Introduction to Operating Systems

Processes and Threads

Contents

List of Figures

List of Tables

1 Acknowledgments

As with any Computer Science course, Introduction to Operating Systems is constantly evolving. This evolution does not happen in a vacuum. In addition to many outside resources being consulted, we have been very fortunate to have important contributions from our TAs. Ted Cooper and Jeremiah Peschka have been particularly valuable in improving these lecture notes, as editors and contributors. Ted and Matt Spohrer developed the quiz infrastructure and questions. The quiz questions form the basis of the study questions now included with each set of notes. Ted and Jeremiah have made significant contributions and improvements to the course projects which use the xv6 teaching operating system.

As always, any omissions and/or errors are mine, notifications of such are appreciated.

2 Learning Objectives

- Explain the differences between a process and a program.
- Define the term *process* and explain its parts.
- Define the term *thread of control* and distinguish it from “process”.
- Define and explain the concept of *process state.*
- Draw and explain the 3-state process model.
- Draw and explain the xv6 process state transition model.
- Draw and explain the 5-state process model.
- List and explain the purpose of the data structures, and their elements, used for process management.
- Explain, from the operating system perspective, process creation.
- Explain the relationship between a parent process and a child process.
- Explain the purpose of the *ZOMBIE* state.
- Explain the purpose of the *RUNNABLE* or *READY* state.
- Explain the steps of a context switch, listing them in the correct order.
- Explain what is meant by the phrase *A context switch is comprised of two context switches.*
- Explain the steps taken by the operating system to run a new program.
- Explain what steps an operating system must take to support the `fork()` system call.
- Explain the steps involved in the `exit()` system call.
- Discuss the advantages and disadvantages of multi-threading.
- Explain basic thread types: kernel, user, combined.

- Discuss the issues with scheduling in the presence of user-level threads.
- Discuss various threading approaches available on a modern Linux system.
- Explain the issues of reentrancy.
- Describe an approach to shared libraries that helps eliminate reentrancy issues.

3 Programs and Processes

3.1 Background

1. A computer platform consists of a collection of hardware resources, such as the processor, main memory, I/O modules, timers, disks, etc.

2. Computer applications are developed to perform some (usually quite specific) task. Typically they accept input (from a user, a program, or maybe a remote system), perform some processing, and generate output.

3. It is inefficient for applications to be written directly for a given hardware platform. The principal reasons are

   (a) Numerous applications can be developed for the same platform. Thus, it makes sense to develop common routines for accessing the resources of the computer. A device driver is one such example.

   (b) The processor only provides minimal support for multiprogramming. Additional software is required to manage the sharing of the processor and other resources by multiple applications concurrently.

   (c) It is necessary to protect the parts of one process from all other processes, as well as allowing sharing under limited circumstances.

4. Operating systems were developed to provide a convenient, feature-rich, secure, and consistent interface for applications to use. The OS is a layer of software between the applications and the computer hardware that supports applications and utilities.

5. Conceptually, an OS provides a uniform, abstract representation of resources that can be requested and accessed by applications. The OS must manage these resource abstractions.

We can now discuss how the OS can manage the execution of applications such that

- Resources are made available to multiple applications.
- The physical processor is switched among multiple applications so as to give the appearance that all are progressing simultaneously.
- The hardware resources are used efficiently.

The approach that we will cover in this course is to use a model wherein an application *program* is loaded into an abstract *process* for execution.

3.2 What is a Program

At its simplest, a program is a collection of instructions that are to be executed by the CPU. The program begins life as *source code* and is, most commonly, transformed into an executable file through the *compilation process*. Alternately, some programs are *interpreted*, but we will ignore interpreted programs in this course.

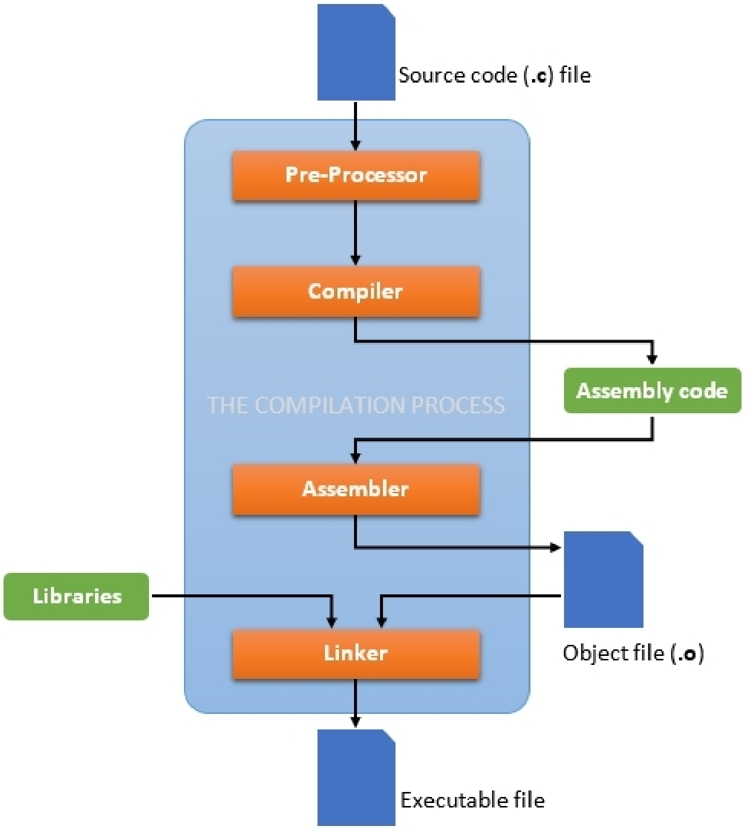


Figure 1: Compilation Process

*From codeforwin.org

An executable program has been assigned virtual addresses by the *linker* stage of the *compilation process*. The operating system *loader* reads the executable program file and places the various parts in memory as directed by the linker. We can thus see that virtual addresses are primarily assigned by the linker. Addresses on the stack and heap are assigned at run time.

A program can be considered a description of all possible actions that any process created from that executable may perform. A process is an instance of the program in execution and may or may not perform all possible actions defined by the program.

A program *becomes* a process when the *loader* copies the contents of the executable file into memory and begins execution at some specified location. The meta data included in an executable file contains information as to where execution should begin. See, for example, the ELF file format.

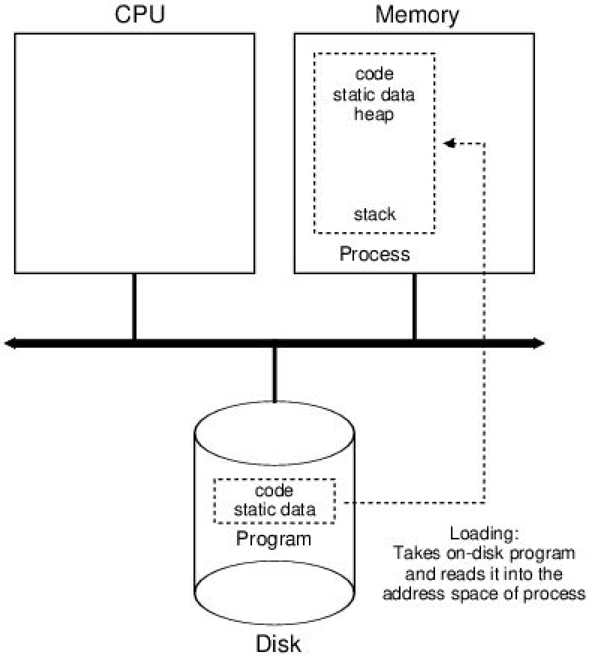


Figure 2: Program Load

3.3 What is a Process

There are several common, intuitive, definitions of a process

1. A program in execution. This is a vague but often useful definition.
2. An instance of a program running on a computer.
3. An entity that can be assigned to and execute on a processor. This is very useful to keep in mind. Processes, not programs, execute.
4. The execution of a sequence of instructions, together with a current state and set of resources. This is a good “big picture” way to look at a process.

We can also break down a process into a number of elements. The two most common are the *program code* and *process data.* At any point in its execution, a process can be uniquely characterized by a number of specific elements

1. The process identifier (PID). A unique identifier (often just a unique number) associated with this process that enables the system to identify it and distinguish it from all other processes on the system.
2. State. If a process is currently executing then it is in the RUNNING state, for example.
3. Priority. The priority level of the process *relative to all other processes* in the system.
4. Program counter (PC) aka instruction pointer (IP). The address of the *next instruction* in the process to be executed. The IP is modified at execution for constructs such as a loop or function call.

5. Memory pointers. Various pointers to process code and associated data. The data can be dynamically allocated on the heap or statically allocated on the stack. The stack and heap are managed differently and the data allocated on them have very different lifetimes.

6. Process context. The CPU register values associated with the process. When the process is in the running state, the context is loaded into the processor. When not in the running state, the context is stored in the process control block.

7. I/O information. Includes outstanding I/O requests, I/O devices in use, the list of open files, etc.

8. Accounting information. May include *elapsed time*, *CPU time used*, ownership information (UID, GID), and other ancillary information.

This information is stored in the process control block, which contains all the information necessary to return a process to the running state without the process being able to determine if it was ever removed from the processor; usually via a context switch or interrupt processing.

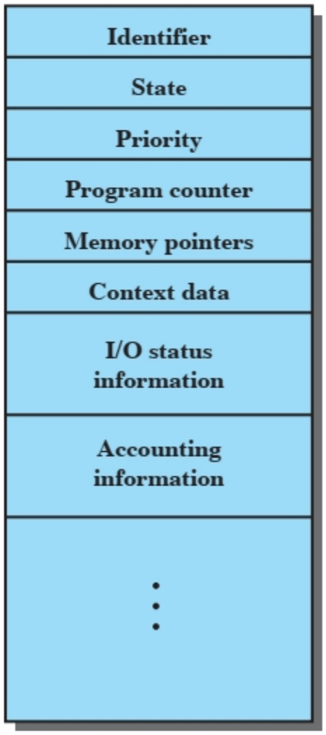
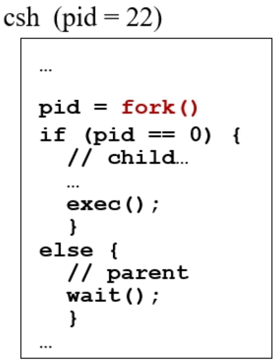


Figure 3: Example Process Control Block

3.4 Process Creation

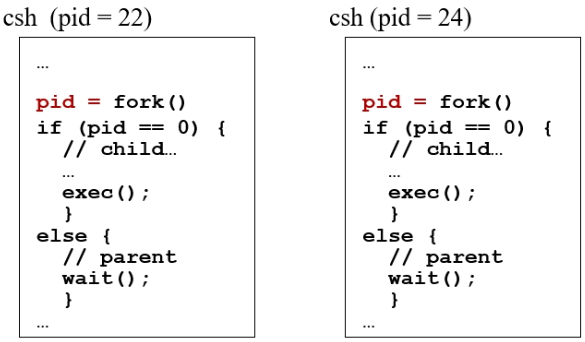
Once the operating system is booted, any new processes are created by the same mechanism: the `fork()` system call. The fork call is unique in that it is called once but returns twice.



```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

(a) Fork invocation

Figure 4: Using the fork() System Call

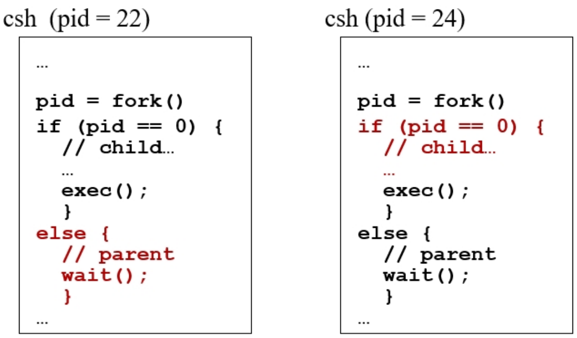


```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

(b) Fork creates a second, almost identical, process

Figure 4: Fork *(continued)*

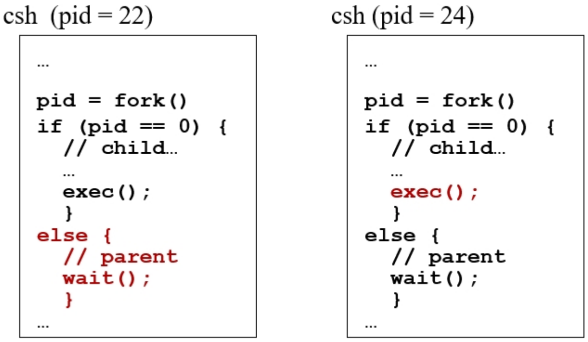


```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

```
…
pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

(c) Return value indicates which is parent/child

Figure 4: Fork *(continued)*

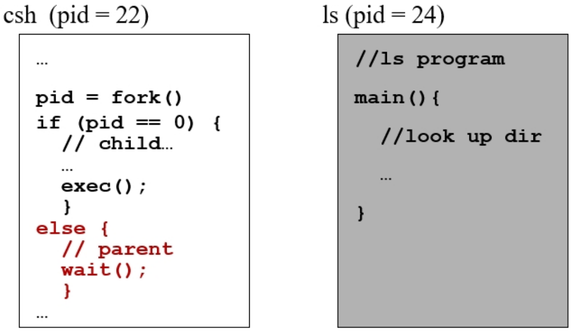


```
...
pid = fork()
if (pid == 0) {
  // child...
  ...
  exec();
  }
else {
  // parent
  wait();
  }
...
```

```
...
pid = fork()
if (pid == 0) {
  // child...
  ...
  exec();
  }
else {
  // parent
  wait();
  }
...
```

(d) Do work specific to parent/child

Figure 4: Fork *(continued)*



```
...
pid = fork()
if (pid == 0) {
  // child...
  ...
  exec();
  }
else {
  // parent
  wait();
  }
...
```

```
//ls program
main(){
  //look up dir
  ...
}
```

(e) For example, run a new program via exec

Figure 4: Fork *(continued)*

The process that calls `fork()` is called the *parent* process. The new, *child*, process knows that it is the child because the return value from `fork()` is 0 in the child. The return value in the parent process will be the PID of the newly created process. Note that 0 is not a valid PID[1].

