5. Processes and Memory Management

- Process Abstraction
- Introduction to Memory Management
- Process Implementation
- States and Scheduling
- Programmer Interface
- Process Genealogy
- Daemons, Sessions and Groups

5. Processes and Memory Management

Logical Separation of Processes

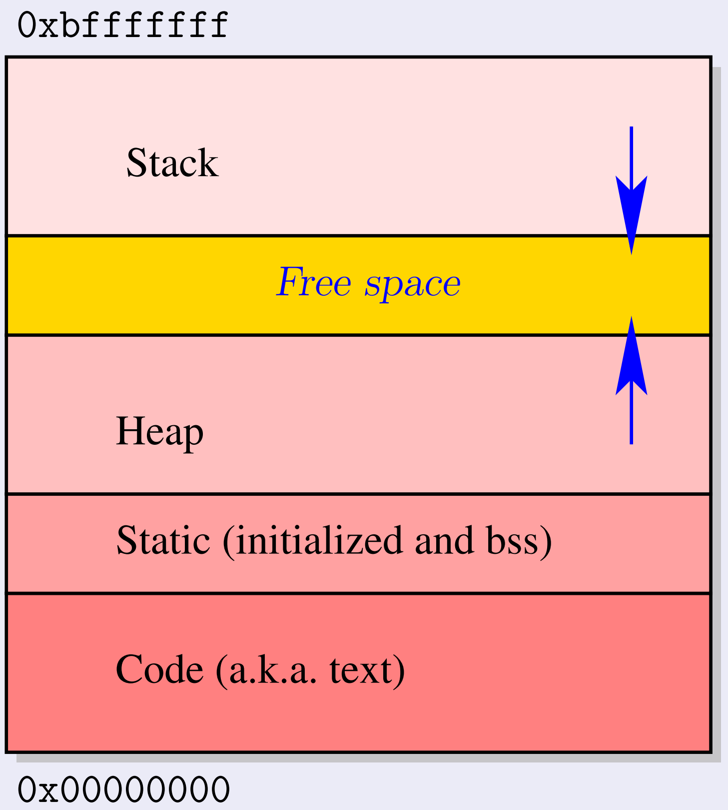
Kernel Address Space for a Process

- Process *descriptor*
  - Memory mapping
  - Open file descriptors
  - Current directory
  - Pointer to kernel stack
- Kernel stack
  - Small by default; grows in extreme cases of nested interrupts/exceptions
- Process table
  - Associative table of PID-indexed process descriptors
  - Doubly-linked tree (links to both children and parent)

Logical Separation of Processes

User Address Space for a Process

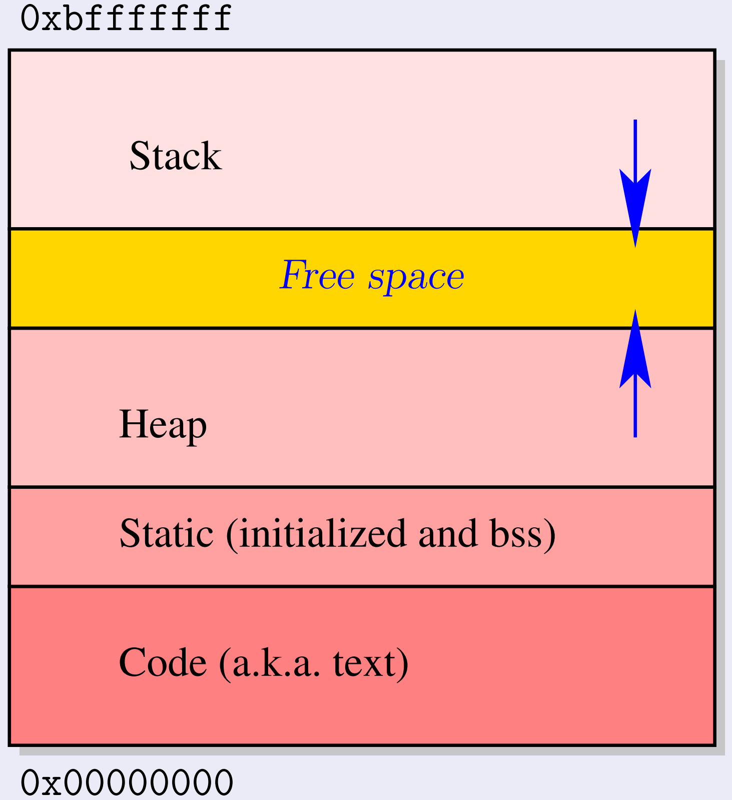
- Allocated and initialized when loading and executing the program
- *Memory accesses in user mode are restricted to this address space*



Logical Segments in Virtual Memory

Per-Process Virtual Memory Layout

- *Code* (also called *text*) segment
  - Linux: ELF format for object files (`.o` and executable)
- *Static Data* segments
  - Initialized global (and C `static`) variables
  - Uninitialized global variables
    - Zeroed when initializing the process, also called *bss*
- *Stack* segment
  - Stack frames of function calls
  - Arguments and local variables, also called `auto`matic variables in C
- *Heap* segment
  - Dynamic allocation (`malloc()`)



