5. Processes and Memory Management

Outline

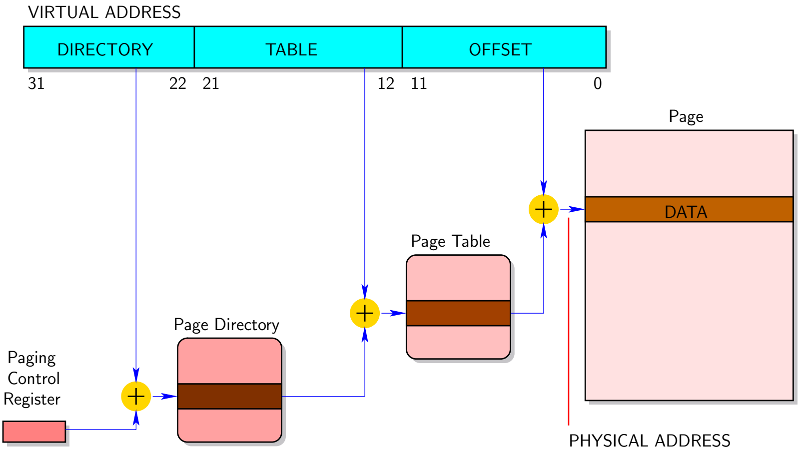
- Introduction to memory management
- Principles
  - Logical separation
  - Process states
  - Scheduling
- Programmer interface
  - Main system calls
  - Examples

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

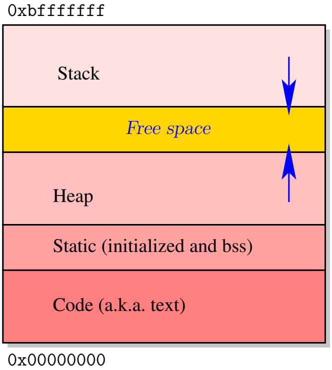
Introduction to Memory Management



Introduction to Memory Management

Per-Process Virtual Memory Layout

- *Code* (also called *text*) segment
- *Static Data* segments
  - Initialized global (and C `static`) variables
  - Uninitialized global variables (zeroed when initializing the process, also called *bss*)
- *Stack* segment: function calls, local variables (also called `auto`matic in C)
- *Heap* segment (`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[0x2000000];

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

Sample Output

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

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

double t[0x2000000];

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);
}
```

Introduction to Memory Management

Lazy Memory Management Principles

- 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
  - Faster and more memory-economical (same principle as *overbooking*) than eager page allocation when a process requests an interval of memory addresses (`malloc()`)

Introduction to Memory Management

Lazy Memory Management Principles

- 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
  - Faster and more memory-economical (same principle as *overbooking*) than eager page allocation when a process requests an interval of memory addresses (`malloc()`)
- Motivation: high-performance process creation
  - *Copy-on-write*: when cloning a process, do not replicate its memory, but mark its pages as "needing to be copied on the next write access"
  - Critical for UNIX, where cloning is the only way to create a new process, knowing that child processes are often short-lived (they terminate or become overlapped by the execution of another program through `execve()`)

C Library Function: malloc()

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to the allocated interval of size bytes of memory
- Returns NULL on error

C Library Function: `malloc()`

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to the allocated interval of `size` bytes of memory
- Returns `NULL` on error
- Note: beyond *demand-paging*, 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 (*overbooking*)
  - 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)

C Library Function: `malloc()`

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to the allocated interval of `size` bytes of memory
- Returns `NULL` on error
- Note: beyond *demand-paging*, 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 (*overbooking*)
  - 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)
- See also `calloc()` and `realloc()`

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`
- You may use the powerful `valgrind` tool to debug dynamic memory management (memory leaks, corrupt calls to `free()`)

Process Tree

- `init` process (a.k.a. process **1**)
- Process uniquely identified with PID

- 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

Process Tree

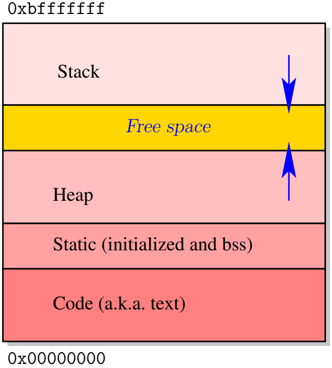
- init process (a.k.a. process **1**)
- Process uniquely identified with PID

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
```

Logical Separation: User Address Space

- *User* address space for the process
  - *Code* (also called *text*) segment
  - *Static Data* segments
    - Initialized global variables
    - Uninitialized global variables (zeroed when initializing the process)
  - *Stack* segment: function calls, local variables (also called `auto`matic in C)
  - *Heap* segment (`malloc()`)
- Code and data segments are extracted from the executed program
  - ELF format for object (`.o` and executable) files
  - Through the `execve()` system call



Logical Separation: Kernel Address Space

- *Kernel* address space for the process
  - Process *descriptor*
    - Repository for all process-related information
      (memory mapping, open file descriptors, current directory, etc.)
    - Link to kernel stack
  - Kernel stack
    (one memory page in general, may grow in extreme cases of nested interrupts/exceptions)
- Process table
  - Hash table of PID-indexed process descriptors
  - Doubly-linked tree (links to both children and parent)

