is different from thread-safety
    - Thread-safety means that multiple threads can call a function at the same time
    - Reentrancy is mainly used in the context of signal handling and interrupts
        - It is therefore also sometimes called signal-safety
- Functions can be interrupted by an interrupt
    - The interrupt handler can execute functions
    - If the interrupted function is called directly or indirectly, it is "re-entered"

- increment_count is thread-safe
  - Multiple threads can call it at the same time
  - There are no race conditions
  - Incrementing count is serialized

```
int increment_count(void) {
    int result;
    omp_set_lock(lock);
    result = count++;
    omp_unset_lock(lock);
    return result;
}
int main(void) {
    omp_init_lock(lock);
    #pragma omp parallel
    increment_count();
    printf("count=%d\n", count);
    omp_destroy_lock(lock);
    return 0;
}
```

- increment_count is thread-safe
    - Multiple threads can call it at the same time
    - There are no race conditions
    - Incrementing count is serialized
- Quiz: Is it also reentrant?

```c
int increment_count(void) {
    int result;
    omp_set_lock(lock);
    result = count++;
    omp_unset_lock(lock);
    return result;
}
int main(void) {
    omp_init_lock(lock);
    #pragma omp parallel
    increment_count();
    printf("count=%d\n", count);
    omp_destroy_lock(lock);
    return 0;
}
```

Thread-Safety…

- increment_count is thread-safe
  - Multiple threads can call it at the same time
  - There are no race conditions
  - Incrementing count is serialized
- Quiz: Is it also reentrant?
- It is not reentrant, though
  1. increment_count is called by application
  2. lock is set by omp_set_lock
  3. Function is interrupted
  4. Interrupt handler calls increment_count
  5. omp_set_lock will cause a deadlock

```c
int increment_count(void) {
    int result;
    omp_set_lock(lock);
    result = count++;
    omp_unset_lock(lock);
    return result;
}
int main(void) {
    omp_init_lock(lock);
    #pragma omp parallel
    increment_count();
    printf("count=%d\n", count);
    omp_destroy_lock(lock);
    return 0;
}
```

- Function is thread-safe and reentrant
  - atomic_fetch_add uses atomic instruction
- Can be interrupted and reentered at any time
  - There is no possibility for a deadlock
- Not all reentrant functions are thread-safe
  - Still often used synonymously (see POSIX)

```c
int increment_count(void) {
    int result;
    result = atomic_fetch_add(
        &count, 1);
    return result;
}

int main(void) {
    #pragma omp parallel
    increment_count();
    printf("count=%d\n", count);
    return 0;
}
```

- Thread-safety and reentrancy are also important for the operating system
- No problems if the operating system executes different applications
    - All cores are in user mode, no possibility for conflicts
- Multiple applications could switch to kernel mode
    - For instance, processes want to do I/O or communicate
    - System calls will switch to kernel mode and access the same OS region in parallel
    - Potential for conflicts within the kernel due to shared buffers etc.

Symmetric Multiprocessing

- Parallel systems typically use symmetric multiprocessing (SMP)
  - All processors and cores are treated equally by the operating system
- Applications can run on all processors in SMP systems
  - Processors can access the same code and data and enter the OS at the same time
  - It is necessary to have appropriate locks to avoid race conditions and deadlocks
- There is also asymmetric multiprocessing
  - For instance, one processor executes an application while the other runs the OS

Outline

Concurrency

- Operating system runs concurrently
  - Definition: Different parts can be executed out-of-order or in partial order
    - "Two distinct events a and b are said to be concurrent if $a \nrightarrow b$ and $b \nrightarrow a$." [Lamport, 1978]
    - "Two events are concurrent if neither can causally affect the other." [Lamport, 1978]
  - Enables parallelism, since concurrent parts can be executed in parallel
- Synchronization is necessary to avoid race conditions
  - Can be achieved using different means, explicitly and implicitly
  - Locks are most common but there are also barriers etc.
  - Lockless algorithms promise better performance than other approaches

Concurrency...