System Call: `brk()`

Resize the Heap Segment

```
#include <unistd.h>

int brk(void *end_data_segment);

void *sbrk(intptr_t displacement);
```

Semantics

- Sets the *end* of the data segment, which is also the end of the heap
  - `brk()` sets the address directly and returns **0** on success
  - `sbrk()` adds a displacement (possibly **0**) and returns the *starting* address of the new area (it is a C function, front-end to `sbrk()`)
- Both are *deprecated* as "programmer interface" functions, i.e., they are meant for kernel development only

Memory Address Space Example

```
#include <stdlib.h>
#include <stdio.h>

double t[0x02000000];

void segments()
{
  static int s = 42;
  void *p = malloc(1024);

  printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
         "static\t%010p\nstatic\t%010p\ntext\t%010p\n",
         &p, sbrk(0), p, t, &s, segments);
}

int main(int argc, char *argv[])
{
  segments();
  exit(0);
}
```

Memory Address Space Example

```
#include <stdlib.h>
#include <stdio.h>

double t[0x02000000];

void segments()
{
  static int s = 42;
  void *p = malloc(1024);

  printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
         "static\t%010p\nstatic\t%010p\ntext\t%010p\n",
         &p, sbrk(0), p, t, &s, segments);
}

int main(int argc, char *argv[])
{
  segments();
  exit(0);
}
```

Sample Output

stack	0xbff86fe0
brk	0x1806b000
heap	0x1804a008
static (bss)	0x08049720
static (initialized)	0x080496e4
text	0x080483f4