Process Descriptor

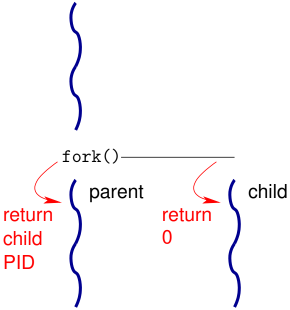
Main Fields of the Descriptor

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 control thread information
Files	current directory and table of file descriptors
Limits	resource limits, see `getrlimit()`
Signals	signal handlers, masked and pending signals

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



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
  - Simple way to detect "from the inside" which of the child or parent runs
  - See also `getpid()`, `getppid()`
- Returns $-\mathbf{1}$ on error
- More general (Linux-specific) system call: `clone()`
  Primitive call 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*!
  - It overrites the *text*, *data*, *bss* and *stack* segments of the process with those of the program loaded
  - Preserve PID, PPID, open file descriptors
    Except if maked `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)
  - Returns -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

```c
#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

```c
#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/F_SETFD: get and return / 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: $\mathbf{0}$

- Returns $-\mathbf{1}$ 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* (ignored 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()

```
#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

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 *daemons*
- Adopts all orphaned processes, continuously, until the system halts
- `man init` and `man shutdown`

Sessions and Process 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 beloning 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`)
  - To arbitrate read/write requests on terminals, e.g., stalling background processes performing terminal I/O
  - 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
  - The calling process is the leader of the new session and the process group leader of the new process group
  - The process group ID and session ID of the calling process are set to the PID of the calling process
  - The calling process will be the only process in this new process group and in this new session
  - It has *no controlling* `tty`
  - Returns the session ID of the calling process (its PID)
- If the calling process is a process group leader
  - Returns -1 and sets `errno` to `EPERM`
  - Rationale: a process group leader cannot "resign" its responsibilities

System Call: `setsid()`

Creating a New Session and Process Group

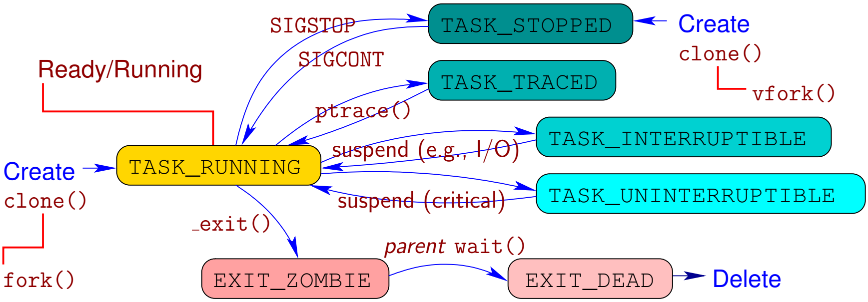
```
#include <unistd.h>

pid_t setsid();
```

Creating a Daemon Process

- A *daemon process* (also called *service* process) is detached from any terminal, session or process group
- "Daemonization" procedure
  1. Call `fork()` in a process **P**
  2. Terminate parent **P** (calling `exit()`)
  3. Call `setsid()` in child **C**
  4. Call `fork()` again in child **C**
  5. Terminate process **C**
  6. Continue execution in *grandchild* process

See, `getsid()`, `tcgetsid()`, `setpgid()`, etc.

Process States (Linux)



- Under Linux, context switch does *not* change process state
- In other OSes, the TASK_RUNNING state is usually split into

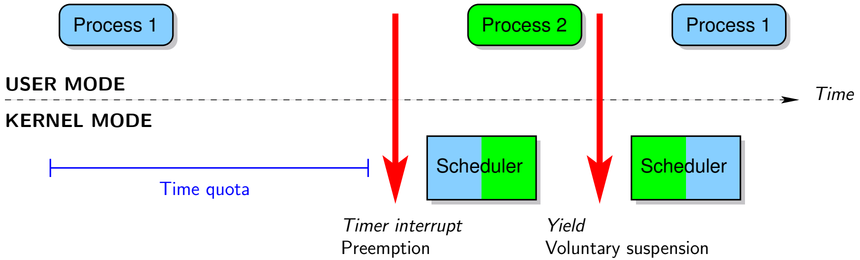
  ready: runnable processes, i.e., waiting to be scheduled

  running: making progress on a processor or hardware thread

Process Scheduling

Preemption

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



Process Scheduling

Preemption

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

A Few Words About Real-Time OSes

- Beyond preemption control, real-time OSes offer deadline and throughput guarantees, reactivity, liveness, etc.
- Real-time scheduling requires static information about processes (e.g., bounds on execution time) and may not be compatible with many services provided by a general-purpose OSes
- Modern OSes tend to include more and more real-time features, largely for media-processing or high-throughput computing (network routing, data bases and web services)

Process Scheduling

Distribute Computations Among Running Processes

- Infamous optimization problem
- Many heuristics... and objective functions
  - Throughput?
  - Reactivity?
  - Deadline satisfaction?
- General answer (or failure to answer): *priorities*
  - `nice()` system call
  - `nice` and `renice` commands
- *Anticipatory scheduler* heuristic (prediction and adaptation)
- Multiple scheduling queues
  - Split processes according to scheduling semantics (e.g., preemptive or not)
  - Performance: priority queues have high complexity