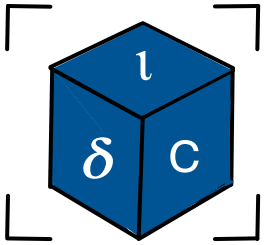
Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi



File: dup

```
#include <unistd.h>
int dup(int oldfd)
```

- Takes *oldfd*, an open file descriptor, and returns a new descriptor that refers to the same open file description.
- The new descriptor is guaranteed to be the lowest unused file descriptor.
- `Returns`

  (new) file descriptor on success, or –1 on error
- newfd = dup(1);

Example

```
main() {
  int fd1, fd2;
  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);
}
```

Example

```
main() {
  int fd1, fd2;

  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);

  write(fd1, "The Brown Dog\n", strlen("The Brown Dog\n"));
  write(fd2, "Jumped over the moon\n", strlen("Jumped over the moon\n"));

  close(fd1);
  close(fd2);
}
```

Example

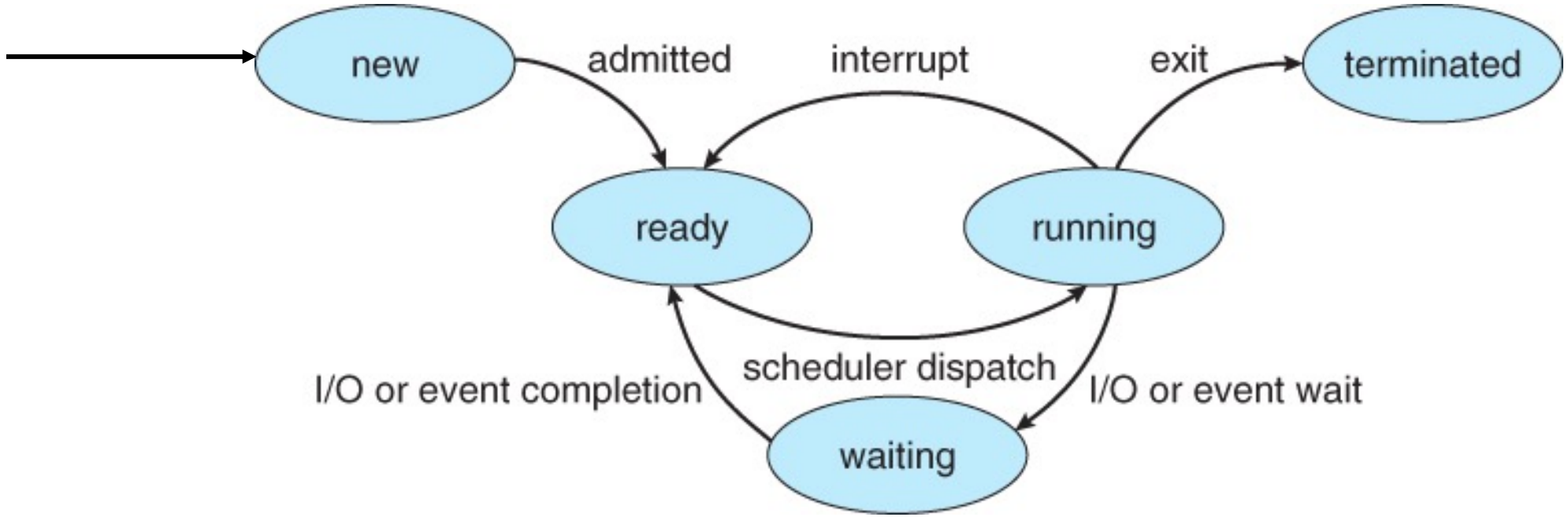
```
#include <fcntl.h>
#include <stdio.h>
main() {
  int fd1, fd2;

  fd1 = open("file2", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = dup(fd1);

  write(fd1, "The Brown Dog\n", strlen("The Brown Dog\n"));
  write(fd2, "Jumped over the moon\n", strlen("Jumped over the moon\n"));

  close(fd1);
  close(fd2);
}
```

Kernel Process States



- Involuntary termination from ready and wait state.