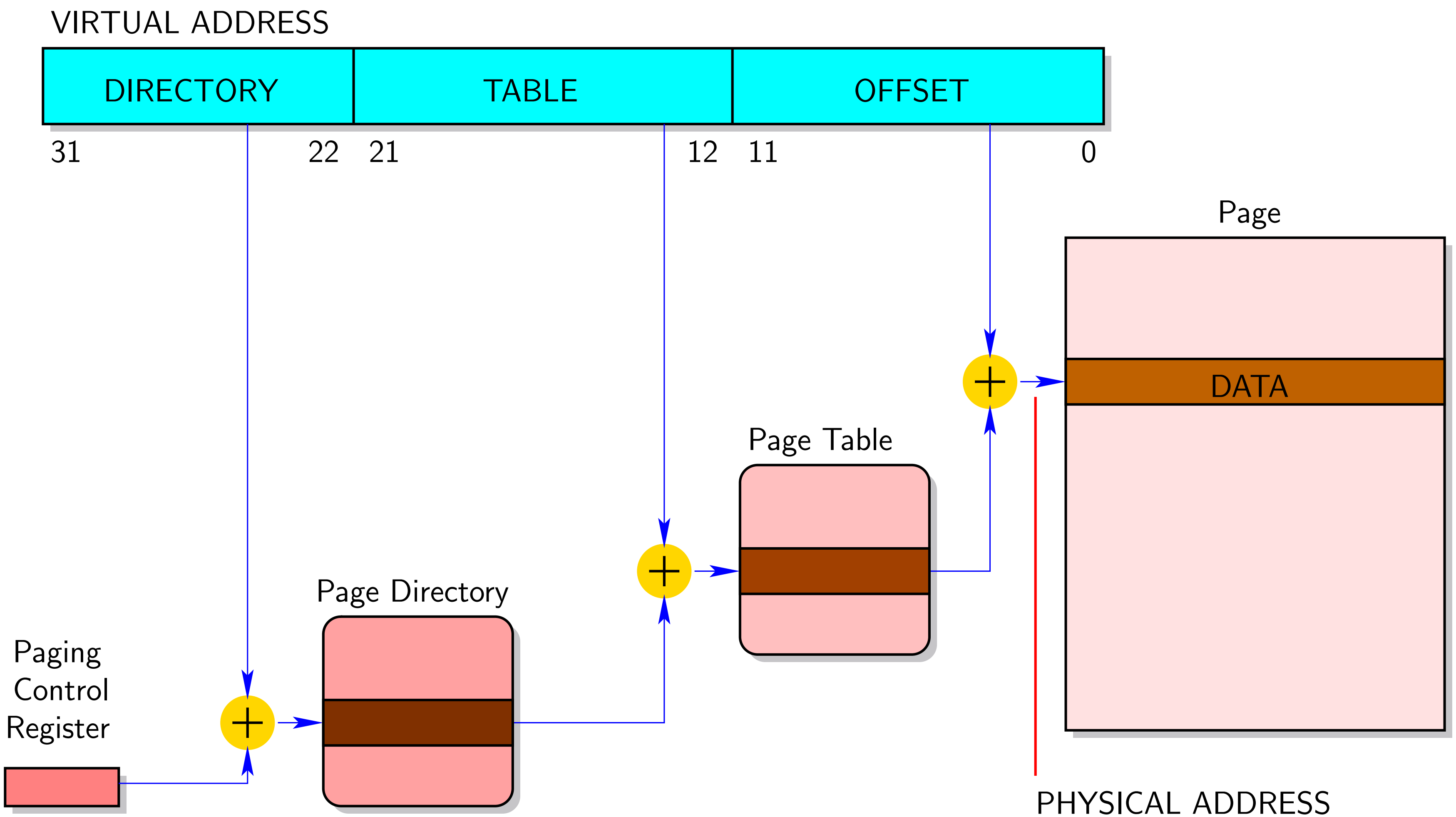
5. Processes and Memory Management

Introduction to Memory Management

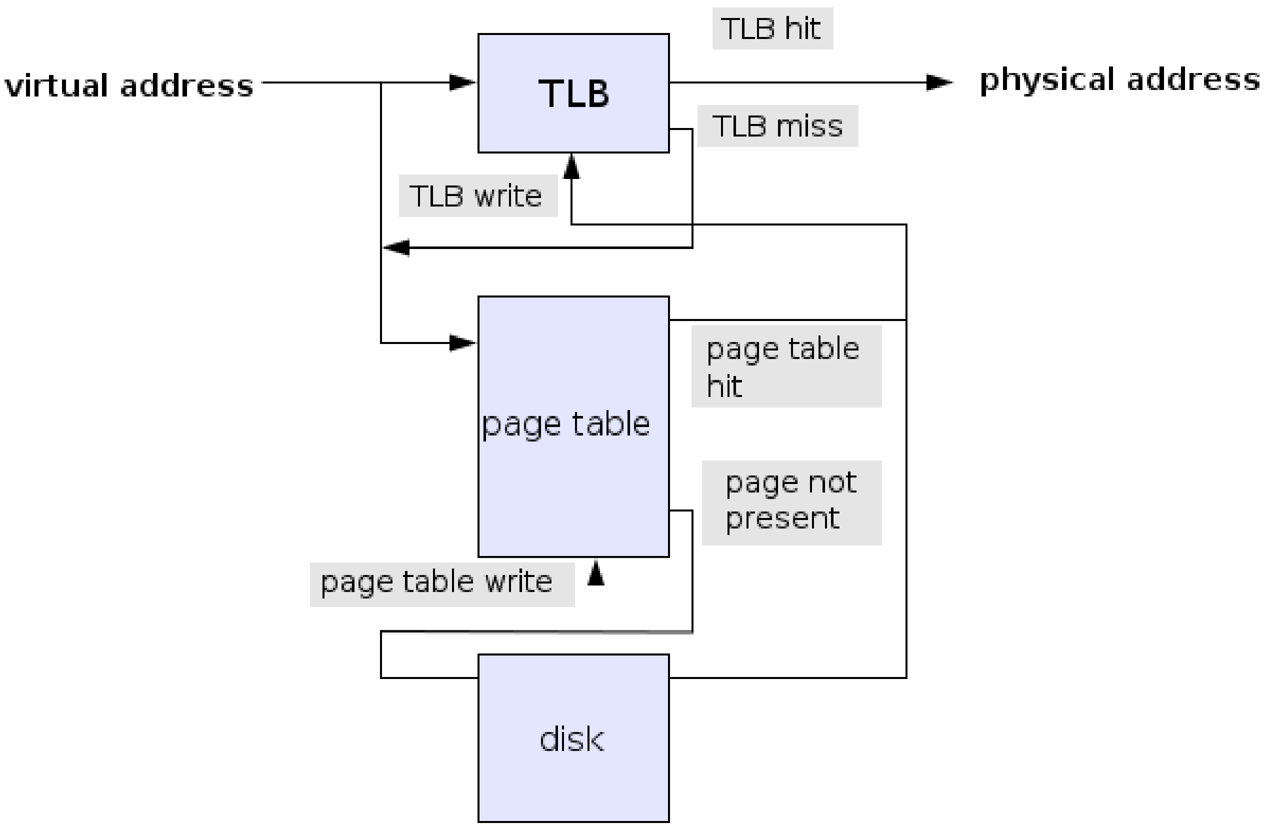
Paging Basics

- Processes access memory through *virtual* addresses
  - Simulates a large *interval* of memory addresses
  - Simplifies memory management
  - Automatic *translation* to *physical* addresses by the CPU (MMU/TLB circuits)
- *Paging* mechanism
  - Provide a protection mechanism for memory regions, called *pages*
  - Fixed $\mathbf{2^n}$ page size(s), e.g., 4kB and 2MB on x86
  - The kernel implements a *mapping* of physical pages to virtual ones
    - *Different for every process*
- Key mechanism to ensure *logical separation* of processes
  - Hides kernel and other processes' memory
  - Expressive and efficient address-space protection and separation

Address Translation for Paged Memory



Page Table Actions



Page Table Structure(s)

Page Table Entry

- Physical address
- Valid/Dirty/Accessed
- Kernel R/W/X
- User R/W/X

Physical Page Mapping

E.g., Linux's `mem_map_t` structure:

- `counter` – how many users are mapping a physical page
- `age` – timestamp for swapping heuristics: Belady algorithm
- `map_nr` – Physical page number