It is important to know that `fork()` does not run a new program. The `fork()` system call creates a new process that is an exact copy of the parent with three exceptions: the PID, the return value from fork(), and ancillary information in the process block identifying the process' parent. While the contents of the parent's memory address space are copied to the memory allocated for the child, the child's memory is separate from the parent's, i.e. the child's writes to memory do not affect the parent's memory, and vice versa. To run a new program, the `exec()` family of system calls is used.

3.5 Process Termination

- Normal exit (voluntary). Typically `exit(0)`, or `exit(EXIT_SUCCESS)`, or (in xv6) exit().
- Error exit (voluntary). Exit value is typically negative, `exit(-1)` or `exit(EXIT_FAILURE)`. Not supported in xv6.

[1] Carefully consider the issues if this were not the case.

- Fatal error (involuntary)
- Killed by signal from another process (involuntary)

Most compilers allow normal program termination via "falling off" the `main` routine or using the `return()` function in `main`.

In xv6, the `exit()` call must be used and it takes no argument. Not ideal, but workable.

3.6 Principle Parts of a Process

- A process virtual address space is comprised of four regions: code, data (sometimes called heap[2]), unused (or unallocated), and stack. *This is a very simplified model that will suffice for this class.* There is also a kernel region mapped to the same address range in *all* processes. We will ignore the kernel mapping when convenient as it is a constant.

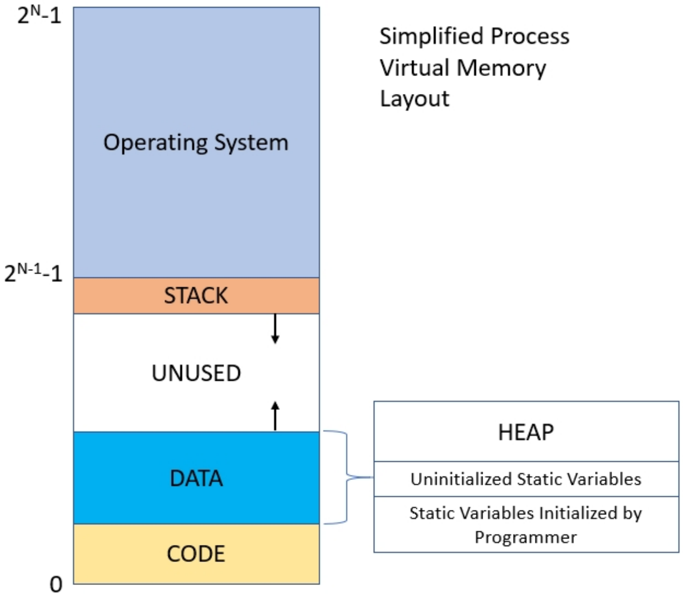


Figure 5: Model of a Process

- Process size
  - ▷ The process size is measured in bytes. Addresses range from 0 to 2^n-1, where *n* is the number of bits in the architecture. It is critical to note that these are *virtual addresses.* Virtual addresses help to isolate and protect one process from another.
  - ▷ In this model, the operating system is assigned the upper half of the address range, from address 2^{n-1} to 2^n-1. For a 32-bit architecture, this gives the kernel 2 GB and the process 2GB of virtual memory. It is also common for the kernel to be assigned the top 1 GB and the process the remaining 3 GB of virtual memory.

[2] The data area is more than the heap, but the use is common.

  - The same kernel is mapped to the high addresses in every process. CPU modes and virtual memory mappings help protect the kernel from errant, or malicious, user programs. While the kernel code is the same for each process, each process has a private stack for use in kernel mode. This is visible to students when system call arguments are removed from the user stack in `sysproc.c` and `sysfile.c`.
  - The size of the *code segment* is known at compile time, as are the sizes for both initialized and uninitialized static data. These parts of the address space do not change over the lifetime of the process.
  - The *data segment* starts relatively small. The *heap* portion can grow towards *higher virtual addresses.* The heap is managed by the user process.
  - The base, or bottom, of the *stack segment* is usually at the high end of the user-mode portion of the process virtual address space, 2^{n-1}-1, and grows towards *lower virtual addresses.* The stack is (primarily) managed automatically for the user process by the operating system and hardware.

    Important: The xv6 kernel does not follow the traditional layout for the stack segment. It is possible to move the stack to the usual location and is a project used by many undergraduate courses in operating systems. If you wish to do this, talk with the instructor at the end of the term.

  Note: all possible virtual memory addresses are within the process address space. Think about the ramifications of this model.

- Accounting information. The operating system must be able to track all processes. In xv6, this is done with the `struct proc`. There is one instance of this structure for each process. This structure is often known as the *process control block* (PCB) in OS literature.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  uint pid;                    // Process ID
  struct proc *parent;         // Parent process. NULL indicates no parent
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

- All processes are accessed through the `ptable` structure, located in `proc.c`. Note that this structure has a lock associated with it. In order to access `most`[3] data in the `ptable.proc[]` array, the lock must be held. A lock is acquired with the `acquire()` kernel function and released with the `release()` kernel function. Both functions are defined in `spinlock.c`.

[3] We will see exceptions in the lecture on concurrency control, but most of the time, the lock must be held.

```
static struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

3.6.1 Other Parts of a Process

There are many *ancillary*[4] parts of a process. Things such as the *CPU context*, open files, current working directory, etc., establish the environment in which a process is executed. The contents of ancillary parts of a process may differ substantially from one instantiation of a program to another. The ancillary parts of the process are stored in the *process control block*, `struct proc` in xv6.

The process control block is where the CPU context is stored for any process not currently executing in the CPU.

3.7 Parents and Children

All processes have a *parent* except for the first process, which is created at boot. This first process is generally called `init` and is considered the ultimate parent for all processes. Each process control block contains a pointer to its parent process. Only for the `init` process will this be a NULL pointer; at any other time this is a catastrophic error. If a process exits while its children are still alive, the operating system must arrange for the `init` process to be set as the parent for these *orphaned* processes; a process called *adoption*[5].

Thus, all processes on a system form a *singly-rooted tree* where interior nodes denote processes with active child processes and leaf nodes denote processes without children. Another way to look at this is that interior nodes take on *both* the role of parent and child. Leaf nodes are child processes only.

When a child process calls `exit()`, the process structure for the child process will be released with the exception of the exit value set by the child process and its process identifier, PID. A process in this state is called a *zombie*. The parent can *reap* the child process and read the exit value with the `wait()` system call. Many systems also have `waitpid()` and similar calls as well; xv6 does not.

The system calls `getpid()` and `getppid()` allow a process to get its own PID and also the PID of its parent. There is no system call to return the PID of child processes. A process can have 0 or more child processes but a process can only have one parent process (`init` being the only exception).

3.8 Switching Processes

We have seen that each active process has a *context*. This context captures the processor state for the process. This enables the process to be temporarily suspended from execution and later resumed without the process being able to tell that it was suspended. It literally happens between one assembly instruction and the next!

It is important to note that a context switch, as it is commonly understood, is multi-step. See Figure 6.

[4] ancillary: providing necessary support to the primary activities. In xv6, additional data necessary for the correct operation of the process.

[5] UNIX developers had a well-developed sense of humor.

3.8.1 XV6 Process Switch

A context switch happens within the kernel; specifically the routine `scheduler()` selects the next process to be put into the CPU and then activates the process by loading its context via the routine `swtch()` which is located in `swtch.S`.

The process context is stored in the `struct context` which is located in the `struct proc` for each process.

```
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

The actual context switch is performed by the `swtch()` routine

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

# Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

# Switch stacks
```

```
    movl %esp, (%eax)
    movl %edx, %esp

  # Load new callee-save registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

A context switch appears to merely *switch the CPU from process A to process B*. However, the process is much more complicated than this.

3.8.2 The Role of the Scheduler

That part of the operating system which allocates a process to a CPU is called the *scheduler*. It is the responsibility of the scheduler to *select the next process* to run in the CPU. One process does not hand off control of the CPU to another process directly. All this work is performed by the scheduler. Each CPU will have its own *private scheduler context*, so that each time that CPU's scheduler runs, it can pick up where it left off in the scheduler's code.

It is a common misconception that one user process "switches" to another user process in a context switch. That is, one process "knows" which process will next be loaded into the processor. If only it were that easy! There are 5 main steps

1. The currently executing process, P_A, gives up the CPU either voluntarily (e.g. by making a system call), or non-voluntarily (e.g. because the timer interrupt fires). The kernel saves P_A's context to its process control block.

2. The code that removes the context for P_A then causes the context for the scheduler to be loaded.

3. The scheduler then runs an algorithm to determine the next process, P_B, to run.

4. The scheduler context is saved.

5. The context for P_B is loaded into the processor. It is important to note that the instruction pointer is the last register loaded from the saved context.

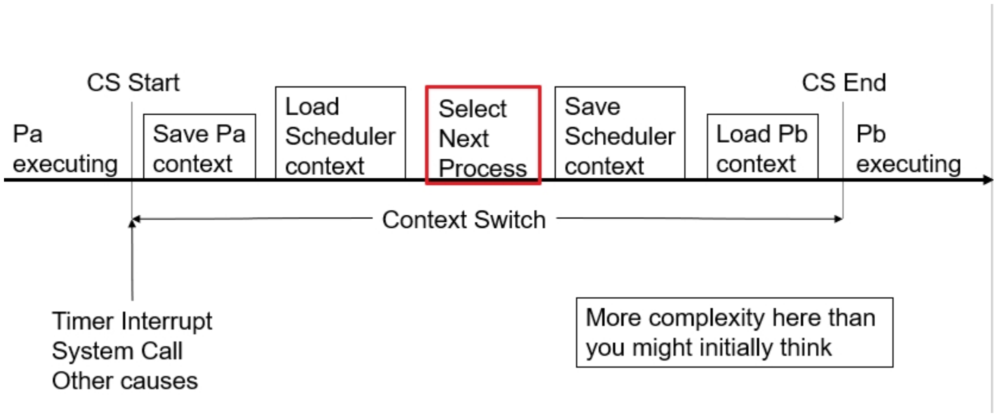


Figure 6: Context Switch

While the *scheduling process* includes taking processes out of the CPU and placing them in other states, the *scheduler* is only involved in the *selection* of the next process to run. In xv6, the scheduler is the routine `scheduler()` in `proc.c`.

The xv6 code that initiates the saving of the current process context and loading the context of the scheduler is located in the routine `sched()` in the file `proc.c`:

```
swtch(&p->context, mycpu()->scheduler);
```

and the code that saves the scheduler context and loads the context of the just-selected process is in the routine `scheduler()`, also in `proc.c`:

```
swtch(&(c->scheduler), p->context);
```

Of course, there is much more work involved. See the code for details.