- Naive approach: Giant lock
    - Used in the beginning of parallel operating systems
    - Implies that only one core can enter kernel mode at a time
    - Massive performance bottleneck, see Python's global interpreter lock (GIL)
- Other extreme: Many fine-granular locks
    - Increases concurrency but also overhead due to locking
    - Harder to implement correctly than giant lock
- Goal: Find right granularity in between these two extremes

Concurrency…

- Achieving thread-safety took years for Linux
  - First versions in 1999 with fine-grained locks for signal handling, interrupts and I/O
- Improved support in 2001 (version 2.4)
  - “All major Linux subsystems are fully threaded” [Tumenbayar, 2002]
  - Including networking, file system, virtual memory, I/O, caches, scheduling etc.
- Optimizations are still routinely performed
  - For instance, allowing systems to scale with more cores
  - File system and memory accesses have to deal with more processes

- Locks implement mutual exclusion
    - That is, only one task can enter the critical region
    - A mutual exclusion lock is also called mutex
- Spinlocks are a way to implement locks
    - Lock regions using a shared variable
    - Lock availability is checked by testing in a loop (spinning)
    - Only makes sense for locks that are only held for a short time
    - We will see an example of a spinlock later

Synchronization…

- Semaphors are data structures for synchronization
  - Critical regions can be implemented using them
- More generic than a lock, which can only be set or unset
  - Semaphors usually implement counting
  - `V` to increment, `P` to decrement
- The semaphor's value is the number of free resources
  - `wait` (`P`): Wait for a free resource, decrement value by 1 and sleep (without consuming CPU time) if new value is negative
  - `signal` (`V`): Signal free resource availability, increment value by 1 and wake up task if old value was negative

- Lockless algorithms promise high performance
  - Most often achieved using atomic operations
  - Typically some standard atomic operations are provided
    - Store, load and exchange
    - Compare-and-exchange and test-and-set
    - Fetch-and-{add,sub,and,or,xor}
- Requires hardware support
  - Separate instructions provided by the processor
  - C11 allows checking with `atomic_is_lock_free`

Outline

Processes

- Reminder: Source code is compiled to an application
  - Application can be started multiple times
  - Each running instance of an application is called a process
- Processes are assigned a unique process ID (PID)
  - They also have a parent process, a thread group and more
- The PID is important when using system-level tools etc.
  - Killing processes using `kill` requires the PID
  - Log entries typically contain the PID for correlation

- The process's PID is returned by getpid
  - getppid returns the parent's PID
- Parent PID depends on how process is started
  - Typically the shell's PID
- The amount of PIDs is limited
  - Limit can be read from /proc/sys/kernel/pid_max
  - On current systems it is 4,194,304 (that is, 2^{22})
  - Used to be 32,768 (that is, 2^{15})
  - PIDs wrap around after reaching the maximum

```c
int main(void) {
    pid_t pid, ppid;

    pid = getpid();
    ppid = getppid();

    printf("pid=%d, ppid=%d\n",
        pid, ppid);

    return 0;
}
```

Starting Processes

- Starting new processes can be complicated
  - Application's code has to be loaded into main memory
  - Data structures have to be set up and initialized
  - Library dependencies have to be loaded (recursively)
- Most of the complexity is hidden from users
  - Simply type app or `./app` into the shell
  - Shell and operating system have to take care of all necessary steps
- Programming languages provide functionality to start new processes
  - High-level functionality like `system`
    - `system("ls -lh /")` just does the right thing
  - Low-level functionality like Python's `subprocess`

Fork

- fork starts a new process
  - The parent ID is that of the original process
  - fork returns the new process's PID
- The new process is a copy of the original one
  - Execution resumes at the fork call
  - Otherwise, processes are independent
- Starting other applications needs one more step
  - exec starts another application

```c
int main(void) {
    pid_t pid, ppid, fork_pid;

    fork_pid = fork();
    pid = getpid();
    ppid = getppid();

    printf("pid=%d, ppid=%d,"
        " fork_pid=%d\n",
        pid, ppid, fork_pid);

    return 0;
}
```

