CSE 120 Principles of Operating Systems

Fall 2026

Lecture 3: Processes

Geoffrey M. Voelker

Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in this class
- Today's topics are processes and process management
  - What are the units of execution?
  - How are those units of execution represented in the OS?
  - How is work scheduled in the CPU?
  - What are the possible execution states of a process?
  - How does a process move from one state to another?

The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
- Also called a job or a task or a sequential process
- A process is a program in execution
  - It defines the instruction-at-a-time execution of a program
  - Programs are static entities with the potential for execution

Process Components

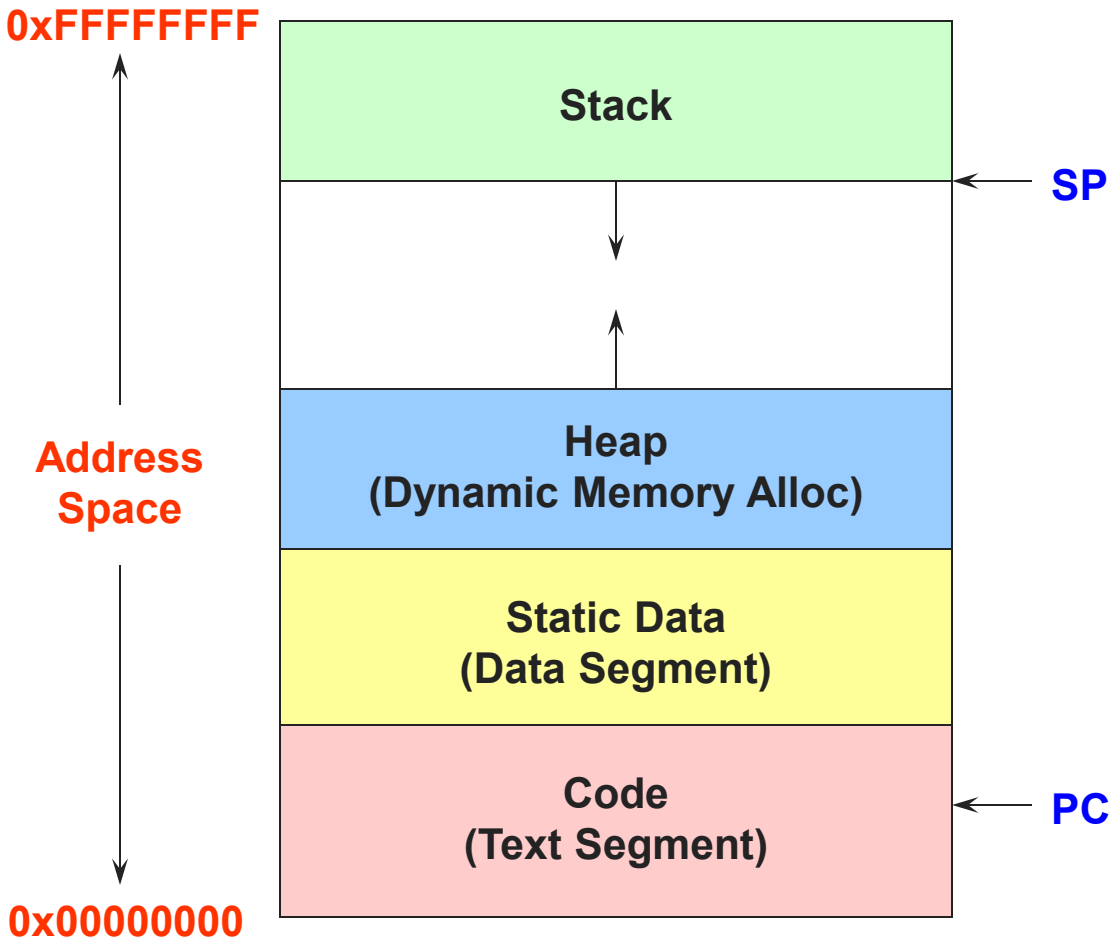
- A process contains all state for a program in execution
  - An address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - » Open files, network connections, etc.
- A process is named using its process ID (PID)