3.9 Executing a Program

For completeness, we now discuss the `exec` family of calls. On Linux systems, this is part of the C library but, with xv6, `exec()` is a system call. The best way to learn about them is to practice with them after reading the on-line documentation; use the command `man 3 exec` on department Linux systems. The following code is a Linux example used for some sections of CS 201.

```
$ cat runit.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

extern int errno;

int
main(int argc, char *argv[])
{
  if (argc < 2) {
    fprintf(stderr, "You must provide a command to run!\n");
    fprintf(stderr, "Exiting...\n");
    exit(EXIT_FAILURE);
  }

  ++argv;  // a very cool trick

  execvp(argv[0], argv);
  fprintf(stderr, "%s: %s\n", argv[0], strerror(errno));

  exit(EXIT_FAILURE);
}
```

Here are some examples of the program executing. Note the error message in the second example and see if you can figure out how the original source code caused it to be displayed.

```
$ gcc -o runit runit.c
$ ./runit ls -a
.  ..  childRunIt.c  fork.c  runit  runit.c  sigchld.c
$
$ ./runit badcommand
badcommand: No such file or directory
$
```

While Linux has several *variants* of the `exec()` system call, xv6 only has the `exec()` system call[6], which works very much like `execve()` in Linux.

The `exec()` system call is often termed the *loader*. This is the part of the operating systems that copies (loads) data from the object code file – executable, library, etc. – into the address space of an *existing* process[7]. There is much more work to do than merely copying data, however.

In computer systems a loader is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

In Unix, the loader is the handler for the system call *execve()*. The Unix loader's tasks include:

1. validation (permissions, memory requirements etc.);

2. copying the program image from the disk into main memory;

3. copying the command-line arguments on the stack;

4. initializing registers (e.g., the stack pointer);

5. jumping to the program entry point (_start).

[6] See the file `exec.c` for all the gory details.

[7] See Wikipedia.

In Microsoft Windows 7 and above, the loader is the LdrInitializeThunk function contained in ntdll.dll, that does the following:

1. initialization of structures in the DLL itself (i.e. critical sections, module lists);
2. validation of executable to load;
3. creation of a heap (via the function RtlCreateHeap);
4. allocation of environment variable block and PATH block;
5. addition of executable and NTDLL to the module list (a doubly-linked list);
6. loading of KERNEL32.DLL to obtain several important functions, for instance BaseThreadInitThunk;
7. loading of executable's imports (i.e. dynamic-link libraries) recursively (check the imports of the imports, their imports and so on, recursively);
8. in debug mode, raising of system breakpoint;
9. initialization of DLLs;
10. garbage collection;
11. calling NtContinue on the context parameter given to the loader function (i.e. jumping to RtlUserThreadStart, that will start the executable)

4 Implementing Safe Mode Transfer

When transferring to or from kernel-mode, care must be taken so that buggy or malicious user processes cannot corrupt the kernel or any other process. The operating system must carefully save process state while still potentially executing instructions that could impact the state of the process that it is currently saving. This can be quite complex.

At a minimum, the mode transfer sequence must provide:

- Limited entry into the kernel. User processes cannot be allowed to transfer control to arbitrary kernel addresses. The transfer must only be done through certain, well-defined, control points. The most common entry point into the kernel is via a *system call*. For example, on entry for a system call, the kernel must ensure that the process is allowed to make that system call (some could be privileged). Then any arguments must be checked for correctness – *the kernel cannot assume that it is always called correctly!* Then permissions for the process may need to be checked and finally, the work performed and a correct return code or an error code generated and returned to the user process.

- Atomic changes to processor state. When in user-mode, the process uses addresses in its non-privileged memory regions. On entering the kernel, however, kernel address are (mostly) used. The processor cooperates with the operating system to ensure that this change occur *atomically* so that no buggy or malicious code can interfere. Also, there are two stacks in use, one for user-mode and one for kernel-mode. It is important to note that all system call arguments will be passed on the user-mode stack and that, once in kernel-mode, the operating system must have a mechanism to retrieve the arguments safely. This transition is complex and processor-specific.

- Transparent, restartable execution. An event can trigger an interrupt at any point in the execution of the user process. The infrastructure for supporting interrupts must be able to respond quickly to the interrupt while enabling the user process to be resumed at the exact point of interruption at some later time.

4.1 Handling Interrupts

When an interrupt occurs, the processor saves its current state (the process context) and then changes to kernel-mode. In many systems, further instances of the interrupt are deferred (remembered and then come back to); other interrupts may also be deferred. Once in the kernel, the correct *interrupt handler* must be invoked. Once the interrupt is handled, the process is reversed and the user process resumed without the process knowing that an interrupt has occurred.

The principle approach used by most processor is the *interrupt vector table*, also called the interrupt descriptor table. This is a section of kernel memory that is indexed by the *interrupt number* – think Intel scaled index mode addressing. So each interrupt type has to have a unique number by which it identifies the interrupt to the system. An example of interrupt numbering for the ARM architecture is shown in Figure 7.

The IVT is organized as a function call jump (or dispatch) table. Each entry corresponds to a specific interrupt type and the entry holds the address of the entry point in the operating system for handling the interrupt, called the handler. This data structure is populated at boot time by the operating system and then is set to read-only to prevent modification.

Processors reserve some interrupt numbers for processor-specific exceptions, such as divide by zero. For example, on the Intel x86 architecture, IVT entries 0 - 31 are reserved for processor exceptions. Entries 32 - 255 (255 is the highest unsigned number that can be stored in a byte) are typically specified by the operating system. The interrupt number for all system calls is, for example, 64 (0x40) for xv6 and 128 (0x80) on Linux.

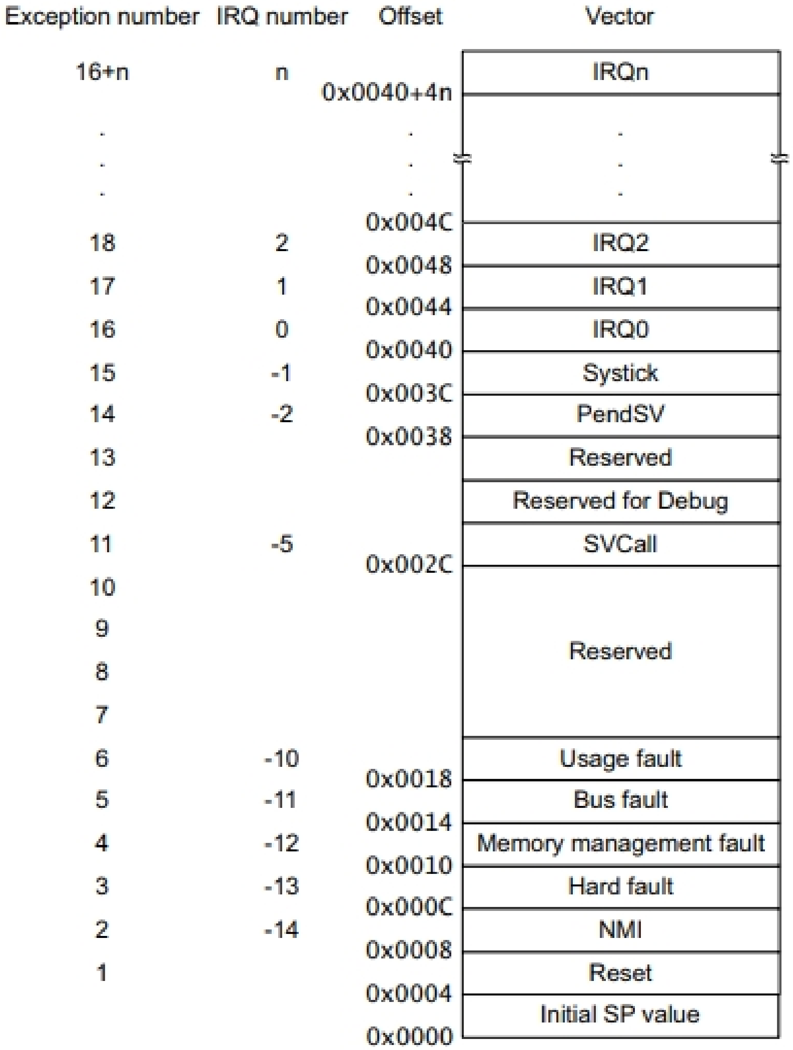


Figure 7: ARM Interrupt Vector Table

Should the process state be saved on its stack, the kernel stack, or someplace else? A common approach is for there to be a separate interrupt stack. At the start of the interrupt, the processor saves critical information here and then the invoked interrupt handler can save any additional registers required. When returning from the interrupt, the interrupt handler pops the registers that it saved off the stack and then executes a special return-from-interrupt instruction, `iret` on x86, which causes the processor to restore the information that it had placed on the stack in order to properly resume the interrupted user process. For many systems, the interrupt stack is the kernel stack for that process.

4.2 X86 Interrupt Processing

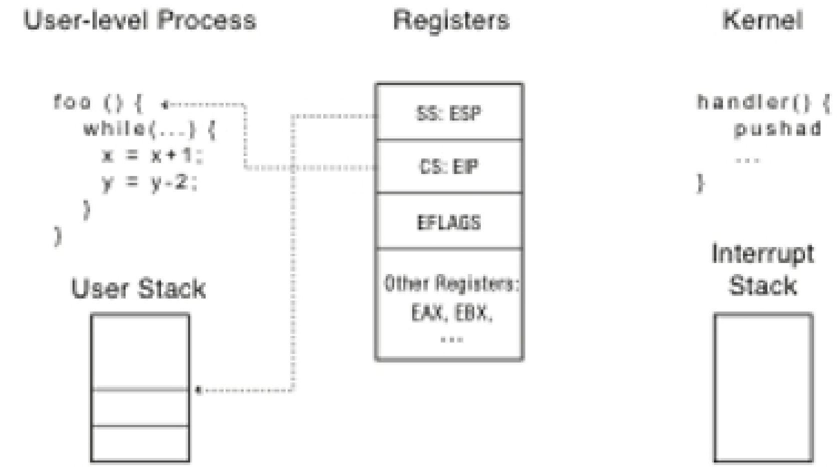


Figure 8: Before Interrupt

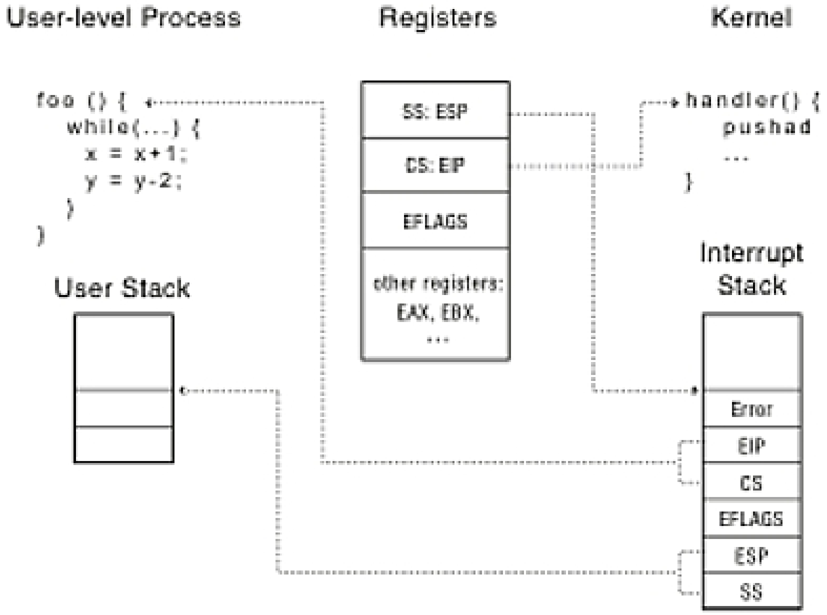
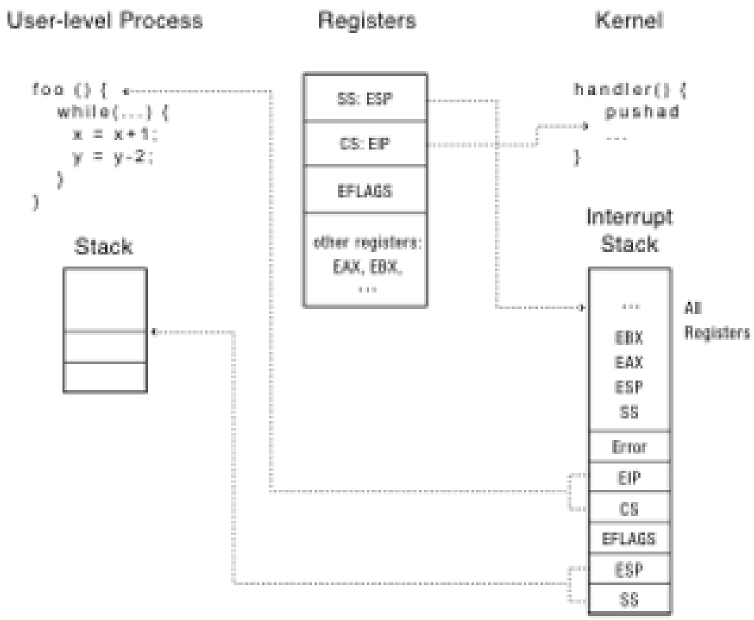