Parent-child state

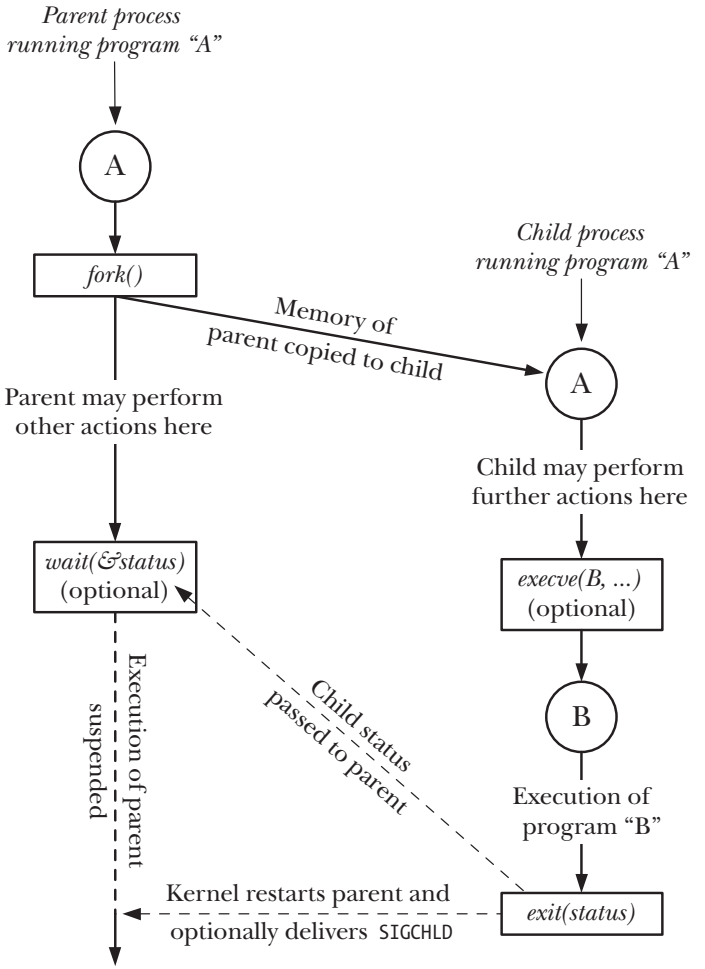


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

Process Creation/ Coordination

- fork()
  - Create a child process
  - Identical to parent EXCEPT for return value of fork() call
  - Determines child/parent
- getpid() / getppid()
  - Get process ID of the currently running process
  - Get parent process ID
- exec() family
  - Replace currently running process with a different image
  - Process becomes something else losing previous code
  - Focus on execvp()
- wait() / waitpid()
  - Wait for any child to finish (wait)
  - Wait for a specific child to finish (waitpid)
  - Get return status of child

Listing

- child_fork.c

Creating a Process: fork()

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

- Create a child process
  - The child is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the **fork()**
- Returns:
  - To child: 0 on success
  - To parent: process ID of the child process or -1 on error, sets **errno**

What makes up a process?