Unix PIDs

```
top - 10:05:04 up 373 days,  1:29,  1 user,  load average: 0.00, 0.01, 0.00
Tasks: 206 total,   1 running, 122 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 98967544 total, 72343520 free,  1141584 used, 25482440 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 96887280 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
27210 voelker   20   0   33536   3692   3160 R   0.3  0.0   0:00.05 top
27211 root      20   0   66208   5360   4664 S   0.3  0.0   0:00.01 sshd
27877 root      20   0       0      0      0 I   0.3  0.0   0:05.72 kworker/0:2
    1 root      20   0  225572   9432   6796 S   0.0  0.0  19:46.34 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:07.77 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root      20   0       0      0      0 I   0.0  0.0   0:57.38 kworker/u1+
    7 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    8 root      20   0       0      0      0 S   0.0  0.0   0:17.02 ksoftirqd/0
    9 root      20   0       0      0      0 I   0.0  0.0 191:58.78 rcu_sched
   10 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
   11 root      rt   0       0      0      0 S   0.0  0.0   0:02.67 migration/0
   12 root      rt   0       0      0      0 S   0.0  0.0   0:57.85 watchdog/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:55.63 watchdog/1
   16 root      rt   0       0      0      0 S   0.0  0.0   0:03.08 migration/1
```

Basic Process Address Space



Process State

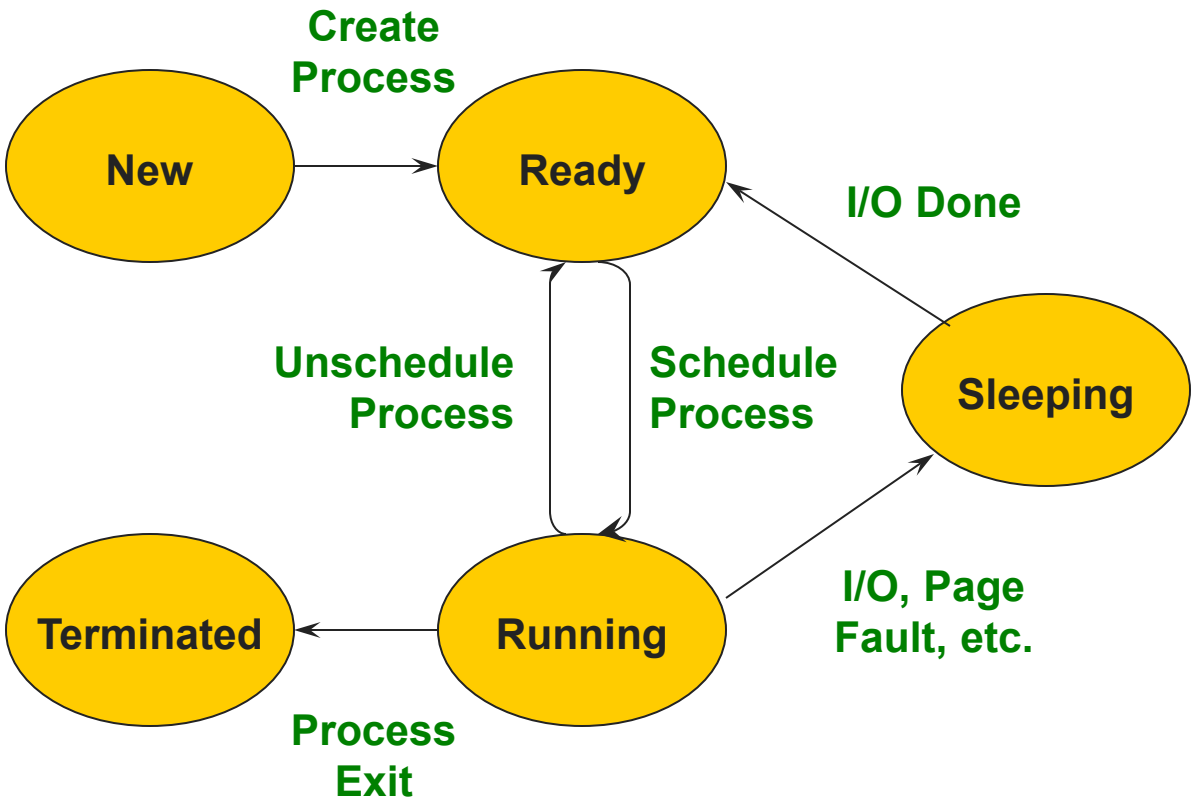
- A process has a state that indicates what it is doing
  - Running: Executing instructions on the CPU
    - » It is the process that has control of the CPU
    - » How many processes can be in the running state simultaneously?
  - Ready: Waiting to be assigned to the CPU
    - » Ready to execute, but another process is executing on the CPU
  - Sleeping: Waiting for an event, e.g., I/O completion
    - » It cannot make progress until event is signaled (disk completes)
- As a process executes, it moves from state to state
  - Unix "ps": STAT/S column indicates execution state
  - What state do you think a process is in most of the time?
  - How many processes can a system support?