Figure 9: During Interrupt Processing



```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

```
handler() {
  pushad
  ...
}
```

Figure 10: After Interrupt

Question: Why is the stack pointer saved twice? Do they both point to the same stack?

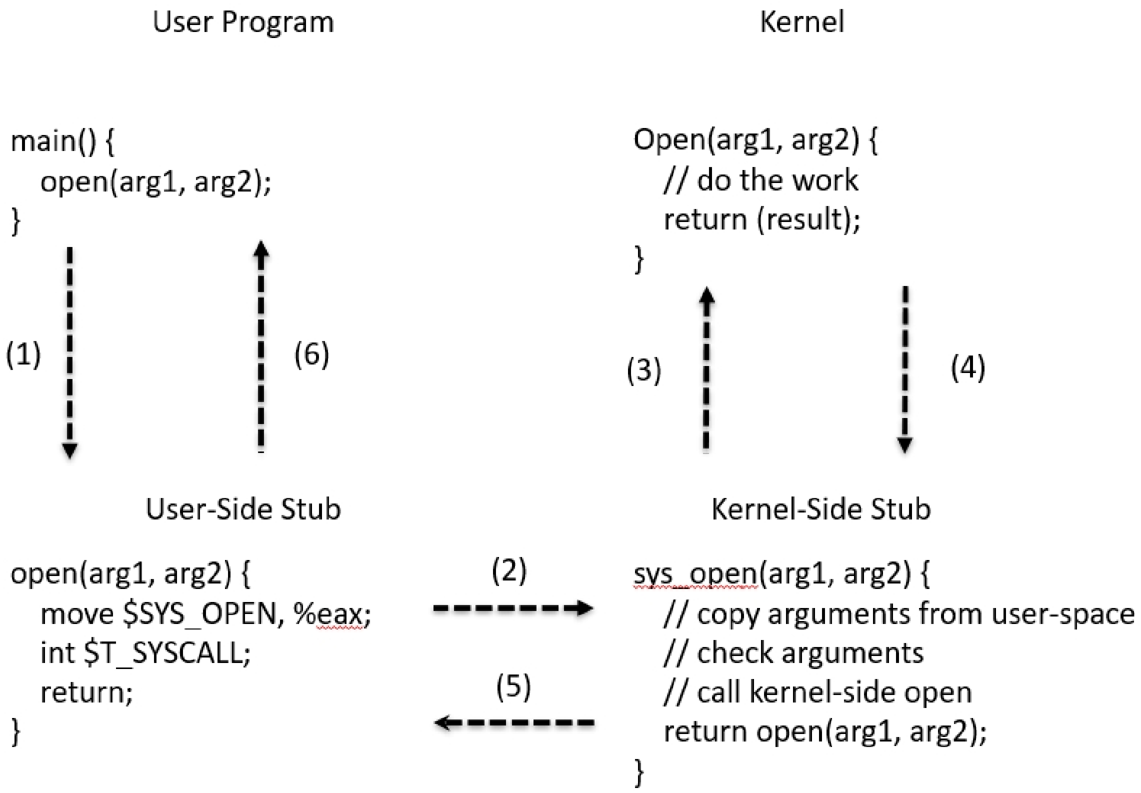
The process for initiating interrupt handling on the x86 architecture has 6 main steps:

1. Mask interrupts. This is to prevent any new interrupts from impacting the mode switch. Note that the interrupts are queued, not ignored.
2. Save key registers. The user-mode stack pointer (`esp` and `ss` registers on x86), the condition codes and flags (`eflags` register on x86), and the instruction pointer (`eip` and `cs` registers on x86). These are saved to a reserved place in the processor until they can be saved to the interrupt stack.
3. Switch to the kernel interrupt stack. The stack pointer is now set to the kernel (interrupt) stack.
4. Push the values from step 2 onto the new stack.
5. Optionally save an error code. Some exceptions generate an error code for later processing. This code is saved here or a dummy value that is ignored for this exception type.
6. Invoke the interrupt handler. Use the IVT as a jump table to begin executing the kernel code for handling this specific interrupt number.

4.3 Implementing Secure System Calls

While system calls look to the user process just like any other function or C library call, they aren't. There is complexity due to the switch in address spaces and stacks, at the least. There are 6 basic steps to implementing a system call as shown in Figure 11.

1. The user program invokes the user-mode stub. This is usually located in a system library.
2. The stub places the system call number in the %eax register and them executes the system call interrupt number to trap into the operating system. The trap handler checks the arguments and any other required checks.
3. The hardware (processor) changes mode to kernel, changes the stack to the kernel (interrupt) stack and performs other bookkeeping.
4. The system call completes and returns to the kernel-side handler.
5. The handler returns to the next instruction in the user-mode stub.
6. The stub returns to the caller with the return code in register %eax.



```
main() {
   open(arg1, arg2);
}
```

```
Open(arg1, arg2) {
   // do the work
   return (result);
}
```

```
open(arg1, arg2) {
   move $SYS_OPEN, %eax;
   int $T_SYSCALL;
   return;
}
```