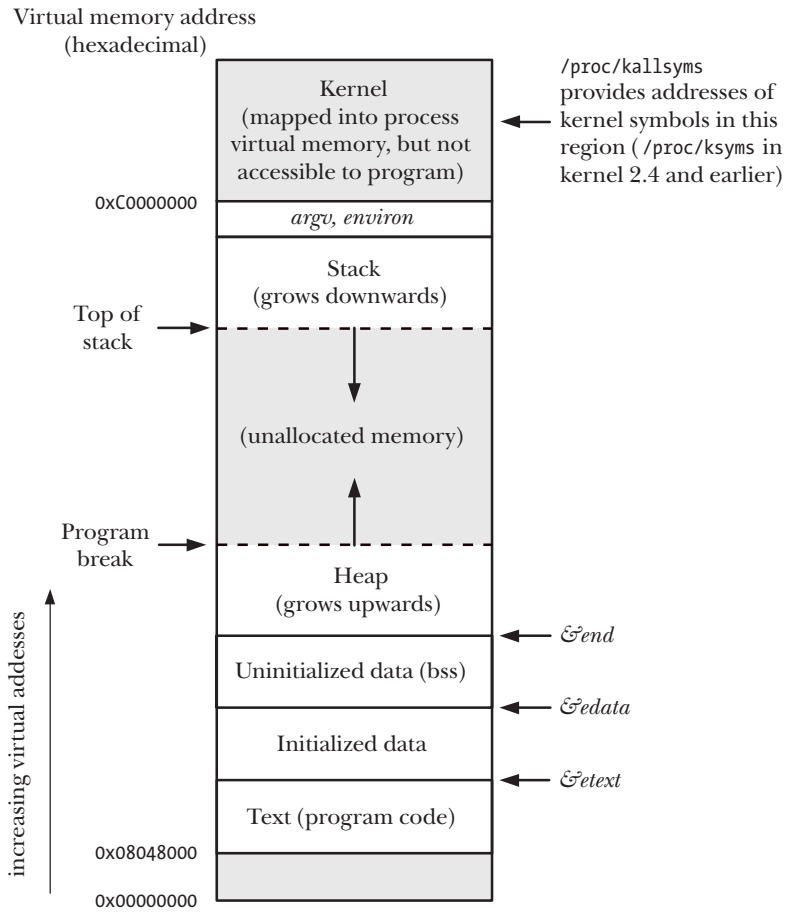
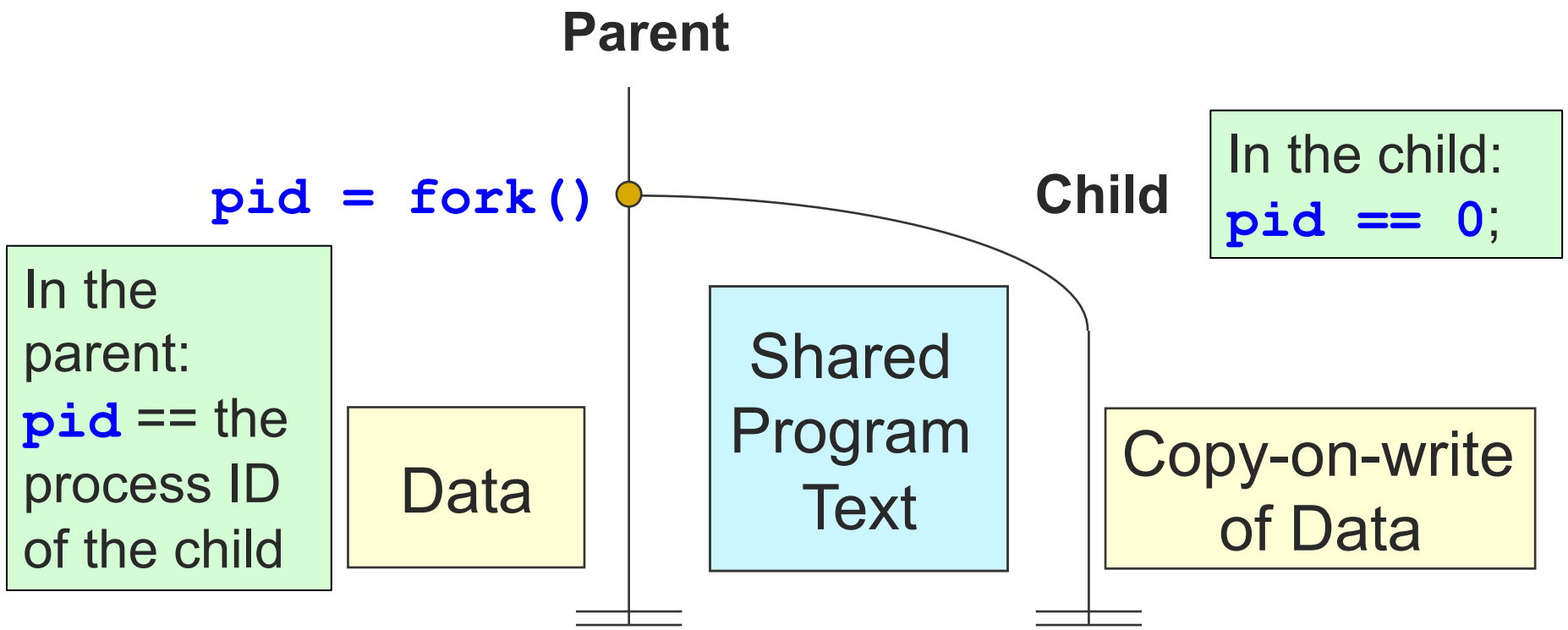