Unix Process States

```
top - 10:05:04 up 373 days,  1:29,  1 user,  load average: 0.00, 0.01, 0.00
Tasks: 206 total,   1 running, 122 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 98967544 total, 72343520 free,  1141584 used, 25482440 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 96887280 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
27210 voelker   20   0   33536   3692   3160 R   0.3  0.0   0:00.05 top
27211 root      20   0   66208   5360   4664 S   0.3  0.0   0:00.01 sshd
27877 root      20   0       0      0      0 I   0.3  0.0   0:05.72 kworker/0:2
    1 root      20   0  225572   9432   6796 S   0.0  0.0  19:46.34 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:07.77 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:+
    6 root      20   0       0      0      0 I   0.0  0.0   0:57.38 kworker/u1+
    7 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_+
    8 root      20   0       0      0      0 S   0.0  0.0   0:17.02 ksoftirqd/0
    9 root      20   0       0      0      0 I   0.0  0.0 191:58.78 rcu_sched
   10 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
   11 root      rt   0       0      0      0 S   0.0  0.0   0:02.67 migration/0
   12 root      rt   0       0      0      0 S   0.0  0.0   0:57.85 watchdog/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:55.63 watchdog/1
   16 root      rt   0       0      0      0 S   0.0  0.0   0:03.08 migration/1
```

Process State Graph



Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in a particular process state
- The OS data structure representing each process is called the Process Control Block (PCB)
- The PCB contains all the info about a process
- The PCB also is where the OS keeps all its CPU hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

PCB Data Structure

- The PCB contains a huge amount of information in one large structure
  - » Process ID (PID)
  - » Process state (ready, sleeping, etc.)
  - » Hardware state (PC, SP, general registers)
  - » Memory management
  - » Scheduling
  - » Accounting
  - » Pointers for process state queues
  - » Etc.
- It is a heavyweight abstraction

struct task_struct (Linux)

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int		__state;

	/* saved state for "spinlock sleepers" */
	unsigned int		saved_state;
	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void		*stack;
	refcount_t		usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int		flags;
	unsigned int		ptrace;

#ifdef CONFIG_MEM_ALLOC_PROFILING
	struct alloc_tag		*alloc_tag;
#endif

#ifdef CONFIG_SMP
	int		on_cpu;
	struct __call_single_node		wake_entry;
	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;

	struct sched_entity		se;
	struct sched_rt_entity		rt;
	struct sched_dl_entity		dl;
	struct sched_dl_entity		*dl_server;
#ifdef CONFIG_SCHED_CLASS_EXT
	struct sched_ext_entity		scx;
#endif
	const struct sched_class		*sched_class;

#ifdef CONFIG_SCHED_CORE
	struct rb_node		core_node;
	unsigned long		core_cookie;
	unsigned int		core_occupation;
#endif

#ifdef CONFIG_CGROUP_SCHED
	struct task_group		*sched_task_group;
#endif
```

struct task_struct (2)

```
#ifdef CONFIG_UCLAMP_TASK
        /*
         * Clamp values requested for a scheduling entity.
         * Must be updated with task_rq_lock() held.
         */
        struct uclamp_se                        uclamp_req[UCLAMP_CNT];
        /*
         * Effective clamp values used for a scheduling entity.
         * Must be updated with task_rq_lock() held.
         */
        struct uclamp_se                        uclamp[UCLAMP_CNT];
#endif
        struct sched_statistics                 stats;