```
sys_open(arg1, arg2) {
   // copy arguments from user-space
   // check arguments
   // call kernel-side open
   return open(arg1, arg2);
}
```

Figure 11: Implementing Secure System Calls

5 Process State Transition Model

A process has a *lifetime*. A process comes into existence through an operating system specific mechanism. A process exiting the system also uses an operating system specific mechanism. In between, all processes follow the same *state machine* , which can be modeled as a *deterministic finite state automaton*[8].

5.1 Idealized Process Model

The idealized model captures the core state transitions for an operating system that uses a process model. It omits details such as *process creation* and what happens after a process calls the `exit()` system call. Such details are implementation specific and will vary from operating system to operating system.

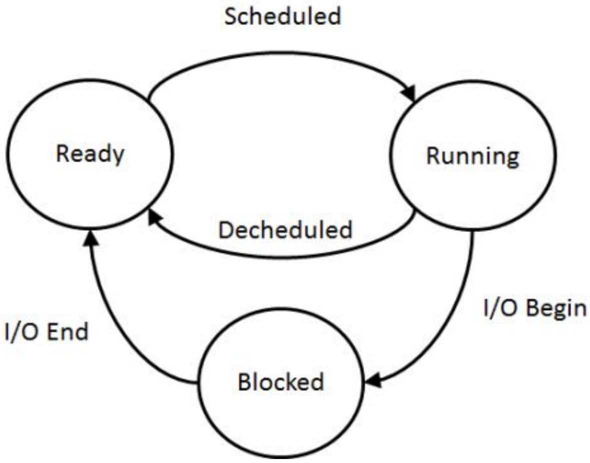


Figure 12: Idealized Process State Transition Model

The simplified model just tracks the process through its execution states. There are more complex models as well. For example, the 5-state process model:

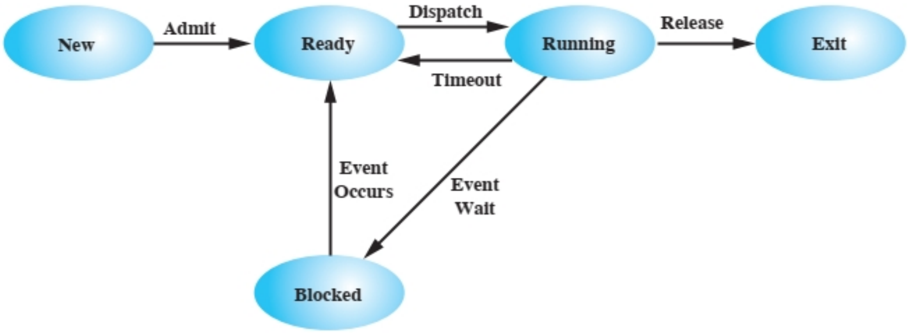


Figure 13: Five State Process Model

These five states are defined as

1. New. The new state is a process that has been created, initialized, and now can be scheduled to run. Once this step is completed, the process transition to the ready state.

[8] See Wikipedia.

Reason	Description
New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands. In modern parlance, we would term these *background* jobs.
Interactive logon	A user at a terminal logs on to the system.
New service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Table 1: Reasons for Process Creation

2. Ready. A process in this state can make progress if it is assigned to a processor. Since there may be more processes in the system than processors, this is a holding state until a processor becomes available to execute the process. This may be an initial execution for the process or a return of an interrupted process (context switch) back to the processor to continue where it left off.

3. Running. The process is now executing directly on a processor. This is the only state in which a process can execute instructions. Note that a process must be in the running state in order to exit the system.

4. Blocked. A process is in this state if it is waiting for some external event such as a signal. We call this state "blocked" or "sleeping" to indicate that it cannot be assigned to a processor until the external event occurs.

   When a process is in the blocked state, there are different ways to manage the waiting processes. The most common are to use a unified (single) blocked queue or to use multiple blocked queues, one per event type. Process priority can be taken into account here but it is not obvious how to implement it. Remember that a blocked process is waiting on an *external event*; that is, an event that must be signaled by a different thread of control.

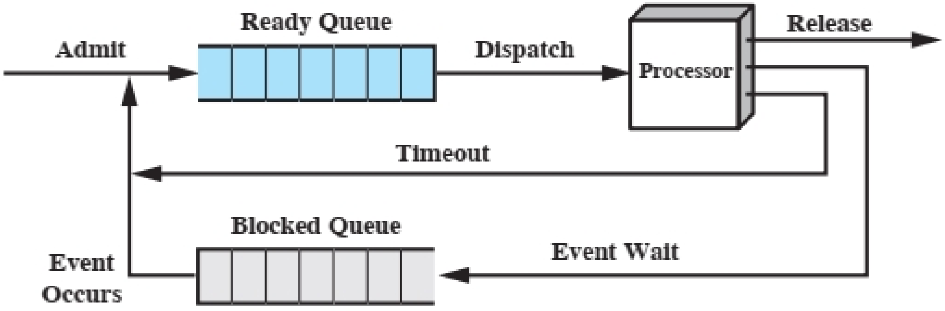


Figure 14: Unified Blocked Queue

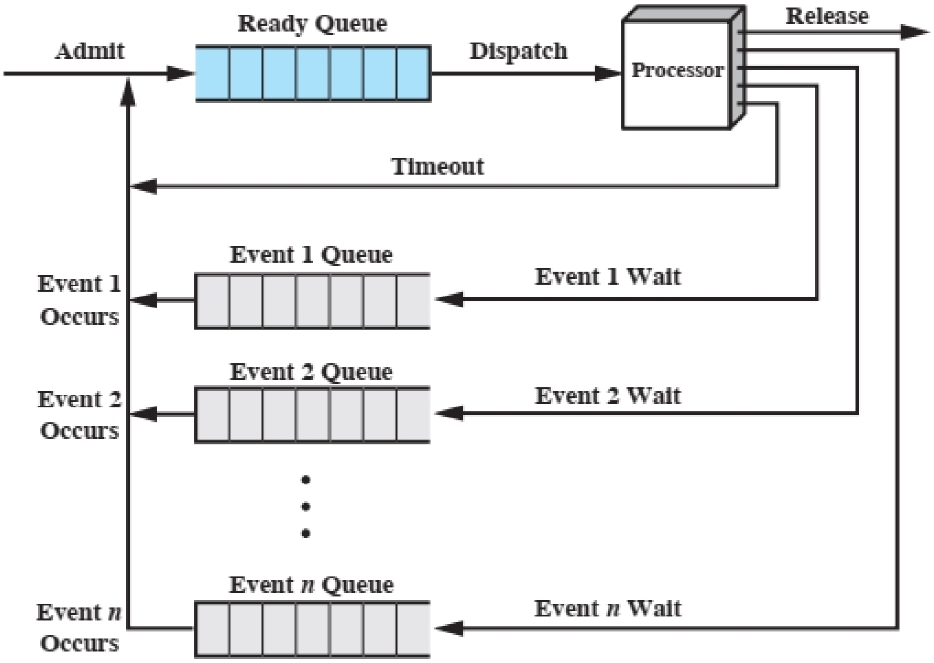


Figure 15: Multiple Blocked Queues

There are many reasons to suspend (block) a process.

Reason	Description
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
User request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource. A user may also use a signal (e.g., control-z) to suspend an executing process.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Table 2: Reasons for Process Suspension

The system may have other uses of the blocked state that is specific to that particular operating system.

5. Exit. The exit state exists for process cleanup. The resources are released. This includes the code, data (stack and heap), open files, etc. The process control block at this stage is mostly invalid, except for information that is returned to the process that created this one (the parent) and accounting information.

Reason	**Description**
Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent request	A parent process typically has the authority to terminate any of its offspring.

Table 3: Reasons for Process Termination

5.2 xv6 Process Model

The xv6 model contains the idealized model at its core. Since this model is an *instantiation* of that model, it must account for *process creation* and *process death*. The full life cycle of a process in xv6 is shown in Figure 16.

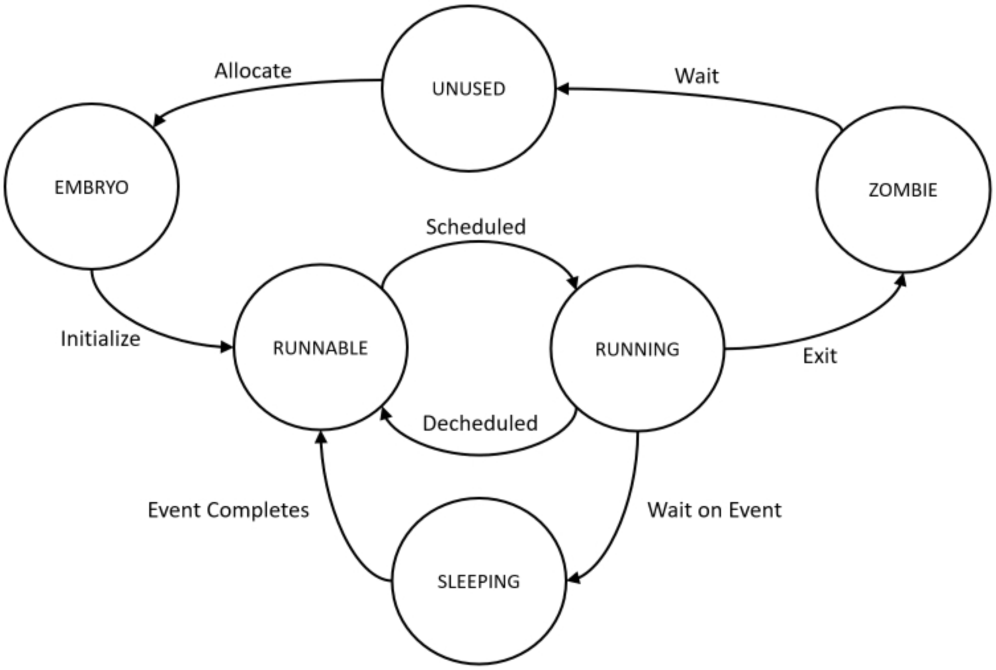


Figure 16: xv6 Process State Transition Model

The xv6 model has six distinct states.

1. UNUSED. For simplicity, xv6 has a limited number of processes. The process control blocks are created at boot time and stored in an array in the `ptable` structure. By default, the maximum number of processes in xv6 is 64, `NPROC`. The routine `allocproc()` finds an UNUSED process and allocates it by changing its state to EMBRYO and setting the process identifier, PID, and completing some other setup steps.

2. EMBRYO. An allocated process still has a lot of work to do before it can be put into the RUNNABLE state to be eventually scheduled on a CPU. The EMBRYO state captures this nicely. Only the thread of control that called `allocproc()` has a pointer to this newly allocated process. It is the responsibility of this thread of control to initialize the process and make it ready for execution. This work occurs in the `fork()` routine in `proc.c` except for the first process which is in initialized in `userinit()` in `proc.c`.

3. RUNNABLE. Any process in the RUNNABLE state is able to be scheduled on a CPU. The *scheduler* is tasked with finding a process in the RUNNABLE state and allocating it to a CPU when appropriate and when such a process exists. The scheduler is `scheduler()` in `proc.c`. Note that in most operating systems, the RUNNABLE state is referred to as the *ready* state.

4. RUNNING. When a process is active on a CPU, its state is set to RUNNING.

5. SLEEPING. When a process leaves the CPU because it is waiting on a future event, such as I/O, it will be placed in the SLEEPING, also called the *blocked*, state.

6. ZOMBIE .Once a process has invoked the exit() system call, in `proc.c`, its state will be set to ZOMBIE, all of its open file descriptors will be closed, and parent of all of its children will be reassigned to the `init` process. The process will remain in this state until its parent process calls the `wait()` system call to reap the ZOMBIE, freeing the process' stack and virtual memory data structures, setting the process' state to UNUSED, and zeroing all remaining fields in the process control block. Now, the process control block is ready to be reused by a future call to `allocproc()`. The `wait()` system call returns the PID of the child process. Note that since the xv6 `exit()` system call does not include an argument – this is different from Linux/Unix – the `wait()` system call does not take an argument either.

5.3 Modern UNIX Process Model

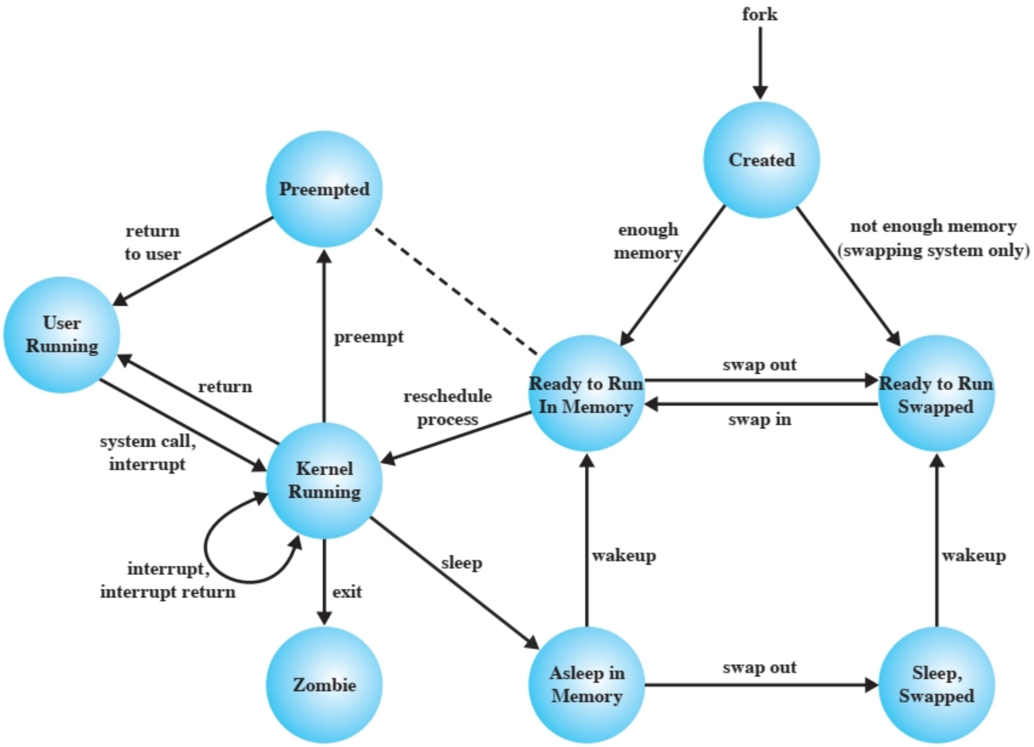


Figure 17: UNIX Process State Transition Model

6 Thread of Control

A process has these principal components:

1. a virtual address space, including at least one stack

2. a collection of operating system state such as open files, sockets, locks, etc.
3. user ID, group ID, process ID
4. at least one CPU context ... or *thread of control*
5. contents of control registers

6.1 What is a thread

Also called a *thread of execution*[9].

Here is what the OSTEP book says about a thread

> [...] a new abstraction for a single running process: that of a **thread**. Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.
>
> The state of a single thread is thus very similar to that of a process. It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a **process control block (PCB)**; now, we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using)[10].

- Threads share all process information except
  - ▷ CPU context: IP, SP, other register values
  - ▷ Stack

  These are termed *thread private* data.

Important: Changes made to shared process state by one thread will be visible to all other threads of the same process.

A *thread of control*, or just *thread*, is the path that an instruction pointer takes through a program together with the data specific to that thread. Otherwise, all threads of the same process share the same code, heap, and process ancillary data.

Each thread has its own *CPU context* which includes its own set of *CPU registers* such as the instruction pointer, stack pointer, etc. Similarly, each thread has its own stack, the stack frames

[9] Here is a nice Wikipedia note.

[10] OSTEP, Ch. 26, *Concurrency and Threads*, p. 1.

for each thread will differ, as will the values for function-local variables. Since each thread will have its own CPU context and private stack, each process can have one or more active threads at any one time.

A process cannot exist without at least one thread. When a process has only a single thread, we do not distinguish between the thread and its process.

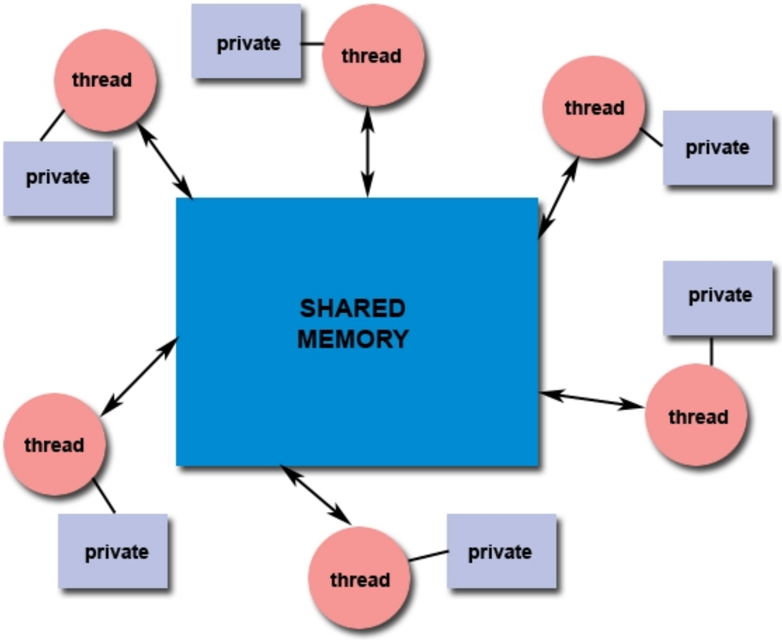


Figure 18: Pthreads Shared Memory Model

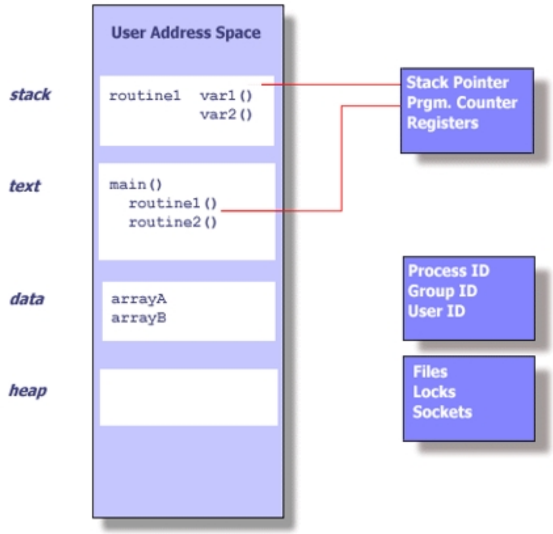


Figure 19: Single Thread of Control

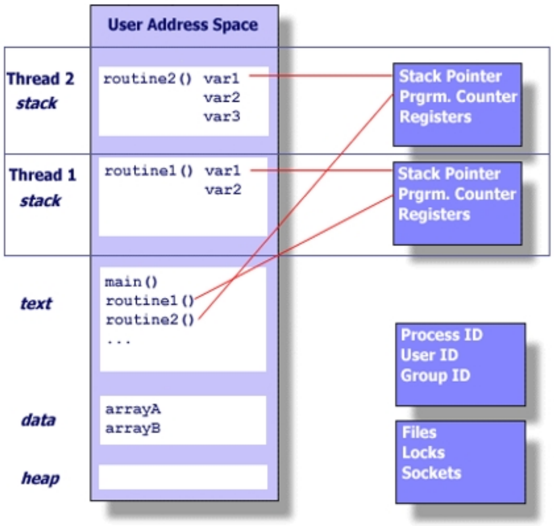


Figure 20: Multiple Threads of Control

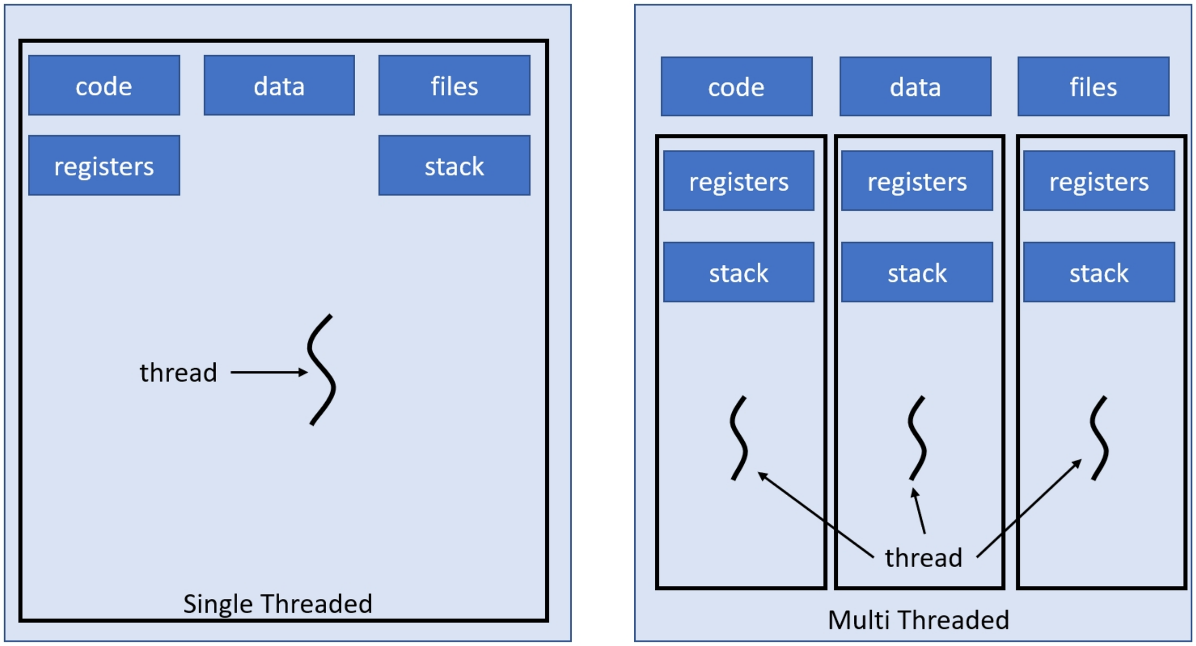


Figure 21: Single vs. Multi-Threading

Threads allow a program to scale to multiple processors/cores dynamically.

Split program into routines to execute in parallel

True or pseudo (interleaved) parallelism

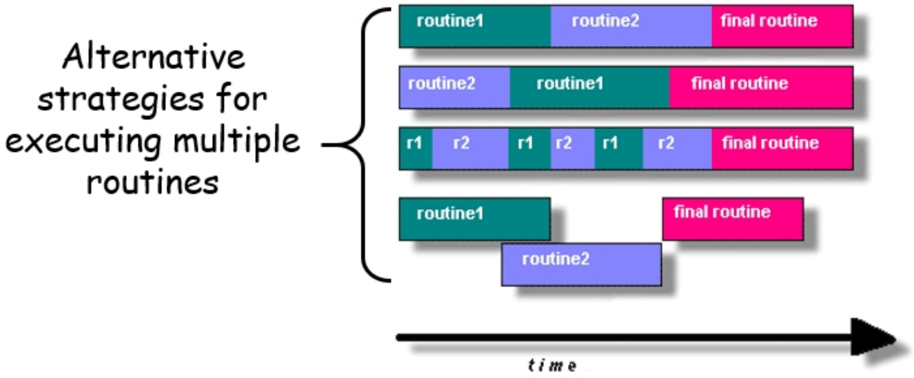


Figure 22: Parallelism with Threads

Question: Why does a multi-threaded program automatically scale to differing numbers of processors?

6.1.1 Advantages of Multi-Threading

1. Utilize multiple CPUs concurrently. Better mapping to multiple CPUs.
2. A thread context switch is much "cheaper" than a process context switch.
3. Low cost communication between threads using shared memory
4. Better memory utilization as threads *share* the process address space.
5. Overlap computation and blocking on a single CPU
   - ▷ Use blocking I/O in one thread
   - ▷ Computation and communication in another thread or threads
6. Simplified handling asynchronous events – one thread per event type!

6.1.2 Disadvantages of Multi-Threading

1. Potential thread interactions
2. Complexity of debugging
3. Complexity of multi-threaded programming
4. Backwards compatibility with existing code

The principle disadvantage is that each thread in the process is now *concurrent*, requiring specialized techniques for ensuring the correctness of the process in operation. With more than one thread of control active in the process at one time, threads can interfere with each other when accessing shared data. Much of this term will be devoted to concurrency and shared data.

6.1.3 XV6 Isn't Multi-Threaded

Since xv6 is a *teaching operating system*, some simplifications were made. Also, multi-threaded operating systems were not common when Sixth Edition UNIX (aka V6), on which xv6 is based, was developed. It is not terribly difficult to add kernel-level threads to xv6 and some schools use it as a project in their undergraduate operating systems class. If you are interested in making this change to xv6, see the instructor at the end of the term.

However, xv6 does use multiple processors which expose similar concurrency issues as multi-threading. The xv6 approach is easier to reason about and also debug (an important consideration!).

6.2 Kernel-Level and User-Level Threads

Does the kernel schedule threads or processes? That's a fairly straightforward question, for which the answer is, *it depends*. Any threads that can be directly scheduled by the operating system – that is, the scheduler is "thread-aware" – are termed *kernel-level* threads (KLTs) and other threads, which are only visible to a user-mode process, and not the kernel, are called *user-level* threads (ULTs).

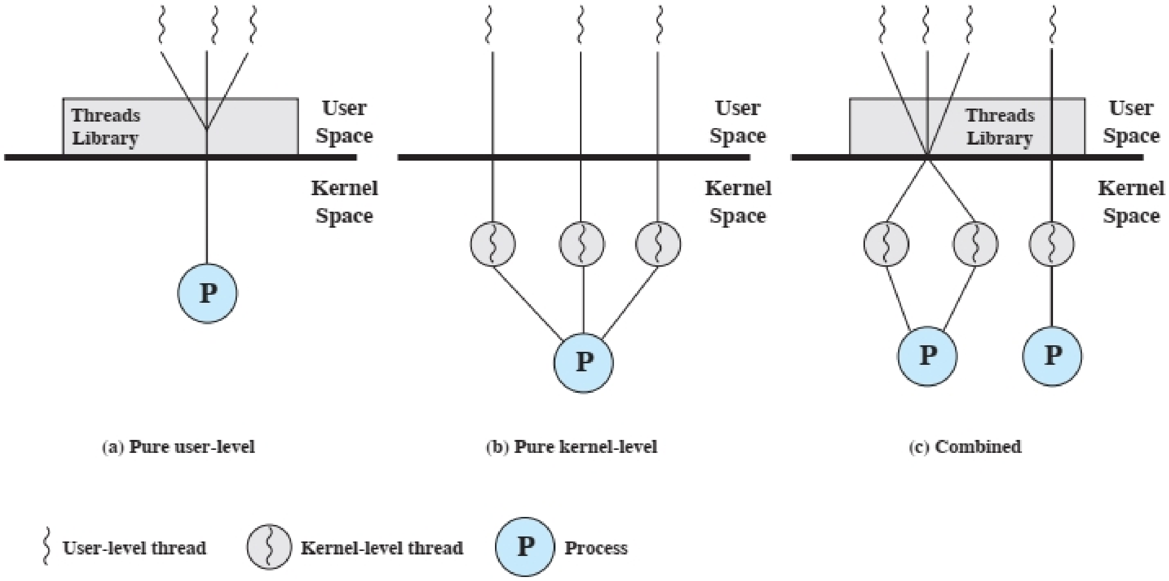


Figure 23: Thread Implementations

Further, a system can have *both* ULTs and KLTs. How individual, or even groups of, ULTs are mapped to KLTs can vary: 1:1, 1:many, many:1 or many:many.

For a pure ULT approach, the user-mode thread library, or the application, must provide a "thread scheduler" that will select the right thread to execute when the kernel scheduler routine selects the process for execution[11].

[11] This would be a very cool project.

6.3 Common Thread Strategies

1. Manager/worker

   ▷ Manager thread handles I/O

   ▷ Manager assigns work to worker threads

   ▷ Worker threads created dynamically or allocated from a *thread pool*

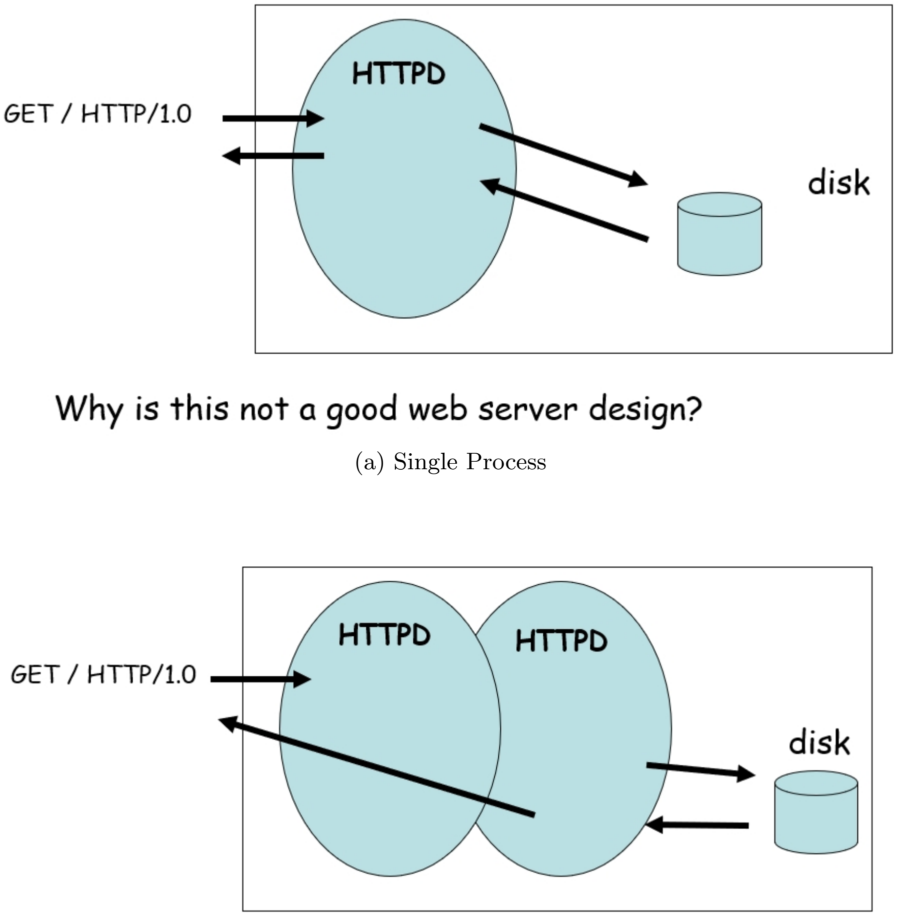
2. Pipeline

   ▷ Each thread handles a different stage of an assembly line

   ▷ Threads hand work off to each other in a *producer-consumer* relationship (like an assembly line)

3. Peer. Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

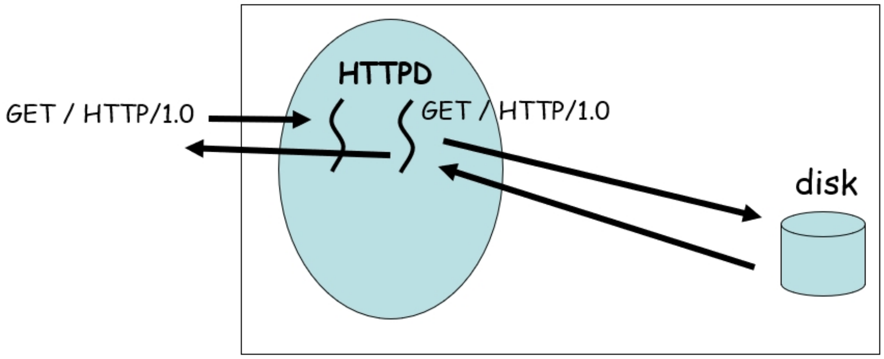
6.4 Processes vs. Threads, An Extended Example



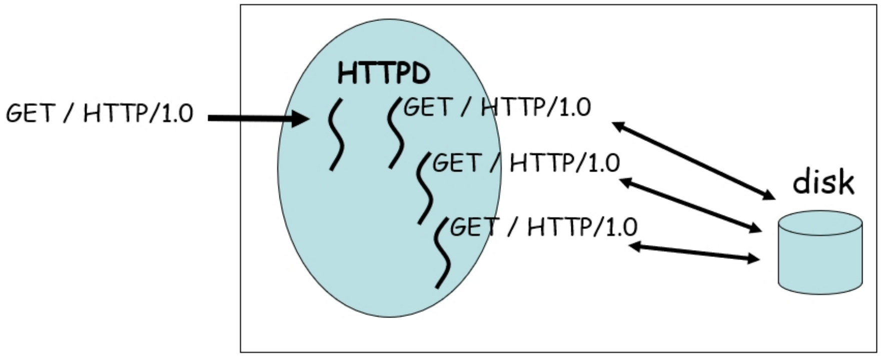
(a) Single Process

(b) Multiple Processes

Figure 24: Web Server Thread Example



(c) Thread per Function



(d) Thread Pool

Figure 24: Web Server *(continued)*

7 Linux Threads Overview

Linux kernel threads map processes (called “tasks”) to kernel threads in a 1:1 mapping. A new task is created with the `clone()` system call (it has a GNU C Library wrapper). Unlike `fork(2)`, `clone()` allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers.

The main use of `clone()` is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

When the child process is created with `clone()`, it executes the function application `fn(arg)`. This differs from `fork(2)`, where execution continues in the child from the point of the `fork(2)` call. The `fn` argument is a pointer to a function that is called by the child process at the beginning of its execution. The `arg` argument is passed to the `fn` function.

When the `fn(arg)` function application returns, the child process terminates. The integer returned by `fn` is the exit code for the child process. The child process may also terminate explicitly by calling `exit(2)` or after receiving a fatal signal[12].

The `child_stack` argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to `clone()`. Stacks grow downwards on all processors that run Linux (except the HP PA processors), so `child_stack` usually points to the topmost address of the memory space set up for the child stack[13].

The low byte of flags contains the number of the termination signal sent to the parent when the child dies. If this signal is specified as anything other than SIGCHLD, then the parent process must specify the __WALL or __WCLONE options when waiting for the child with wait(2). If no signal is specified, then the parent process is not signaled when the child terminates.

There are several flags that can be used with `clone()`:

- CLONE_CHILD_CLEARTID
- CLONE_CHILD_SETTID
- CLONE_FILES
- CLONE_FS
- CLONE_IO
- CLONE_NEWCGROUP
- CLONE_NEWIPC
- CLONE_NEWNET
- CLONE_NEWNS
- CLONE_NEWPID
- CLONE_NEWUSER
- CLONE_NEWUTS
- CLONE_PARENT
- CLONE_PARENT_SETTID
- CLONE_PID
- CLONE_PTRACE
- CLONE_SETTLS
- CLONE_SIGHAND
- CLONE_STOPPED
- CLONE_SYSVSEM
- CLONE_THREAD
- CLONE_UNTRACED
- CLONE_VFORK
- CLONE_VM