Plus a free area for page allocation and dealocation

Saving Resources and Enhancing Performance

Lazy Memory Management

- Motivation: high-performance memory allocation
  - *Demand-paging*: delay the allocation of a memory page and its *mapping* to the process's virtual address space until the process *accesses* an address in the range associated with this page
  - Allows *overcommitting*: more economical than eager allocation (like overbooking in public transportation)
- Motivation: high-performance process creation
  - *Copy-on-write*: when cloning a process, do not replicate its memory, but mark its pages as "*need to be copied on the next write access*"
  - Critical for UNIX
    - Cloning is the only way to create a new process
    - Child processes are often short-lived: they are quickly overlapped by the execution of another program (see `execve()`)

Software Caches

- Buffer cache for block devices, and page cache for file data
- Swap cache to keep track of clean pages in the swap (disk)

C Library Function: malloc()

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to a *fresh interval* of `size` bytes of *heap* memory
- Return `NULL` on error
- See also `calloc()` and `realloc()`

C Library Function: `malloc()`

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to a *fresh interval* of `size` bytes of *heap* memory
- Return `NULL` on error
- See also `calloc()` and `realloc()`
- Warning: many OSes *overcommit* memory by default (e.g., Linux)
  - Minimal memory availability check and optimistically return non-`NULL`
  - Assume processes will not use all the memory they requested
  - When the system really runs out of free physical pages (after all swap space has been consumed), a kernel heuristic selects a non-`root` process and kills it to free memory for the requester (quite unsatisfactory, but often sufficient)

System Call: `free()`

Free Dynamic Memory

```
#include <stdlib.h>

void free(void *ptr);
```

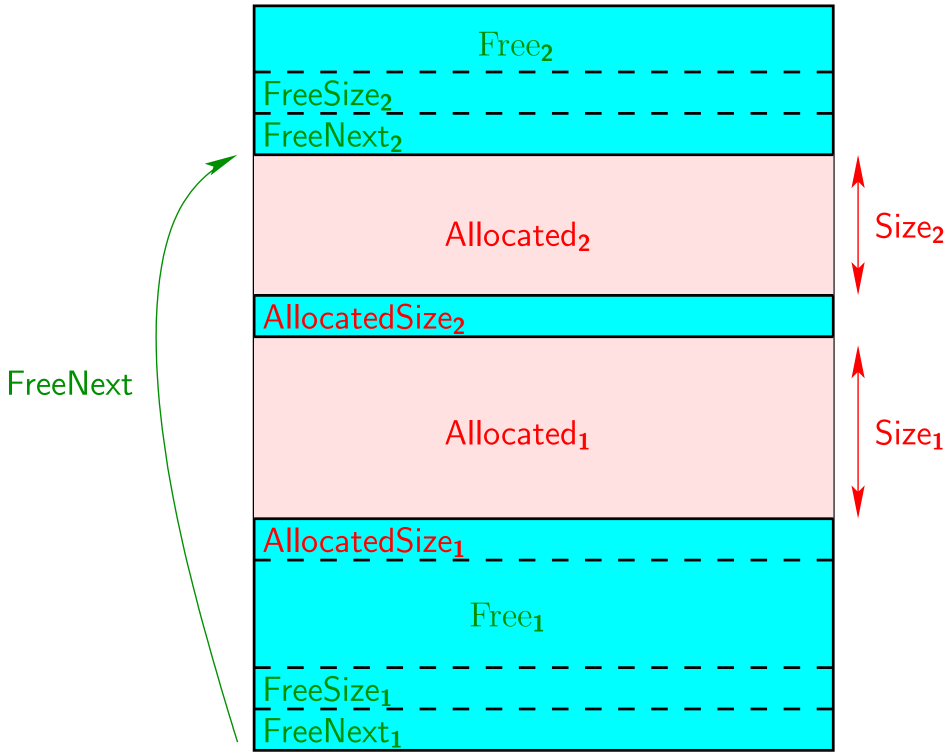
Semantics

- Frees the memory interval pointed to by `ptr`, which *must* be the return value of a previous `malloc()`
- Undefined behaviour if it is not the case
  (very nasty in general, because the bug may reveal much later)
- No operation is performed if `ptr` is `NULL`
- The dedicated `valgrind` tool instruments memory accesses and system calls to track memory leaks, phantom pointers, corrupt calls to `free()`, etc.