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Creating a process

- A program can use this **pid** difference to do different things in the parent and child



Fork()

- Only system call which returns two values.

```
pid_t pid = fork();
if (pid == 0) {
printf("hello from child\n"); }
else {
printf("hello from parent\n"); }
```

Fork()

```
printf("I'm printed once!\n");
fork();
printf("I'm printed twice!\n");
```

Fork issues

- Determining PIDs
  - A parent can only determine the PID of the child through a fork(). A child can always determine the PID through getppid() call.
- Which first?
  - Implementation of fork is not standard across kernels.
  - Child Vs parent scheduling
- Output of fork remains indeterminate
  - Switching between parent and child depends on many factors
    - Machine load, OS CPU scheduler
    - Output interleaving is nondeterministic; Cannot determine output by looking at code

Fork

```
int main() {
fork()
fork()
fork()
return 0;
}
```

Chain and Fan

- Write code to make chain

- Write code to make fan
  - Code to make N children of one parent process

Chain and Fan

- Write code to make chain

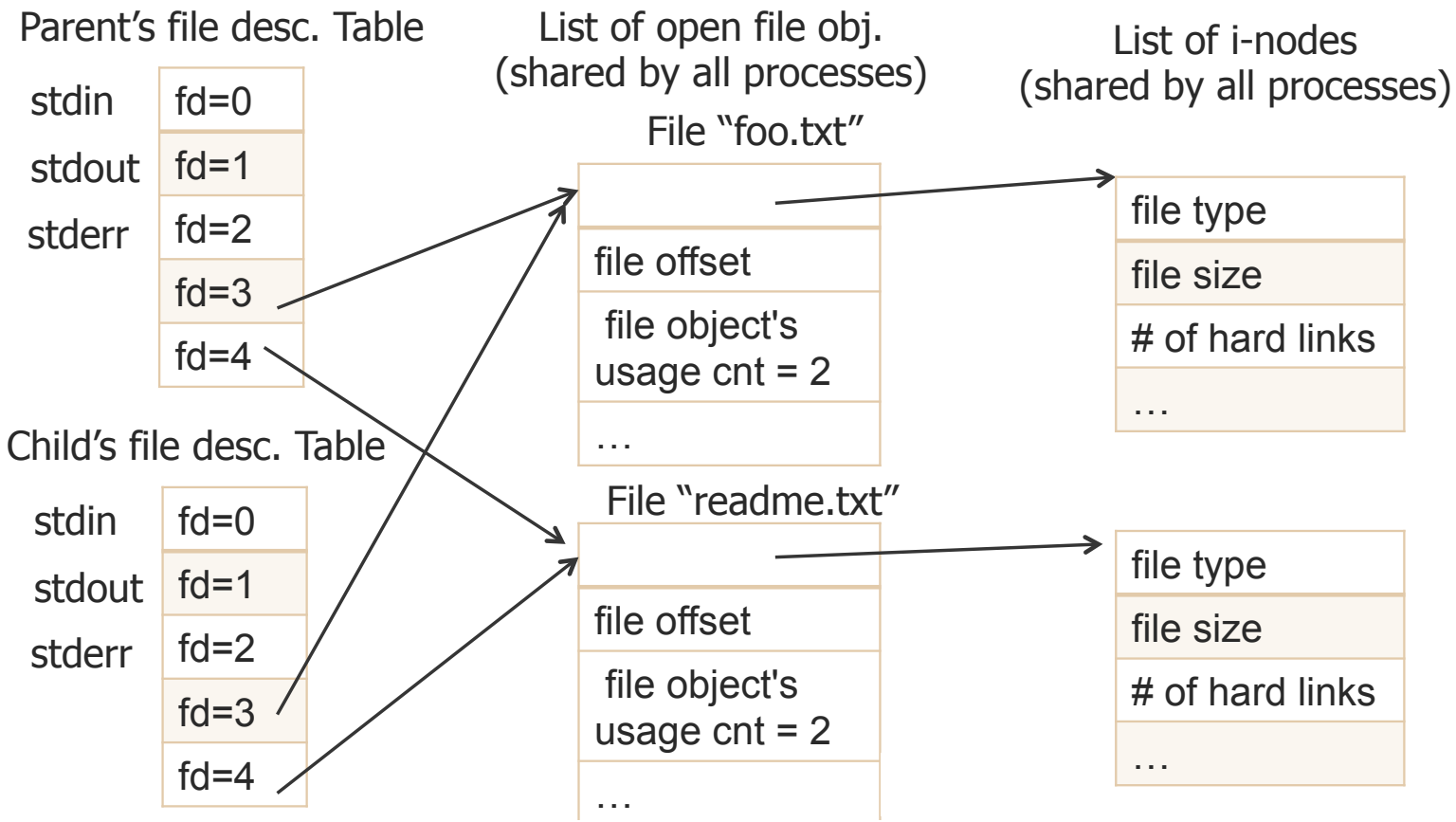
```
pid_t childpid;
for (i=1;i<n;i++)
        if (childpid = fork())  // child keeps forking
                break;
```

- Write code to make fan
  - Code to make N children of one parent process

```
pid_t childpid;
for (i=1;i<n;i++)
    if ((childpid = fork()) <= 0) // parent keeps forking
    break;
```

fork and dup

- When fork() is called, all file descriptors are duplicated as if dup() is called.



Example

```
#include <fcntl.h>
#include <stdio.h>
main() {
  char s[1000];
  int i, fd;
  fd = open("file3", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  i = fork();
  sprintf(s, "fork() = %d.\n", i);
  write(fd, s, strlen(s));
  close (fd);
}
```

Fork and memory

- Conceptually, fork() creates copies of the parent's text, data, heap, and stack segments.
- In practice, this is wasteful copying if the new child's program text is replaced
- The kernel employs a technique known as copy-on-write.
  - After the fork(), the kernel traps any attempts by either the parent or the child to modify one of these pages, and makes a duplicate copy of the about-to-be-modified page.