Linux dedicates three global descriptor table (GDT) entries for thread-local storage. This information may be accessed with the `get_thread_area()` and `set_thread_area()` system calls. The `get_thread_area()` system call reads the GDT entry indicated by `u_info->entry_number` and fills in the rest of the fields in `u_info`. The `set_thread_area()` system call sets a TLS entry in the GDT.

The Linux documentation on the proper use of `clone()` and its relationship to pthreads is scarce. If you happen to know of a link, please send it to me.

[12] It is not clear if any task in the group will receive a `SIGCHLD` or other signal when the child process exits, or how another thread would detect this thread termination.

[13] It is not clear if the size of the stack can be changed once the new process / thread has been created.

8 POSIX Threads

On many systems, the POSIX threads library is called “pthreads”. An introduction to pthreads on Linux is available on The Geek Stuff.

Programs that have the following characteristics may be well suited for pthreads:

1. Work that can be executed, or data that can be operated on, by multiple tasks simultaneously

2. Block for potentially long I/O waits

3. Use many CPU cycles in some places but not others

4. Must respond to asynchronous events

5. Some work is more important than other work (priority interrupts)

8.1 Native POSIX Threads Library

On Linux, this is the modern Pthreads implementation, aka NPTL. By comparison with LinuxThreads (deprecated, not described here), NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called *1:1 implementations*, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux clone(2) system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux `futex(2)` system call.

With NPTL, all of the threads in a process are placed in the same thread group; all members of a thread group share the same PID. NPTL does not employ a manager thread.

NPTL makes internal use of the first two real-time signals; these signals cannot be used in applications. See nptl(7) for further details.

NPTL still has at least one non-conformance with POSIX.1:

1. Threads do not share a common nice value.

Some NPTL non-conformances occur only with older kernels:

1. The information returned by `times(2)` and `getrusage(2)` is per-thread rather than process-wide.

2. Threads do not share resource limits.

3. Threads do not share interval timers.

4. Only the main thread is permitted to start a new session using `setsid(2)`.

5. Only the main thread is permitted to make the process into a process group leader using `setpgid(2)`.

6. Threads have distinct alternate signal stack settings. However, a new thread’s alternate signal stack settings are copied from the thread that created it, so that the threads initially share an alternate signal stack.

If the stack size soft resource limit (see the description of RLIMIT_STACK in setrlimit(2)) is set to a value other than unlimited, then this value defines the default stack size for new threads. To be effective, this limit must be set before the program is executed, perhaps using the ulimit -s shell built-in command (limit stacksize in the C shell)[14].

8.2 LLNL Tutorial

The Lawrence Livermore National Laboratory (LLNL) POSIX Threads Programming tutorial is a very popular way to learn about multi-threaded programming.

Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads. Threads do not have a parent-child hierarchy as do processes.

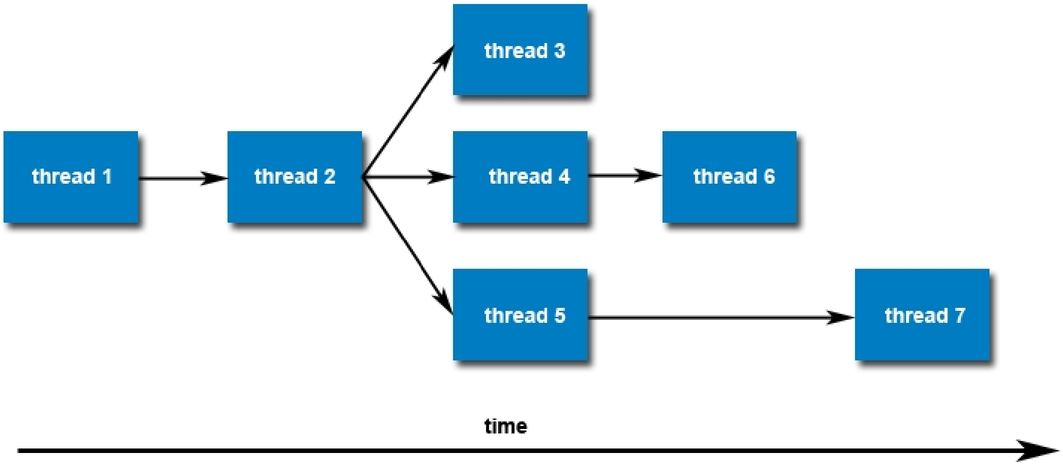


Figure 25: Pthreads Peer Model

Private data includes the thread stack and processor context. The POSIX standard does not dictate the size of a thread's stack. This is implementation dependent and varies. Exceeding the default stack limit is often very easy to do, with the usual results: program termination and/or corrupted data. Stack size is not automatically and dynamically managed. The POSIX standard does not appear to have a way to grow or shrink an existing thread stack once the thread is created.

9 GNU Portable Threads

GNU Pth[15] is a very portable POSIX/ANSI-C based library for Unix platforms which provides non-preemptive priority-based scheduling for multiple threads of execution (aka "multithreading") inside event-driven applications. All threads run in the same address space of the application process, but each thread has its own individual program counter, run-time stack, signal mask and errno variable.

The thread scheduling itself is done in a cooperative way, i.e., the threads are managed and dispatched by a priority- and event-driven non-preemptive scheduler. The intention is that this way

[14] See "man 7 pthreads".

[15] This section is from the GNU Pth manual.

both better portability and run-time performance is achieved than with preemptive scheduling. The event facility allows threads to wait until various types of internal and external events occur, including pending I/O on file descriptors, asynchronous signals, elapsed timers, pending I/O on message ports, thread and process termination, and even results of customized callback functions.

Pth also provides an optional emulation API for POSIX.1c threads ("Pthreads") which can be used for backward compatibility to existing multithreaded applications. See Pth's `pthread(3)` manual page for details.

GNU Pth uses an N:1 mapping to kernel-space threads, i.e., the scheduling is done completely by the GNU Pth library and the kernel itself is not aware of the N threads in user-space. Because of this there is no possibility to utilize SMP as kernel dispatching would be necessary.

The Pth library was designed and implemented between February and July 1999 by Ralf S. Engelschall after evaluating numerous (mostly preemptive) thread libraries and after intensive discussions with Peter Simons, Martin Kraemer, Lars Eilebrecht and Ralph Babel related to an experimental (matrix based) non-preemptive C++ scheduler class written by Peter Simons.

Pth was then implemented in order to combine the non-preemptive approach of multithreading (which provides better portability and performance) with an API similar to the popular one found in Pthread libraries (which provides easy programming).

So the essential idea of the non-preemptive approach was taken over from Peter Simons scheduler. The priority based scheduling algorithm was suggested by Martin Kraemer. Some code inspiration also came from an experimental threading library (rsthreads) written by Robert S. Thau for an ancient internal test version of the Apache webserver. The concept and API of message ports was borrowed from AmigaOS' Exec subsystem. The concept and idea for the flexible event mechanism came from Paul Vixie's eventlib (which can be found as a part of BIND v8).

The newest version is now called nPth. nPth is a library to provide the GNU Pth API and thus a non-preemptive threads implementation.

In contrast to GNU Pth is is based on the system's standard threads implementation. This allows the use of libraries which are not compatible to GNU Pth. Experience with a Windows Pth emulation showed that this is a solid way to provide a co-routine based framework. See the GnuPG download page for more information.

10 ISO C Threads

The GNU C Library Reference Manual (local copy) has information on C threads in chapter 35. For more information on ISO C threads as implemented in the GNU C Library, see the GNU documentation.

11 Reentrancy

Reentrancy is not the same thing as idempotence, in which the function may be called more than once yet generate exactly the same output as if it had only been called once. Generally speaking, a function produces output data based on some input data (though both are optional, in general). Shared data could be accessed by any function at any time. If data can be changed by any function (and none keep track of those changes), there is no guarantee to those that share a datum that that datum is the same as at any time before.

Data has a characteristic called *scope*, which describes where in a program the data may be used. Data scope is either global (outside the scope of any function and with an indefinite extent) or local (created each time a function is called and destroyed upon exit).

Local data is not shared by any routines, re-entering or not; therefore, it does not affect reentrance. Global data is defined outside functions and can be accessed by more than one function call, either in form of global variables (data shared between all functions), or as static variables (data shared by all calls to a given function). In object-oriented programming, global data is defined in the scope of a class and can be private, making it accessible only to functions of that class. There is also the concept of instance variables, where a class variable is bound to a class instance. For these reasons, in object-oriented programming, this distinction of "global" is usually reserved for the data accessible outside of the class (public), and for the data independent of class instances (static).

Reentrancy is distinct from, but closely related to, thread-safety. A function can be thread-safe and still not reentrant. For example, a function could be wrapped all around with a mutex (which avoids problems in multithreading environments), but, if that function were used in an interrupt service routine, it could starve waiting for the first execution to release the mutex.

Reentrancy of a subroutine that operates on operating-system resources or non-local data depends on the atomicity of the respective operations. For example, if the subroutine modifies a 64-bit global variable on a 32-bit machine, the operation may be split into two 32-bit operations, and thus, if the subroutine is interrupted while executing, and called again from the interrupt handler, the global variable may be in a state where only 32 bits have been updated. The programming language might provide atomicity guarantees for interruption caused by an internal action such as a jump or call. Then the function `f` in an expression like `(global:=1) + (f())`, where the order of evaluation of the subexpressions might be arbitrary in a programming language, would see the global variable either set to 1 or to its previous value, but not in an intermediate state where only part has been updated. (The latter can happen in C, because the expression has no sequence point.) The operating system might provide atomicity guarantees for signals, such as a system call interrupted by a signal not having a partial effect. The processor hardware might provide atomicity guarantees for interrupts, such as interrupted processor instructions not having partial effects[16].

There is a short introductory article on reentrancy at embedded. The article contains a nice note on recursion[17].

> No discussion of reentrancy is complete without mentioning recursion, if only because there's so much confusion between the two.
>
> A function is recursive if it calls itself. That's a classic way to remove iteration from many sorts of algorithms. Given enough stack space, this is a perfectly valid-though tough to debug-way to write code. Since a recursive function calls itself, clearly it must be reentrant to avoid trashing its variables. So all recursive functions must be reentrant, but not all reentrant functions are recursive.

[16] See Wikipedia.

[17] However, it does not distinguish between *direct* recursion and *indirect* recursion. You should already be familiar with both forms.

12 Study Questions

1. List and explain 5 main differences between a process and a program.
2. What is meant by the term “process state”?
3. List and define the states in the 5-state process model
4. What is a “process control block”?
5. What is a “process context”?
6. What are the principle data elements of the process context?
7. Where is the context for a process stored when the process is not actively executing in a processor?
8. List and summarize 3 reasons that a process may need to be created.
9. List and explain the system call(s) for process creation.
10. List and explain the system call(s) for new program loading and execution.
11. How is a process transitioned from the “zombie” to “unused” state in xv6?
12. Explain the 5 steps of the context switch process.
13. Explain what is meant by the phrase *a context switch is comprised of two context switches*.
14. What is the purpose of the scheduler routine?
15. What is unique about the `fork()` system call?
16. Why doesn’t the `fork()` system call take an argument?
17. Explain how to interpret the return value from `fork()`.
18. Define the terms *parent process* and *child process*.
19. Differentiate the terms *program* and *process*.
20. List 3 advantages of multi-threading.
21. List 3 disadvantages of multi-threading.
22. What are two differences between the *manager/worker* strategy and *pipeline* strategy for organizing multi-threaded programs?
23. How does the loader know *where* in the memory region of the process to copy the program instructions?
24. What are some of the problems with a single-threaded web server model?
25. List and describe 3 characteristics of a single-threaded program which would tend to indicate that it is a candidate for reimplementation as a multi-threaded program.

26. Explain why all recursive programs are automatically reentrant.

27. Give example pseudo code of a non-recursive but reentrant function.

28. Define the term *idempotent*.

29. Explain why global variables are problematic in a multi-threaded environment.

30. List and explain 3 parts of a process that belong to all threads of control within that process.

31. List and explain why a thread of control *must* have a separate thread context and private data.

32. List and explain the three main ways that user- and kernel-level threads can map to each other.

33. List and explain three common thread management strategies.

34. (T/F) In an operating system without kernel- or user-level threads, global variables are not a problem.

35. Explain why the answer to the previous question is F.

36. Explain the purpose of the *SLEEPING* or *BLOCKED* state.

37. Explain the purpose of the `wait()` system call.

38. List and describe 3 reasons for process termination.

39. Explain why a process executing a system call could be moved to the *SLEEPING* state.

40. For a system with only user-level threads, explain why I/O can be a problem.

41. Why might a system provide multiple event queues?

42. What is meant by the term *voluntary context switch*?

43. What is meant by the term *involuntary context switch*?

44. In the 5-state process model of Figure 13, what is another name for the transition labeled "dispatched"?

45. List and explain the steps taken by the UNIX loader, `execve()`.

46. In the code example for section 3.9, explain the "cool trick".

47. Why doesn't a process track its children in its process control block?

48. (T/F) It is a requirement, in a single-threaded process, that the stack grow downward from the highest address of the process.

49. Why is a separate stack provided for kernel mode?

50. In a system with mixed kernel- and user-level threads, why is it important to map user threads to kernel threads carefully?

Consider the following C program fragment. Assume that system calls and library functions behave as specified in the lecture notes and the xv6 codebase, and that these programs are user programs running in xv6. Assume that fork() always succeeds.