Memory Management of User Processes

Memory Allocation

- Appears in every aspect of the system
  - Major performance impact: highly optimized
- *Free list*: record linked list of free zones in the *free* memory space only
  - Record the address of the *next free zone*
  - Record the size of the allocated zone prior to its effective bottom address



Memory Management of User Processes

Memory Allocation

- Appears in every aspect of the system
  - Major performance impact: highly optimized
- *Buddy system*: allocate contiguous pages of physical memory
  - Coupled with free list for intra-page allocation
  - Contiguous physical pages improve performance (better TLB usage and DRAM control)

Intervals:
A: 64kB
B: 128kB
C: 64kB
D: 128kB

Empty	1024					
Allocate A	A	64	128	256		512
Allocate B	A	64	B	256		512
Allocate C	A	C	B	256		512
Allocate D	A	C	B	D	128	512
Free C	A	64	B	D	128	512
Free A	128		B	D	128	512
Free B	256			D	128	512
Free D	1024					

5. Processes and Memory Management

Process Descriptor

Main Fields of the Descriptor

State	ready/running, stopped, zombie...
Kernel stack	typically one memory page
Flags	e.g., `FD_CLOEXEC`
Memory map	pointer to table of memory page descriptors (maps)
Parent	pointer to parent process (allow to obtain PPID)
TTY	control terminal (if any)
Thread	TID and thread information
Files	current directory and table of file descriptors
Limits	resource limits, see `getrlimit()`
Signals	signal handlers, masked and pending signals

Operations on Processes

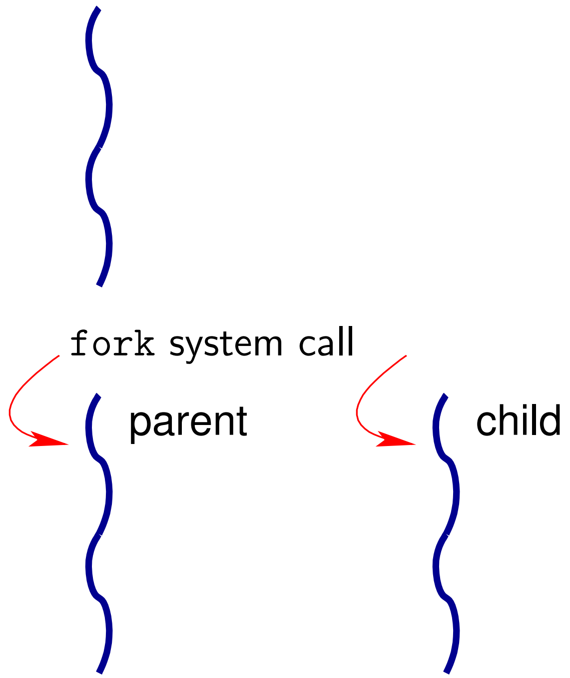
Basic Operations on Processes

- Cloning
  `fork()` system call, among others
- Joining (*see next chapter*)
  `wait()` system call, among others
- Signaling events (*see next chapter*)
  `kill()` system call, signal handlers

Creating Processes

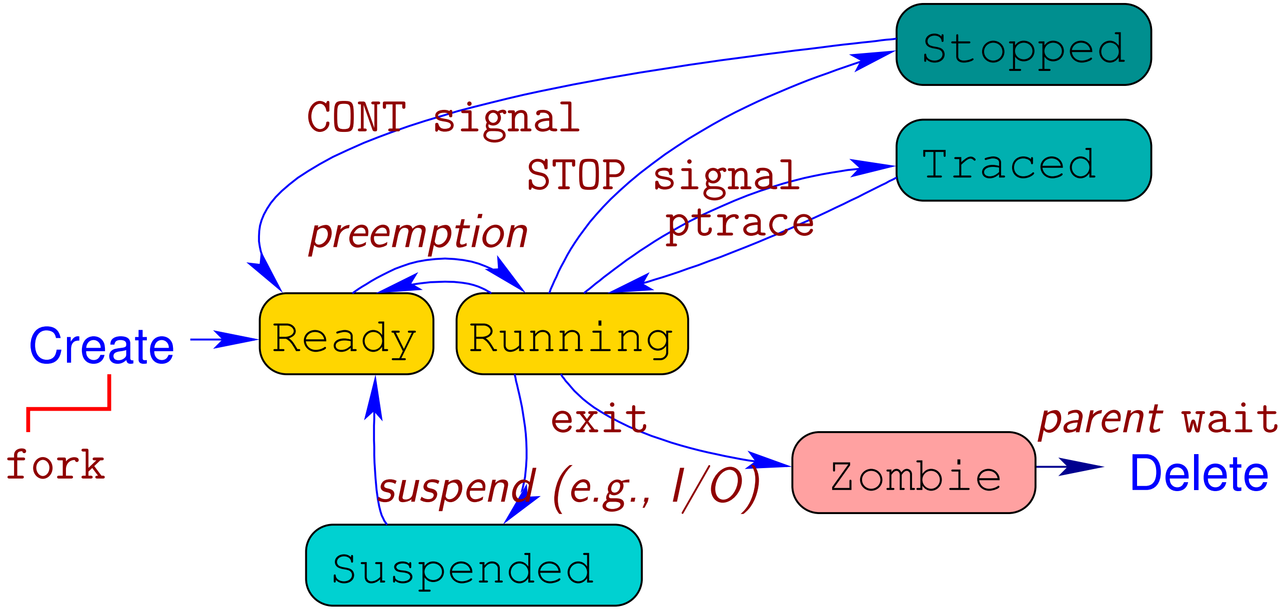
Process Duplication

- Generate a clone of the *parent* process
- The *child* is almost identical
  - It executes the same program
  - In a copy of its virtual memory space