#ifdef CONFIG_PREEMPT_NOTIFIERS
        /* List of struct preempt_notifier: */
        struct hlist_head                       preempt_notifiers;
#endif
#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int                            btrace_seq;
#endif
        unsigned int                            policy;
        unsigned long                           max_allowed_capacity;
        int                                     nr_cpus_allowed;
        const cpumask_t                         *cpus_ptr;
        cpumask_t                               *user_cpus_ptr;
        cpumask_t                               cpus_mask;
        void                                    *migration_pending;
#ifdef CONFIG_SMP
        unsigned short                          migration_disabled;
#endif
        unsigned short                          migration_flags;

#ifdef CONFIG_PREEMPT_RCU
        int                                     rcu_read_lock_nesting;
        union rcu_special                       rcu_read_unlock_special;
        struct list_head                        rcu_node_entry;
        struct rcu_node                         *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */

#ifdef CONFIG_TASKS_RCU
        unsigned long                           rcu_tasks_nvcsw;
        u8                                      rcu_tasks_holdout;
        u8                                      rcu_tasks_idx;
        int                                     rcu_tasks_idle_cpu;
        struct list_head                        rcu_tasks_holdout_list;
        int                                     rcu_tasks_exit_cpu;
        struct list_head                        rcu_tasks_exit_list;
#endif /* #ifdef CONFIG_TASKS_RCU */

#ifdef CONFIG_TASKS_TRACE_RCU
        int                                     trc_reader_nesting;
        int                                     trc_ipi_to_cpu;
        union rcu_special                       trc_reader_special;
        struct list_head                        trc_holdout_list;
        struct list_head                        trc_blkd_node;
        int                                     trc_blkd_cpu;
#endif /* #ifdef CONFIG_TASKS_TRACE_RCU */

        struct sched_info                       sched_info;

        struct list_head                        tasks;
#ifdef CONFIG_SMP
        struct plist_node                       pushable_tasks;
        struct rb_node                          pushable_dl_tasks;
#endif

        struct mm_struct                        *mm;
        struct mm_struct                        *active_mm;
        struct address_space                    *faults_disabled_mapping;

        int                                     exit_state;
        int                                     exit_code;
        int                                     exit_signal;
```

struct task_struct (3)

```
        /* The signal sent when the parent dies: */
        int                                     pdeath_signal;
        /* JOBCTL_*, siglock protected: */
        unsigned long                           jobctl;

        /* Used for emulating ABI behavior of previous Linux versions: */
        unsigned int                            personality;

        /* Scheduler bits, serialized by scheduler locks: */
        unsigned                                sched_reset_on_fork:1;
        unsigned                                sched_contributes_to_load:1;
        unsigned                                sched_migrated:1;
        unsigned                                sched_task_hot:1;

        /* Force alignment to the next boundary: */
        unsigned                                :0;

        /* Unserialized, strictly 'current' */

        /*
         * This field must not be in the scheduler word above due
         * queueing no longer being serialized by p->on_cpu. However:
         *
         * p->XXX = X;                          ttwu()
         * schedule()                             if (p->on_rq && ..) // false
         *   smp_mb__after_spinlock();      if (smp_load_acquire(&p->on_cpu) && //true
         *   deactivate_task()                        ttwu_queue_wakelist())
         *     p->on_rq = 0;                p->sched_remote_wakeup = Y;
         *
         * guarantees all stores of 'current' are visible before
         * ->sched_remote_wakeup gets used, so it can be in this word.
         */
        unsigned                                sched_remote_wakeup:1;
#ifdef CONFIG_RT_MUTEXES
        unsigned                                sched_rt_mutex:1;
#endif
```

+625 lines

```
        /* Bit to tell TOMOYO we're in execve(): */
        unsigned                                in_execve:1;
        unsigned                                in_iowait:1;
#ifndef TIF_RESTORE_SIGMASK
        unsigned                                restore_sigmask:1;
#endif
#ifdef CONFIG_MEMCG_V1
        unsigned                                in_user_fault:1;
#endif
#ifdef CONFIG_LRU_GEN
        /* whether the LRU algorithm may apply to this access */
        unsigned                                in_lru_fault:1;
#endif
#ifdef CONFIG_COMPAT_BRK
                                                brk_randomized:1;

