Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 16: Managing multiple address spaces

Prof. Ryan Huang

Physical memory allocation

How to divide physical memory among processes?

- Fairness vs. efficiency

Variable space

- A process' set of pages grows and shrinks dynamically
- Global replacement: can evict pages from any process
  - One process can ruin it for the rest

Fixed space

- Each process gets a fixed number of physical pages
- Local replacement: can only evict a process' own pages
  - Some processes may do well while others suffer

Virtual memory performance

Physical memory is a cache for address space

- Cache miss is a page fault
- Swap page from disk to memory, slow (disk I/O)

Performance degrades rapidly as miss rate goes up

- Avg access time = hit rate * hit time + miss rate * miss time
- E.g., hit time = .0001 ms; miss time = 10 ms
  - Average access time (100% hit rate) = .0001 ms
  - Average access time (99% hit rate) = .100099 ms
  - Average access time (90% hit rate) = 1.0009 ms

High miss rate → thrashing

- OS spent most time in paging data back and forth from disk
- Little time spent doing useful work (making progress)

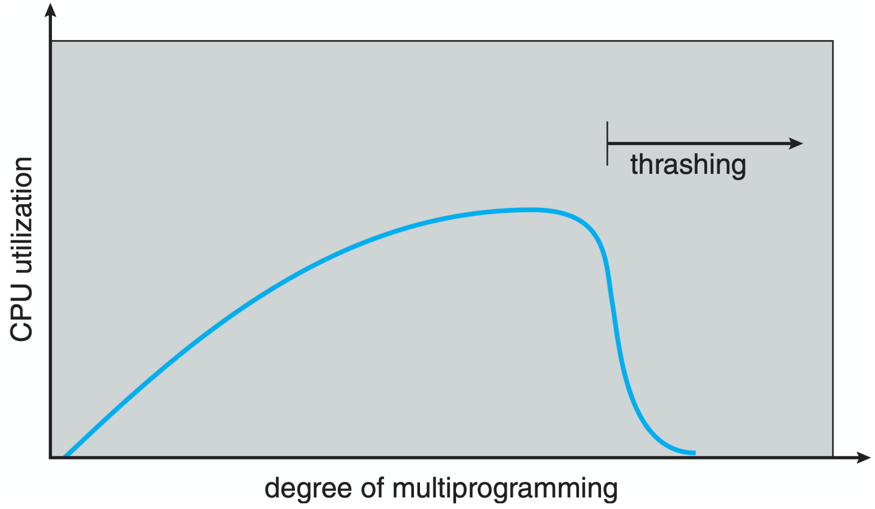
Reasons for thrashing

The system is overcommitted

- No idea which pages should be in memory to reduce faults

Common reasons

- Access pattern has no locality
- Each process fits individually, but too many processes



Solutions to thrashing?

Buy more DRAM!

Change scheduler to reduce thrashing

- E.g., system has 1 GB memory
- How to run processes A, B, C, D (each 500 MB)?

- Why does this help?

Working set

Thrashing depends on portion of address space being actively used by each process

What do we mean by “actively using”?

A working set of a process models the dynamic locality of its memory usage

- $WS(t,w)$= {P | Page P was referenced in the time interval $(t-w,t)$ }
  - w – working set window (in absolute time or units of mem references)

WS size: # of unique pages referenced in (t−w, t)

- The working set size changes with program locality
- During periods of poor locality, the working set size becomes larger

Working set (cont'd)

Intuition: want the working set to be in memory to prevent heavy faulting

- *Expect* to run for *w* time without getting a fault
- Sum of all working sets should fit in memory
- Only run subset of processes that fit in memory

Difficult questions

- How do we determine *w*?
- How to handle changes in working set?

Not a practical model, but the intuition is still valid

- When people ask, "*How much memory does Firefox need?*", they are in effect asking for the size of Firefox's working set

Creating a process

A process is created by another process

- Parent is creator, child is created (Unix: ps "PPID" field)
- What about the first process?
  - Unix: init process (PID 1), ancestors of all other processes

OS provides process creation system calls

After creating a child process

- the parent process may either wait for the child to finish its task or continue in parallel

Process creation: Unix

In Unix, processes are created using fork()

```
int fork()
```

1. Creates and initializes a new Process Control Block (PCB)
2. Creates a new address space
3. Initializes the address space with a **copy** of the address space of the parent
4. Initializes the kernel resources to point to the parent's resources (e.g., open files)
5. Places the PCB on the ready queue

`fork()` returns twice

- Huh?
- Returns the child's PID to the parent, "0" to the child

Fork() example

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

```
$ gcc -o test test.c

$ ./test
```

My child is 87631

Child of ./test is 87631

Duplicating address spaces

child_pid = 87631

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf("Child...");
} else {
  printf("Parent...");
}
```

Parent process

child_pid = 0

PC