5. Processes and Memory Management

Process States

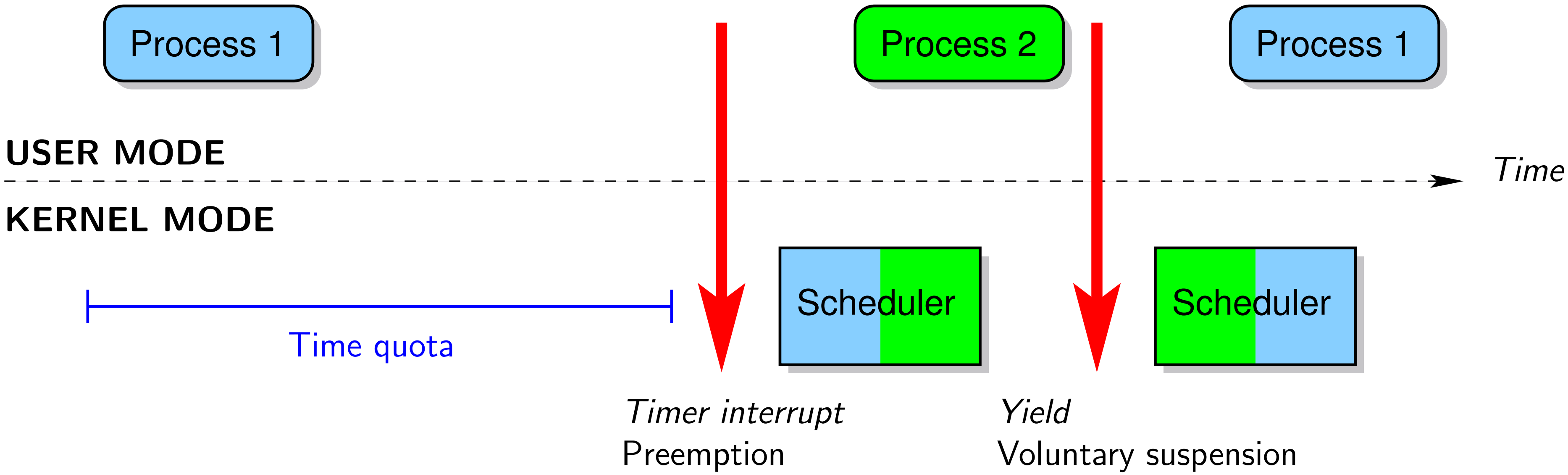


- Ready (runnable) process waits to be scheduled
- Running process make progress on a hardware thread
- Stopped process awaits a continuation signal
- Suspended process awaits a wake-up condition from the kernel
- Traced process awaits commands from the debugger
- Zombie process retains termination status until parent is notified
- Child created as Ready after `fork()`
- Parent is Stopped between `vfork()` and child `execve()`

Process Scheduling

Preemption

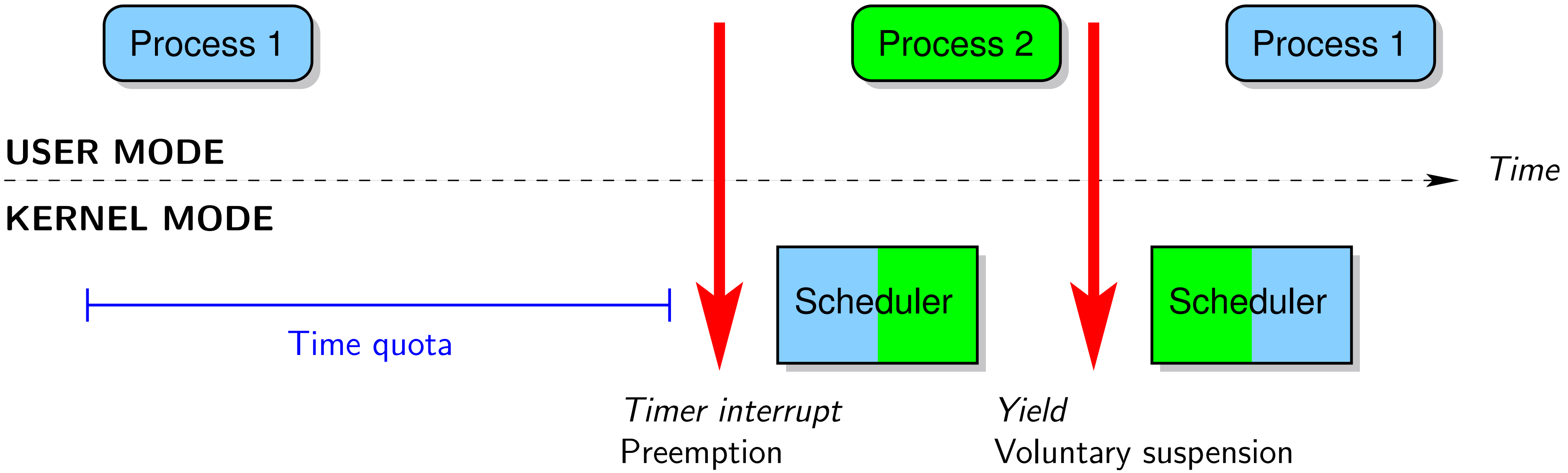
- Default for multiprocessing environments
- Fixed *time quota* (typically $\mathbf{1}$ms to $\mathbf{10}$ms)
- Some processes, called *real-time*, may not be preempted