GROUPS
 userland-initiated cgroup migration */
                                                no_cgroup_migration:1;
rozen/stopped (used by the cgroup freezer) */
                                                frozen:1;
#endif
#ifdef CONFIG_BLK_CGROUP
        unsigned                                use_memdelay:1;
#endif
#ifdef CONFIG_PSI
        /* Stalled due to lack of memory */
        unsigned                                in_memstall:1;
#endif
#ifdef CONFIG_PAGE_OWNER
        /* Used by page_owner=on to detect recursion in page tracking. */
        unsigned                                in_page_owner:1;
#endif
#ifdef CONFIG_EVENTFD
        /* Recursion prevention for eventfd_signal() */
        unsigned                                in_eventfd:1;
#endif
```

PCBs and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in the process PCB
  - What happens to the code that is executing?
- The steps of changing the CPU hardware state from one process to another is called a context switch
  - This can happen 100 or 1000 times a second!

State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has one queue for each process state
  - Ready, sleeping, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, the OS unlinks the PCB from one queue and linked into another

Process Creation

- A process is created by another process
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 0 or 1))?
- The parent defines (or donates) resources and privileges for its children
  - User ID is inherited: children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel

Process Creation: Windows

- The system call on Windows for creating a process is called, surprisingly enough, CreateProcess:

  `BOOL CreateProcess(char *prog, char *args)` (simplified)

- CreateProcess
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by “prog” into the address space
  - Copies “args” into memory allocated in address space
  - Initializes the saved hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the ready queue

CreateProcessA function (processthreadsapi.h)

Article • 09/23/2022 • 13 minutes to read

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

Syntax

C++ Copy

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

In this article

Syntax
Parameters
Return value
Remarks

Process Creation: Unix

- In Unix, the system call fork() creates a process

```
int fork()
```

- fork()
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- Fork returns twice
  - Huh?
  - Returns the child's PID to the parent, "0" to the child

FORK(2) BSD System Calls Manual FORK(2)

NAME

fork -- create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t
fork(void);
```

DESCRIPTION

Fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES

Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

ERRORS

Fork() will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.

fork()