```
child_pid = fork();
if (child_pid == 0) {
  printf("Child...");
} else {
  printf("Parent...");
}
```

Child process

Divergence

child_pid = 87631

```
child_pid = fork();
if (child_pid == 0) {
  printf("Child...");
} else {
  printf("Parent...");
}
```

PC

Parent process

child_pid = 0

```
child_pid = fork();
if (child_pid == 0) {
  printf("Child...");
} else {
  printf("Parent...");
}
```

PC

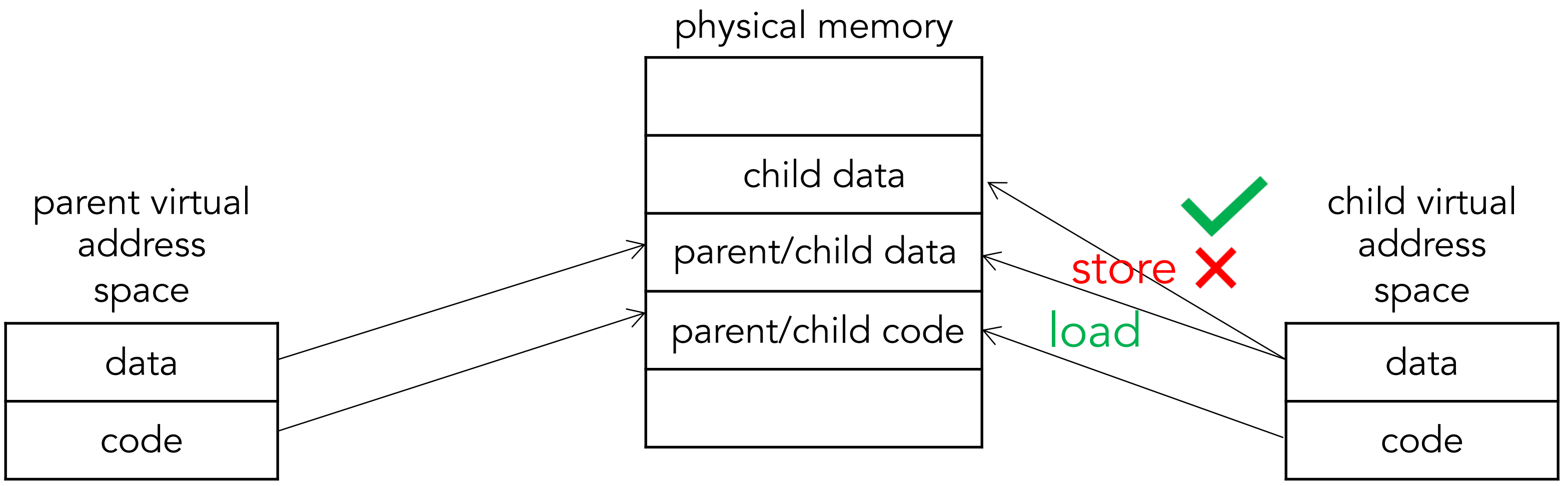
Child process

Avoiding work on fork()

Copying entire address space is expensive

Instead, make it look like it's copied, but do the work later

This is called copy-on-write



What if parent process does a store to data?

Process creation: Unix (2)

How do we actually start a new program?

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

`execv()`

1. Stops the current process
2. Replace the process' address space with "`prog`"
3. Initializes hardware context and args for the new program
4. Places the PCB onto the ready queue

Note: `exec` does *not* create a new process

What does it mean for `exec` to return?

Why `fork()`?

Most calls to `fork` followed by `exec`

- could also combine into one `spawn` system call

Very useful when the child…

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

Example: web server

Example: shell

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

Why `fork()`?

Most calls to `fork` followed by `exec`

- could also combine into one `spawn` system call

Very useful when the child…

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task

Real win is simplicity of interface

- Tons of things you might want to do to child process:
  - manipulate file descriptors, set environment variables, reduce privileges, …
- Yet **`fork()`** requires no arguments at all

Process creation: Windows

Without `fork`, needs tons of different options for creating a new process

Example: Windows `CreateProcess` system call

- Combines both `fork()` and `exec()`

```
BOOL WINAPI CreateProcess(
  _In_opt_ LPCTSTR lpApplicationName,
  _Inout_opt_ LPTSTR lpCommandLine,
  _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_ BOOL bInheritHandles,
  _In_ DWORD dwCreationFlags,
  _In_opt_ LPVOID lpEnvironment,
  _In_opt_ LPCTSTR lpCurrentDirectory,
  _In_ LPSTARTUPINFO lpStartupInfo,
  _Out_ LPPROCESS_INFORMATION lpProcessInformation
);
```

- Also `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`, …