Virtual Memory
(advanced usage)

- `fork()` and `exec()`
- Swapfiles & overcommitting
- Zero pages
- `mmap()` (files)
- `MAP_ANONYMOUS`
- `MAP_SHARED`

`fork()` and `exec()`

- How do we create a new process in UNIX?
- And how does this relate to virtual memory?

- When we create a process, there are **many** different options we might imagine.

Class Exercise
Take 2 minutes with a neighbor – how many different options & variants can you think of?

`fork()` and `exec()`

- This is how ugly things are in Windows:

  https://learn.microsoft.com/en-us/windows/win32/procthread/creating-processes

- UNIX's solution:
  - **`fork()`** duplicates the current process
  - Call once, return inside **two different processes!**
  - Then, change whatever you want
  - Finally, call `exec()`

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

int main() {
  printf("BEFORE: pid %d\n", getpid());

  int child_pid = fork();

  printf("AFTER: pid %d ppid %d "
         "child_pid %d\n",
         getpid(), getppid(), child_pid);

  return 0;
}
```

Class Exercise

Run this on your own computer. What does `fork()` return in the parent and child processes?

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

int main() {
  printf("BEFORE: pid %d\n", getpid());

  int child_pid = fork();

  printf("AFTER: pid %d ppid %d "
         "child_pid %d\n",
         getpid(), getppid(), child_pid);

  return 0;
}
```

Question

Where is `child_pid` stored?

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

int main() {
  int rc = fork();

  if (rc == 0) {
    ... make some changes ...
    exec(...);
  } else {
    wait(...);
  }

  return 0;
}
```

NOTE:
`exec` replaces a process with another process. Complete memory replacement!

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

int main() {
  int rc = fork();

  if (rc == 0) {
    ... make some changes ...
    exec(...);
  } else {
    wait(...);
  }

  return 0;
}
```

NOTE:
Often, a child process only lasts for a short time before `exec()` replaces it.

`fork()` and `exec()`

- Child processes get a **complete duplicate** of the virtual memory space
- But they often don't last very long, before all of the memory is replaced by `exec()`

Q: How do we do this without having a huge memory hit?

A: COW! (copy-on-write)

Swap Files and Overcommitting

- **Overcommiting** memory means allowing programs to allocate more memory than you actually have
  - Most programs don’t use all of the memory they ask for
  - Even if they use it, they don’t touch it often

Swap Files and Overcommitting

- A **swap file/swap disk** is space on disk which can be used to “spill” pages from memory
  - Write a page to disk
  - Disable all page table entries that used it
  - Leave on disk until the page is touched again

Zero Pages

- We often allocate new virtual memory pages, which are “empty”
  - Need to allocate physical pages
  - Don’t want to leak old data – security risk!

- Sometimes, processes read but don’t write some pages
  - Especially in large allocations

Zero Pages

- The OS typically keeps a single "**zero page**" in memory
  - Physical page, not virtual
  - Any number of maps to it (many!)
  - Users get COW copies
  - One process can have *many* such maps (one for each unmodified page)

Virtual Memory
(advanced usage)

`mmap()`

`mmap()`

- `mmap()` is a UNIX system call that adds new virtual memory pages
  - For now: think of mapping a file
  - Later, we'll discuss "anonymous" memory

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

The `open()` syscall returns an integer "file descriptor." We'll need this.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

We don't care what address we get, but we want exactly one page of memory.

We want a readonly page.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

We want a private copy (COW) of the page. No links to others.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

This is the file we want to map. We want to map starting at byte 0 in the file.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

`mmap()` returns a pointer to the newly-allocated virtual memory page.

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

int main() {
  int fd = open("/bin/bash", O_RDONLY);
  printf("fd %d\n", fd);

  char *buf = mmap(NULL, 4096,
                   PROT_READ,
                   MAP_PRIVATE,
                   fd, 0);

  printf("The first 8 bytes are:\n");
  for (int i=0; i<8; i++)
    printf("  %d: 0x%02x\n", i, buf[i]);

  return 0;
}
```

You can directly access this memory, just like any `malloc()` buffer.

`mmap()`

- Why use `mmap()` instead of `malloc()` and `read()` ?
  - Faster (especially if the file is in cache)
  - Less memory duplication
  - Mapping can be read-only to the process
  - If not in use, OS can easily drop without needing swap memory
  - Shared memory is possible (see below)

MAP_ANONYMOUS

- The `MAP_ANONYMOUS` flag creates zero pages (normally, writable). The `fd` and `offset` args are ignored.

```
char *buf = mmap(NULL, 4096,
                 PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,
                 -1, 0);
```

MAP_SHARED

- The `MAP_SHARED` indicates that this process wants to share the page with other maps of the page, in this process or others. **Not** COW.
  - If writable, then writing to the page (eventually) changes the file **DANGER!**
  - Changes made by other procs will be visible here

```
char *buf = mmap(NULL, 4096,
                 PROT_READ | PROT_WRITE,
                 MAP_SHARED,
                 fd, 0);
```