- fork starts a new process
  - The parent ID is that of the original process
  - fork returns the new process's PID
- The new process is a copy of the original one
  - Execution resumes at the fork call
  - Otherwise, processes are independent
- Starting other applications needs one more step
  - exec starts another application

```
pid=140226, ppid=139580,
    ↪ fork_pid=140227
pid=140227, ppid=140226,
    ↪ fork_pid=0
```

- execve replaces current application
  - execl etc. are wrappers for execve
- Usual way to start new processes
  - For instance, shell forks and executes new application such as ls or find
- Have to make sure to close files etc.
  - Some resources are inherited by new process
  - File descriptors can be marked close-on-exec

```c
int main(void) {
    pid_t fork_pid = fork();
    if (fork_pid == 0) {
        execl("/usr/bin/ls",
            "ls", NULL);
    } else {
        int status;
        waitpid(fork_pid,
            &status, 0);
        printf("fork_pid=%d,"
            " status=%d\n",
            fork_pid, status);
    }
    return 0;
}
```

Exec

- execve replaces current application
  - execl etc. are wrappers for execve
- Usual way to start new processes
  - For instance, shell forks and executes new application such as ls or find
- Have to make sure to close files etc.
  - Some resources are inherited by new process
  - File descriptors can be marked close-on-exec

```
clone0
clone0.c
...
fork_pid=38843, status=0
```

- All processes are forked
- `init` is started by the kernel
    - Typically systemd (check `/sbin/init`)
    - Responsible for bringing up and down the system
    - Special signal handling
- Processes without a parent are adopted by `init`

```
$ pstree
systemd-+-NetworkManager---2*[{NetworkManager}]
        |-...
        |-systemd-+-(sd-pam)
        |         |-...
        |-systemd-journal
        |-systemd-logind
        |-systemd-machine
        |-systemd-oomd
        |-systemd-resolve
        |-systemd-udevd
        |-systemd-userdbd---3*[systemd-userwor]
        |-...
        `-...
```

```
$ ls -l /proc/$$/fd
lrwx------. 1 user group 64 Nov 29 00:33 0 -> /dev/pts/2
lrwx------. 1 user group 64 Nov 29 00:33 1 -> /dev/pts/2
lrwx------. 1 user group 64 Nov 29 00:33 2 -> /dev/pts/2
```

- `/proc/PID` contains information about specific processes
  - The `fd` directory contains all open file descriptors
    - File descriptors 0, 1 and 2 are standard input, output and error
  - Also available: Current working directory (`cwd`), environment variables (`environ`), application (`exe`) and much more

Shared Memory

- Threads within a process can access the same variables
  - Processes are isolated from each other
- Processes still might have to communicate with each other
  - See Python's `multiprocessing` module, which is implemented using processes
- Portable Operating System Interface (POSIX) defines functionality for this
  - It also covers most aspects shown previously

Shared Memory...

- Separate process address spaces
  - No access to shared variables for communication or synchronization
- Shared memory objects
  - Behave like normal file descriptors
  - Usually implemented as normal files in /dev/shm (tmpfs)
  - mmap allows implicit access
- shm_* functions can be used to manage shared memory objects

```c
int main(void) {
    pid_t pid = getpid();
    int fd = shm_open("/shm",
        O_RDWR | O_CREAT, 0600);
    if (fork() == 0) {
        pwriteall(fd, &pid);
        printf("pid=%d\n", pid);
    } else {
        sleep(1);
        preadall(fd, &pid);
        printf("forked_pid=%d\n", pid);
    }
    shm_unlink("/shm");
    return 0;
}
```

Shared Memory…

- Separate process address spaces
  - No access to shared variables for communication or synchronization
- Shared memory objects
  - Behave like normal file descriptors
  - Usually implemented as normal files in /dev/shm (tmpfs)
  - mmap allows implicit access
- shm_* functions can be used to manage shared memory objects

```
pid=40458
forked_pid=40458
```

Limits

- Linux limits resource usage by default
  - This includes the number of processes, the size of the stack etc.
- Can be shown and modified using `ulimit`
  - `ulimit -a` gives an overview of all limits
- Limits have a soft and a hard limit
  - Users cannot increase above hard limit
- Limits are per-process

```
-t: cpu time (seconds)    unlimited
...
-s: stack size (kbytes)   8192
...
-u: processes             125835
-n: file descriptors      1024
...
-i: pending signals       125835
...
```