Process Scheduling

Voluntary Yield

- Suspend execution and yield to the kernel
  - E.g., I/O or synchronization
  - Only way to enable a context switch for *real-time* processes



5. Processes and Memory Management

System Call: fork()

Create a Child Process

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork();
```

Semantics

- The *child* process is identical to its *parent*, except:
  - Its PID and PPID (parent process ID)
  - Zero resource utilization (initially, relying on copy-on-write)
  - No pending signals, file locks, inter-process communication objects
- *On success, returns the child PID in the parent, and* **0** *in the child*
  - Simple way to detect "from the inside" which of the child or parent runs
  - See also `getpid()`, `getppid()`
- Return -1 on error
- Linux: `clone()` is more general, for both *process* and *thread* creation

System Call: fork()

Create a Child Process

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork();
```

Typical Usage

```
switch (cpid = fork()) {
  case -1:                         // Error
    perror("‘my_function’: ‘fork()’ failed");
    exit(1);
  case 0:                          // The child executes
    continue_child();
    break;
  default:                         // The parent executes
    continue_parent(cpid);  // Pass child PID for future reference
}
```

System Call: `execve()` and variants

Execute a Program

```
#include <unistd.h>

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

Semantics

- Arguments: absolute path, argument array (a.k.a. vector), environment array (shell environment variables)
- On success, the call *does not return*!
  - Overwrites the process's *text*, *data*, *bss*, *stack* segments with those of the loaded program
  - Preserve PID, PPID, open file descriptors
    - Except if made `FD_CLOEXEC` with `fcntl()`
  - If the file has an SUID (resp. SGID) bit, set the *effective* UID (resp. GID) of the process to the file's *owner* (resp. group)
  - Return **−1** on error

System Call: `execve()` and variants

Execute a Program

```
#include <unistd.h>

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

Error Conditions

- Typical `errno` codes

  `EACCES`: execute permission denied (among other explanations)

  `ENOEXEC`: non-executable format, or executable file for the wrong OS or processor architecture

System Call: execve() and variants

Execute a Program: Variants

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execle(const char *path, const char * arg, ..., char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Arguments

- `execl()` operates on `NULL`-terminated argument list
  Warning: `arg`, the *first argument* after the pathname/filename corresponds to `argv[0]` (the program name)
- `execv()` operates on argument array
- `execlp()` and `execvp()` are `$PATH`-relative variants (if `file` does not contain a '/' character)
- `execle()` also provides an environment

System Call: execve() and variants

Execute a Program: Variants

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execle(const char *path, const char * arg, ..., char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Environment

- Note about environment variables
  - They may be manipulated through getenv() and setenv()
  - To retrieve the whole array, declare the global variable
    `extern char **environ;`
    and use it as argument of execve() or execle()
  - More information: `$ man 7 environ`

I/O System Call: fcntl()

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Some More Commands

F_GETFD: get the file descriptor flags

F_SETFD: set the file descriptor flags to the value of arg
Only FD_CLOEXEC is defined: sets the file descriptor to be closed upon calls to execve() (typically a security measure)

I/O System Call: fcntl()

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Return Value

- On success, `fcntl()` returns a (non-negative) value which depends on the command
  - `F_GETFD`: the descriptor's flags
  - `F_GETFD`: **0**
- Return **−1** on error

System Call: _exit()

Terminate the Current Process

```
#include <unistd.h>

void _exit(int status);
```

Purpose

- Terminates the calling process
  - Closes any open file descriptor
  - Frees all memory pages of the process address space (except shared ones)
  - Any child processes are inherited by process **1** (init)
  - The parent process is sent a SIGCHLD signal (ignored by default)
  - If the process is a *session leader* and its *controlling terminal* also controls the session, disassociate the terminal from the session and send a SIGHUP signal to all processes in the *foreground group* (terminate process by default)
- The call never fails and *does not return*!

System Call: _exit()

Terminate the Current Process

```
#include <unistd.h>

void _exit(int status);
```

Exit Code

- The *exit code* is a *signed byte* defined as `(status & 0xff)`
- **0** means normal termination, non-zero indicates an error/warning
- There is no standard list of exit codes
- It is collected with one of the `wait()` system calls

System Call: _exit()

C Library Front-End: exit()

```c
#include <stdlib.h>

void exit(int status);
```

- Calls any function registered through atexit() (in reverse order of registration)
- Use this function rather than the low-level _exit() system call

5. Processes and Memory Management

Bootstrap and Processes Genealogy

Swapper Process

Process 0

- *One per CPU* (if multiprocessor)
- Built from scratch by the kernel and runs in kernel mode
- Uses *statically*-allocated data
- Constructs memory structures and initializes virtual memory
- Initializes the main kernel data structures
- Creates kernel threads (swap, kernel logging, etc.)
- Enables interrupts, and creates a kernel thread with $\mathrm{PID} = \mathbf{1}$