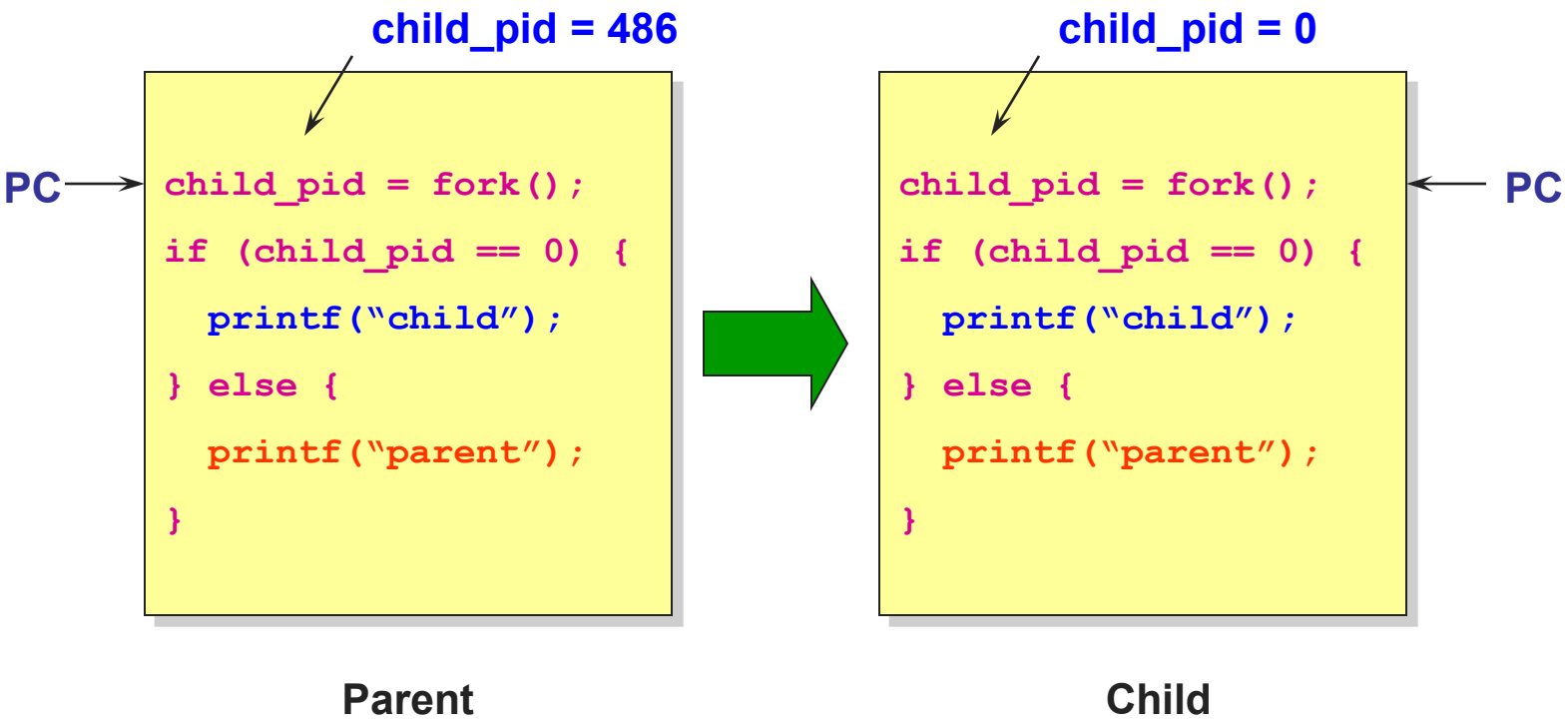
```
int main(int argc, char *argv[])
{
   char *name = argv[0];
   int child_pid = fork();
   if (child_pid == 0) {
       printf("Child of %s is %d\n", name, getpid());
       return 0;
   } else {
       printf("My child is %d\n", child_pid);
       return 0;
   }
}
```

What does this program print?

Example Output

```
alpenglow (18) ~/tmp> cc fork.c
alpenglow (19) ~/tmp> ./a.out
My child is 486
Child of a.out is 486
```

Duplicating Address Spaces



Divergence

child_pid = 486

```
child_pid = fork();
if (child_pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

Parent

child_pid = 0

```
child_pid = fork();
if (child_pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

Child

Example Continued

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> a.out
Child of a.out is 693
My child is 693
```

Why is the output in a different order?

Why fork()?

- Very useful when the child…
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task
- Example: Web server

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    Handle client request and exit
  } else {
    Close socket
  }
}
```

New Programs: Unix

- Then…how do we start a new program?

```
int exec(char *prog, char *argv[])
```

- exec() system call
  - Stops the current process
  - Loads the program "prog" into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

- Can exec ever return?

Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), Windows: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Free allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures and free
- Note that a process does not need to clean up itself
  - Why does the OS have to do it?

wait() a second…

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Unix wait() (Windows: WaitForSingleObject)
  - Suspends the current process until any child process ends
  - waitpid() suspends until the specified child process ends
- Wait returns a status value…what is it?
- Unix: Every process must be “reaped” by a parent
  - What happens if a parent process exits before a child?
  - What do you think a “zombie” process is?

Unix Shells

```
while (1) {
   char *cmd = read_command();
   int child_pid = fork();
   if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR for pipes, redirection, etc.
        exec(cmd);
        printf("exec failed");
   } else {
        waitpid(child_pid);
   }
}
```

Fun With Exec (Extra)

- fork() is used to create a new process, exec is used to load a program into the address space
- What happens if you run “exec bash” in your shell?
- What happens if you run “exec ls” in your shell? Try it.

- fork() can return an error. Why might this happen?

Process Summary

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (Win), fork/exec (Unix)

Next time…

- Read Chapters 26, 27
- Complete Project 0
- Submit group info
- Complete Homework 1