```
int
main(int argc, char **argv)
{
    int x;
    x = 0;
    int ret = fork();
    if(ret == 0) {
        x = x + 1;
    }
    else {
        x = x + 2;
    }
    exit();
}
```

51. When the child calls exit(), what possible values can x have from the child's perspective? Select all that apply:

    (a) 0

    (b) 1

    (c) 2

    (d) 3

52. When the parent calls exit(), what possible values can x have from the parent's perspective? Select all that apply:

    (a) 0

    (b) 1

    (c) 2

    (d) 3

Now, consider this modified version of the program fragment:

```
int
main(int argc, char **argv)
{
    int x;
    x = 0;
    int ret = fork();
    x = x + 1;
    if(ret == 0) {
        x = x + 1;
    }
    else {
        x = x + 2;
    }
    exit();
}
```

53. When the child calls exit(), what possible values can x have from the child's perspective? Select all that apply:

    (a) 0

    (b) 1

    (c) 2

    (d) 3

54. When the parent calls exit(), what possible values can x have from the parent's perspective? Select all that apply:

    (a) 0

    (b) 1

    (c) 2

    (d) 3

Consider the following C program fragment. Assume that system calls and library functions behave as specified in the lecture notes and the xv6 code base, and that these programs are user programs running in xv6. Assume that system calls always succeed.