Bootstrap and Processes Genealogy

Init Process

Process 1

- *One per machine* (if multiprocessor)
- Shares all its data with process **0**
- Completes the initalization of the kernel
- Switch to user mode
- Executes `/sbin/init`, becoming a regular process and burying the structures and address space of process **0**

Executing `/sbin/init`

- Builds the OS environment
  - From `/etc/inittab`: type of bootstrap sequence, control terminals
  - From `/etc/rc*.d`: scripts to run system *daemon*s
- Adopts all orphaned processes, continuously, until the system halts
- `$ man init` and `$ man shutdown`

Process Tree

Simplified Tree From `$ pstree | more`

```
init-cron
    |-dhclient3
    |-gdm---gdm-+-Xorg
    |           `-x-session-manag---ssh-agent
    |-5*[getty]
    |-gnome-terminal-+-bash-+-more
    |                |      `-pstree
    |                |-gnome-pty-helper
    |                `-{gnome-terminal}
    |-klogd
    |-ksoftirqd
    |-kthread-+-ata
    |         |-2*[kjournald]
    |         `-kswapd
    |-syslogd
    `-udevd
```

5. Processes and Memory Management

Example: Network Service Daemons

Internet "Super-Server"

- inetd, initiated at boot time
- Listen on specific ports — listed in /etc/services
  - Each configuration line follows the format:
    service_name port/protocol [aliases ...]
    E.g., `ftp 21/tcp`
- Dispatch the work to predefined daemons — see /etc/inetd.conf — when receiving incomming connections on those ports
  - Each configuration line follows the format:
    service_name socket_type protocol flags user_name daemon_path arguments
    E.g., `ftp stream tcp nowait root /usr/bin/ftpd`

Process Sessions and Groups

Process Sessions

- Orthogonal to process hierarchy
- Session ID = PID of the leader of the session
- Typically associated to user *login*, interactive *terminals*, *daemon* processes
- The *session leader* sends the `SIGHUP` (*hang up*) signal to every process belonging to its session, and only if it belongs to the *foreground* group associated to the *controlling terminal* of the session

Process Groups

- Orthogonal to process hierarchy
- Process Group ID = PID of the group leader
- General mechanism
  - To distribute signals among processes upon global events (like `SIGHUP`)
  - Interaction with terminals, e.g., stall background process writing to terminal
  - To implement *job control* in shells
    `$ program &`, **Ctrl-Z**, `fg`, `bg`, `jobs`, `%1`, `disown`, etc.

System Call: setsid()

Creating a New Session and Process Group

```
#include <unistd.h>

pid_t setsid();
```

Description

- If the calling process is not a process group leader
  - Calling process is the leader and only process of a new group and session
  - Process group ID and session ID of the calling process are set to the PID of the calling process
  - Calling process has no controlling terminal any more
  - Return the session ID of the calling process (its PID)
- If the calling process is a process group leader
  - Return -1 and sets `errno` to `EPERM`
  - Rationale: a process group leader cannot "resign" its responsibilities

System Call: `setsid()`

Creating a Daemon (or Service) Process

- A *daemon process* is detached from any terminal, session or process group, is adopted by `init`, has no open standard input/output/error, has / for current directory
- "Daemonization" procedure
  1. Call `signal(SIGHUP, SIG_IGN)` to ignore `HUP` signal (see signals chapter)
  2. Call `fork()` in a process **P**
  3. Terminate parent **P**, calling `exit()` (may send `HUP` to child if session leader)
  4. Call `setsid()` in child **C**
  5. Call `signal(SIGHUP, SIG_DFL)` to reset `HUP` handler (see signals chapter)
  6. Change current directory, close descriptors 0, 1, 2, reset `umask`, etc.
  7. Continue execution in child **C**
- Note: an alternative procedure with a double `fork()` and `wait()` in the grand-parent is possible, avoiding to ignore the `HUP` signal

System Call: `setsid()`

Creating a Daemon (or Service) Process

- A *daemon process* is detached from any terminal, session or process group, is adopted by `init`, has no open standard input/output/error, has / for current directory
- "Daemonization" procedure
  1. Call `signal(SIGHUP, SIG_IGN)` to ignore `HUP` signal (see signals chapter)
  2. Call `fork()` in a process **P**
  3. Terminate parent **P**, calling `exit()` (may send `HUP` to child if session leader)
  4. Call `setsid()` in child **C**
  5. Call `signal(SIGHUP, SIG_DFL)` to reset `HUP` handler (see signals chapter)
  6. Change current directory, close descriptors 0, 1, 2, reset `umask`, etc.
  7. Continue execution in child **C**
- Note: an alternative procedure with a double `fork()` and `wait()` in the grand-parent is possible, avoiding to ignore the `HUP` signal

See, `getsid()`, `tcgetsid()`, `setpgid()`, etc.
See also `daemon()`, not POSIX but convenient integrated solution