File Descriptors

- Maximum for file descriptors is 1,024
  - Cannot open more files afterwards
- Only applies to currently open files
  - Closing files alleviates the problem
- Reached easily in parallel programs
  - For example, each thread opens files

```c
int main(void) {
    int fd;

    for (int i = 0; i < 1024; i++) {
        fd = open("fd.c", O_RDONLY);
        if (fd == -1) {
            printf("error=%s\n",
                strerror(errno));
            return 1;
        }
        printf("Opened file %d.\n", i);
    }

    return 0;
}
```

File Descriptors

- Maximum for file descriptors is 1,024
  - Cannot open more files afterwards
- Only applies to currently open files
  - Closing files alleviates the problem
- Reached easily in parallel programs
  - For example, each thread opens files

```
Opened file 0.
Opened file 1.
Opened file 2.
...
Opened file 1018.
Opened file 1019.
Opened file 1020.
error=Too many open files
```

- Why were we able to open only 1,021 files?
    1. Some file descriptors reserved for safety
    2. Three file descriptors open from the start
    3. 1,021 is the hard limit for file descriptors

```
Opened file 0.
Opened file 1.
Opened file 2.
...
Opened file 1018.
Opened file 1019.
Opened file 1020.
error=Too many open files
```

- Stack size limited to 8 MiB
  - Can be significant with many threads
  - Each thread gets its own stack
- Stack size can be set for each thread
  - Might make sense to limit the size if many threads are running
- Heap is shared between all threads
  - `malloc` is thread-safe

```c
void rec(int depth) {
    printf("depth=%d\n", depth);
    rec(depth + 1);
}

int main(void) {
    rec(0);
    return 0;
}
```

- Stack size limited to 8 MiB
  - Can be significant with many threads
  - Each thread gets its own stack
- Stack size can be set for each thread
  - Might make sense to limit the size if many threads are running
- Heap is shared between all threads
  - `malloc` is thread-safe
- Crashes after ca. 260,000 steps
  - Around 32 bytes stack memory per step

```
...
depth=261754
depth=261755
depth=261756
depth=261757
depth=261758
segmentation fault (core dumped)
```

- Functionality presented so far allows starting new processes
  - Control is quite limited using `fork` and `exec`
- How do we start new threads?
  - Using established interfaces like OpenMP and POSIX Threads
  - We will still take a deeper look to understand the internals
- Linux has a `clone` system call that offers more control
  - This is also used to implement POSIX Threads semantics

- clone allows creating new processes
- Offers more control than fork
  - Address space, file descriptors and signal handlers can be shared
  - Allows placing processes in namespaces
- Allows specifying function to execute
  - Stack has to be managed manually
- There is a newer clone3 system call

```c
int main(void) {
    int status;
    pid_t pid;

    clone(func,
        stack + sizeof(stack),
        SIGCHLD,
        "Hello world.");
    pid = wait(&status);
    printf("pid=%d, cpid=%d, "
        "status=%d\n",
        getpid(), pid,
        WEXITSTATUS(status));
    return 0;
}
```

Clone

- clone allows creating new processes
- Offers more control than fork
  - Address space, file descriptors and signal handlers can be shared
  - Allows placing processes in namespaces
- Allows specifying function to execute
  - Stack has to be managed manually
- There is a newer clone3 system call

```c
char stack[1024 * 1024];

int func(void* arg) {
    printf("%s\n", (char*)arg);
    printf("pid=%d\n",
        getpid());
    return 42;
}
```

```
Hello world.
pid=48472
pid=48471, cpid=48472, status=42
```

- clone can also be used for threads
  - Process is placed in the same thread group
  - Shares PID with parent process
  - Has a separate thread ID (TID)
- Semantics for POSIX Threads
  - Otherwise, threads could have own PIDs
- clone is very specialized
  - Only use if you know what you are doing
  - Examples most likely contain bugs (no thread-local storage etc.)

```c
int main(void) {
    clone(func,
        stack + sizeof(stack),
        CLONE_THREAD
        | CLONE_SIGHAND
        | CLONE_VM,
        "Hello world.");
    while (atomic_load(&lock)
        == 0);
    printf("pid=%d\n",
        getpid());

    return 0;
}
```

Clone...