```
int
main(int argc, char **argv)
{
    int x,y;
    x = 0;
    y = 0;
    int ret = fork();
    if(ret == 0) {
        y = 1;
    }
    else {
        x = 1;
    }
    exit();
}
```

55. When the child calls exit(), what possible values can (x, y) have from the child's perspective? Select all that apply:

    (a) (0, 0)

(b) (0, 1)

(c) (1, 0)

(d) (1, 1)

56. When the parent calls exit(), what possible values can (x, y) have from the parent's perspective? Select all that apply:

    (a) (0, 0)

    (b) (0, 1)

    (c) (1, 0)

    (d) (1, 1)

Now, for this question and the next, consider this modified program fragment:

```
int
main(int argc, char **argv)
{
    int x,y;
    x = 0;
    y = 0;
    int ret = fork();
    if(ret == 0) {
        y = 1;
        exit();
    }
    else {
        x = 1;
    }
    y = y + 1;

    exit();
}
```

57. When the child calls exit(), what possible values can (x, y) have from the child's perspective? Select all that apply:

    (a) (0, 0)

    (b) (0, 1)

    (c) (1, 0)

    (d) (1, 1)

58. When the parent calls exit(), what possible values can (x, y) have from the parent's perspective? Select all that apply:

    (a) (0, 0)

    (b) (0, 1)

    (c) (1, 0)

(d) (1, 1)

Consider this C program fragment. Assume that system calls and library functions behave as specified in the lecture notes and the xv6 code base, and that these programs are user programs running in xv6. Assume that system calls always succeed. Recall that in xv6, printf() takes at least two arguments: the first is a number representing which file descriptor to print to (1 is STDOUT), the second is a string to print (which may include format specifiers), and remaining arguments are of variable length and correspond to the format specifiers in the string. Don't assume any particular guaranteed or consistent time slice length.

```
int
main(int argc, char **argv)
{
    int ret = fork();
    if(ret == 0) {
        printf(1, "child\n");
    }
    else {
        printf(1, "parent\n");
    }
    exit();
}
```

59. Which of the following outputs are possible? Note that we have not enumerated all possible outputs, just some. Select all that apply:

    (a) child
    parent

    (b) parent
    child

    (c) parent
    parent

    (d) child
    child

    (e) parcenht
    ild

For this question, consider this modified version of the program fragment. Assume that wait() always succeeds:

```
int
main(int argc, char **argv)
{
    int ret = fork();
    if(ret == 0) {
        printf(1, "child\n");
    }
    else {
        wait();
        printf(1, "parent\n");
    }
    exit();
}
```

60. Which of the following outputs are possible? Note that we have not enumerated all possible outputs, just some. Select all that apply:

    (a) child
    parent

    (b) parent
    child

    (c) parent
    parent

    (d) child
    child

    (e) parcenht
    ild

Consider the following program fragment. Assume that system calls and library functions behave as specified in the lecture notes and the xv6 codebase, and that these programs are user programs running in xv6. Assume that all system calls succeed. Assume that no loops are optimized out by the compiler, i.e. all code runs exactly as written:

```
int
main(int argc, char **argv)
{
    int i;
    int ret = fork();
    if(ret == 0) {
        for(i=0; i<1000; ++i) { /* do nothing */ ; }
    }
    else {
        wait();
        printf(1, "parent\n");
    }
    exit();
}
```

61. After fork() has returned from the parent's perspective, and before the child has run, what states could the child be in? Select all that apply:

    (a) UNUSED

    (b) EMBRYO

(c) RUNNABLE
(d) RUNNING
(e) SLEEPING
(f) ZOMBIE

62. After fork() has returned from the child's perspective, but before the child calls exit(), what states could the child be in? Select all that apply:

    (a) UNUSED
    (b) EMBRYO
    (c) RUNNABLE
    (d) RUNNING
    (e) SLEEPING
    (f) ZOMBIE

63. After the parent has called wait(), but before wait() has returned from the parent's perspective, what states could the parent be in? Select all that apply:

    (a) UNUSED
    (b) EMBRYO
    (c) RUNNABLE
    (d) RUNNING
    (e) SLEEPING
    (f) ZOMBIE

64. When the child calls exit(), what state is it in? Select one:

    (a) UNUSED
    (b) EMBRYO
    (c) RUNNABLE
    (d) RUNNING
    (e) SLEEPING
    (f) ZOMBIE

65. What state does the child transition to during its call to exit()? Select one:

    (a) UNUSED
    (b) EMBRYO
    (c) RUNNABLE
    (d) RUNNING SLEEPING
    (e) ZOMBIE

66. After wait() returns from the parent's perspective, what states could the child be in? Assume that there are no other processes running in xv6. Select all that apply:

(a) UNUSED
(b) EMBRYO
(c) RUNNABLE
(d) RUNNING
(e) SLEEPING
(f) ZOMBIE

67. After wait() returns from the parent's perspective, what states could the child be in? Assume that there are other processes running in xv6, which could make any system calls, and recall that once a process enters the UNUSED state, it may be subsequently reused. Select all that apply:

(a) UNUSED
(b) EMBRYO
(c) RUNNABLE
(d) RUNNING
(e) SLEEPING
(f) ZOMBIE

---

68. The heap is part of the ____________? Select all that apply:

(a) Process
(b) Program

69. The text section is part of the ____________? Select all that apply:

(a) Process
(b) Program

70. For our purposes, the CPU has two main modes: user and kernel. What are the advantages of CPU virtualization using these two modes? Select all that apply:

(a) Isolation
(b) Protection
(c) Simplification of the programming model for user programs
(d) Management of hardware resources is restricted to privileged instructions only available from kernel mode
(e) User programs can request privileged services via system calls
(f) The kernel is a trusted entity

71. The compilation step that assigns virtual addresses is the ________________________ step.

72. The helper function argint() is used to recover integer arguments to a system call from the kernel stack.

(a) True
(b) False

73. The argptr() helper function checks to ensure that the pointer passed to the system call points to enough memory to hold the requested data.

(a) True
(b) False

74. We often say that a process has code, data, and stack sections. However, there is more information associated with a process. This additional information is called

(a) Totalitarian
(b) Ancillary
(c) Actuarial
(d) Statutory
(e) Ambulatory

75. Data about the process itself is stored in the

(a) Stack frame
(b) CPU context
(c) Process control block
(d) Task frame
(e) Ptable lock

76. Which system call is used to create a new process, once the kernel is up and running?

(a) The wait() system call
(b) The exec() system call
(c) The exit() system call
(d) The fork() system call
(e) The pthread_create() system call
(f) The pthread_cond_wait() system call

77. Which system call does a running process make to overwrite its current program with a new program, and run the new program?

(a) The wait() system call
(b) The exec() system call
(c) The exit() system call
(d) The fork() system call
(e) The pthread_create() system call
(f) The pthread_cond_wait() system call

78. Which routine in xv6 selects the next process to run?

  (a) fork()
  (b) wait()
  (c) yield()
  (d) sched()
  (e) scheduler()
  (f) swtch()

79. A shared data structure that may be accessed by more than one thread of control simultaneously or in alternation is called a ____________ data structure.

  (a) Local
  (b) Virtual
  (c) Abstract
  (d) Concurrent
  (e) Public

80. A multi-threaded program can dynamically adjust to the number of processors at run time.

  (a) True
  (b) False

81. Because a process has a minimum of one stack and one context, a process always contains at least one thread.

  (a) True
  (b) False

82. Thread private data includes ____________. Select all that apply.

  (a) A stack
  (b) A heap
  (c) A list of open files
  (d) A context
  (e) A unique PID

83. The stack grows ____________ on IA32 systems (as described in the lecture notes and xv6).

  (a) Up (towards higher addresses)
  (b) Down (towards lower addresses)

84. The heap grows ____________.

  (a) Up (towards higher addresses)
  (b) Down (towards lower addresses)

85. Once the instruction pointer for the selected process is retrieved from its stack, the context switch process is complete.

  (a) True
  (b) False

13 More Information on Threads

- *POSIX Threads Programming*, Blaise Barney, Lawrence Livermore National Laboratory.
- *Programming with POSIX Threads*, David R. Butenhof, Addison-Wesley, ISBN 0-201-63392-2.
- Yolinux.org pthreads tutorial.
- Chapter 12 of *Computer Systems: A Programmer's Perspective*, 3rd ed., Bryant and O'Hallaron, Prentice Hall, 2015. ISBN-13: 978-0134092669. This is the text used for CS 201, our prerequisite, at PSU.
- *Is Parallel Programming Hard, And, If So, What Can You Do About It?*. Edited by Paul E. McKenney, Linux Technology Center, IBM Beaverton. Not for the beginner!
- Appendix B of *The Little Book of Semaphores*, Allen B. Downey. Thread cleanup with pthreads.