Copy-on-write

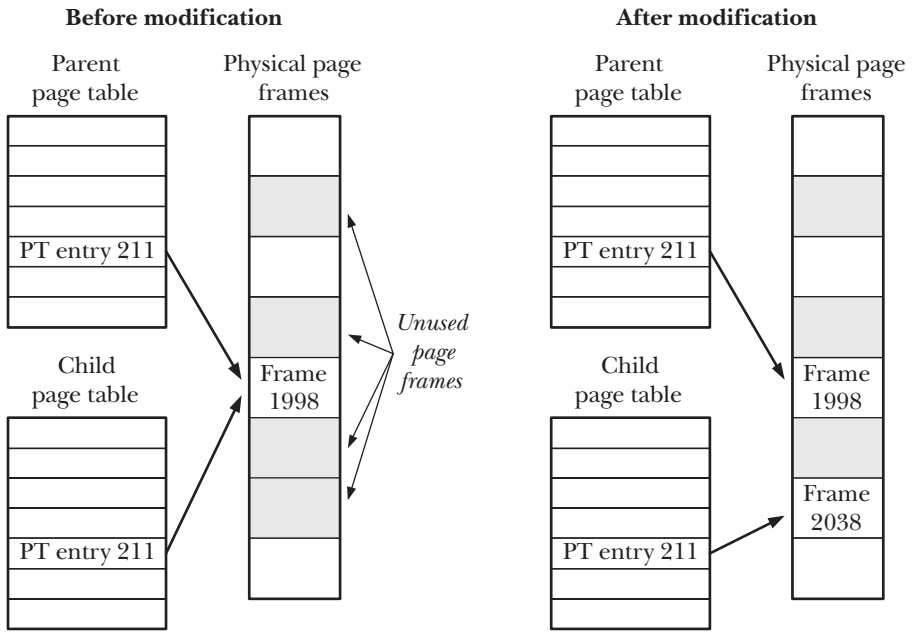
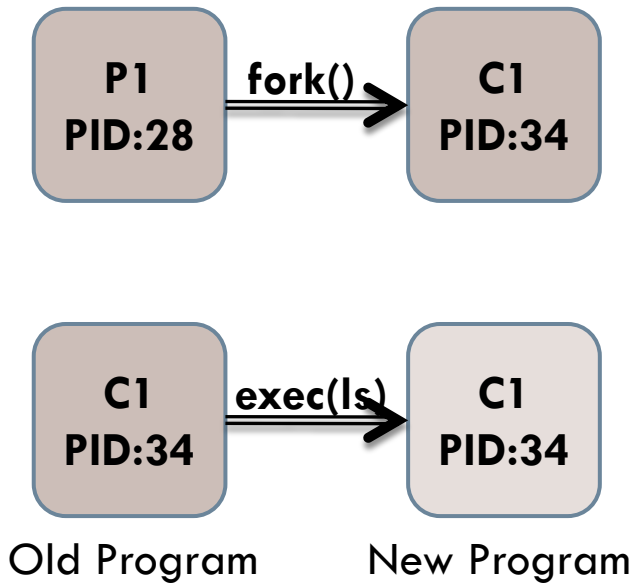


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

Process Creation/ Coordination

- fork()
  - Create a child process
  - Identical to parent EXCEPT for return value of fork() call
  - Determines child/parent
- getpid() / getppid()
  - Get process ID of the currently running process
  - Get parent process ID
- exec() family
  - Replace currently running process with a different image
  - Process becomes something else losing previous code
  - Focus on execvp()
- wait() / waitpid()
  - Wait for any child to finish (wait)
  - Wait for a specific child to finish (waitpid)
  - Get return status of child

Load a new program into the child---exec()



Exec*

- e – An array of pointers to environment variables is explicitly passed to the new process image.
- l – Command-line arguments are passed individually (a list) to the function.
- p – Uses the PATH environment variable to find the file named in the file argument to be executed.
- v – Command-line arguments are passed to the function as an array (vector) of pointers.

execv(): Loading and running programs

- **int execv(char *filename, char *argv[])**
- Transforms the calling process into a new process
  - Runs executable **filename**
  - With argument list **argv**
- Does not return(unless error)
- Overwrites code, data, and stack
  - keeps pid, open files and signal context
- Parameters:
  - argv is a pointer to the argument list tobe made available to the new process
- To pass arguments and environment, use:
  - **int execve(char *filename, char *argv[], char *envp[])**