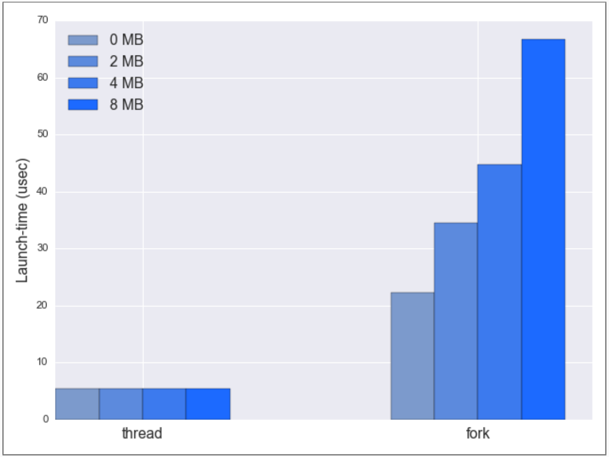
- clone can also be used for threads
  - Process is placed in the same thread group
  - Shares PID with parent process
  - Has a separate thread ID (TID)
- Semantics for POSIX Threads
  - Otherwise, threads could have own PIDs
- clone is very specialized
  - Only use if you know what you are doing
  - Examples most likely contain bugs (no thread-local storage etc.)

```c
char stack[1024 * 1024];
atomic_int lock = 0;

int func(void* arg) {
    printf("%s\n", (char*)arg);
    printf("pid=%d\n",
        getpid());
    atomic_store(&lock, 1);
    return 42;
}
```

```
Hello world.
pid=50663
pid=50663
```

- Starting threads is significantly faster than starting processes
  - 5 µs vs. more than 20 µs
- Threads share address space
  - No copies necessary
- Processes copy virtual memory
  - Changes are copy-on-write



[Bendersky, 2018]

Outline

Summary

- Parallel programming often requires low-level knowledge
  - Having a basic understanding of operating system concepts is necessary
- Privileged operations have to be performed in kernel mode
  - Switching to kernel mode can be expensive
- Modern operating systems are thread-safe and reentrant
  - Can execute applications and system calls in parallel
- There are performance characteristics and resource limits to keep in mind
  - Threads are typically faster to spawn than processes

References

[Bendersky, 2018] Bendersky, E. (2018). **Launching Linux threads and processes with clone.** https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/.

[Lamport, 1978] Lamport, L. (1978). **Time, clocks, and the ordering of events in a distributed system.** *Commun. ACM*, 21(7):558–565.

[Linux man-pages project, 2021] Linux man-pages project (2021). **strerror(3).** https://man7.org/linux/man-pages/man3/strerror.3.html.

[Sven, 2006] Sven (2006). **This explains the CPU ring scheme.** https://en.wikipedia.org/wiki/File:CPU_ring_scheme.svg.

[Tumenbayar, 2002] Tumenbayar, E. (2002). **Linux SMP HOWTO.** https://tldp.org/HOWTO/SMP-HOWTO-3.html.

Daemon

- Processes part of process group and session
  - Process group is sent `SIGHUP` when session leader terminates
  - Shell is usually the session leader
- `fork` and `setsid` for daemons
  - `setsid` makes process session leader
  - Leader can run in the background

```c
int main(void) {
    pid_t pid, ppid;

    if (fork() == 0) {
        setsid();
    }
    pid = getpid();
    ppid = getppid();
    printf("pid=%d, ppid=%d, "
        "sid=%d\n",
        pid, ppid, getsid(0));

    return 0;
}
```

Daemon

- Processes part of process group and session
  - Process group is sent `SIGHUP` when session leader terminates
  - Shell is usually the session leader
- `fork` and `setsid` for daemons
  - `setsid` makes process session leader
  - Leader can run in the background

```
pid=38378, ppid=6865, sid=6865
pid=38379, ppid=38378, sid=38379
```