Example

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
int fd;
fd = open(argv[1],O_RDWR|O_CREAT,S_IRWXU); //create an output file
dup2(fd, 1);                                //redirect output to file
close(fd);                                  //free unused file descriptor
char* array[] = {"ls", "-la", NULL};
execv("/bin/ls", array);
printf("This string should not be printed!\n");
}
```

Process Creation/ Coordination

- fork()
  - Create a child process
  - Identical to parent EXCEPT for return value of fork() call
  - Determines child/parent
- getpid() / getppid()
  - Get process ID of the currently running process
  - Get parent process ID
- exec() family
  - Replace currently running process with a different image
  - Process becomes something else losing previous code
  - Focus on execvp()
- wait() / waitpid()
  - Wait for any child to finish (wait)
  - Wait for a specific child to finish (waitpid)
  - Get return status of child

Waiting for a child to finish – wait()

#include <sys/types.h>

#include <wait.h>

pid_t wait(int *status);

- Suspends/blocks calling process until child has finished
- Allow parent to be able to monitor the children to find out when and how they terminate.
- Returns:
  - Process ID of **a** terminated child on success
  - -1 on error, sets **errno**
- Parameters:
  - **status**: is a memory buffer set by **wait** in which termination status of child is populated, and evaluated using specific macros defined for **wait**.

Example n.c

- Observe wait for each child by a parent
- child_wait.c
- child_status.c
- child_allstatus.c

Wait() limitations

- The *wait()* system call has a number of limitations:
  - If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
  - If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.

Waiting for specific child to finish—waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statloc, int options)
```

- Returns:
  - process ID : if OK,
  - 0 : if non-blocking option && no zombies around
  - -1 : on error
- Parameters:
  - Pid o child process
  - statloc: status
  - options

wait() Vs waitPID()

Wait()	Waitpid()
wait blocks the caller until **a** child process terminates	waitpid can be either blocking or non-blocking:If *options* is 0, then it is blocking If *options* is WNOHANG, then is it non-blocking
if more than one child is running then wait() returns the first time **one of the parent's offspring exits**	waitpid is more flexible: If *pid == -1*, it waits for any child process. In this respect, waitpid is equivalent to wait If *pid > 0*, it waits for the child whose process ID equals pid If *pid == 0*, it waits for any child whose process group ID equals that of the calling process If *pid < -1*, it waits for any child whose process group ID equals that absolute value of pid

Example

- Observe waiting for a specific child
- Get status of the exited child
- Several children

Orphans and Zombies

- A parent may not outlive a child
- Who becomes the parent of an *orphaned* child?
  - The orphaned child is adopted by *init*, the ancestor of all processes, whose process ID is 1.
  - A way to also determine if true parent is alive assuming child was created by a non-init process
- What happens to a child that terminates before its parent has had a chance to perform a *wait()*?
  - The zombie

Zombies—corpses revived by witchcraft

- What happens on termination?
  - When process terminates, still consumes system resources
- Entries in various tables & info maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

Gathering information about Zombies

- Performed by parent on terminated child (using wait or waitpid)
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by init process (pid == 1)
  - So, only need explicit reaping in long-running processes n e.g., shells and servers

Zombies Vs Orphans

- Zombie: has completed execution, still has an entry in the process table as parent performs wait later.
- Orphan: parent has finished or terminated while the child process is still running

Example k.c

- Observe Zombie

exit()

void exit(int status)

- Exits a process
- Normally return with status 0

atexit()

- Registers functions to be executed upon exit

```
void cleanup(void) {
printf("cleaning up\n"); }
int main() {
atexit(cleanup);
fork();
exit(0);
}
```

Process Termination

- Voluntary termination
  - Normal exit
    - return zero from main(), exit(0)
  - Error exit
    - exit(1)
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - kill procID, end task

When a process terminates

- When a child process terminates:
  - Exit handlers are called in reverse order of registration
  - Open files are flushed and closed
  - Parent process is notified via signal SIGCHLD (more on this later)
  - Exit status is available to parent via wait()
  - Child’s resources are de-allocated
    - File descriptors, memory, semaphores, file locks, …