OPERATING SYSTEMS

HEADING: :	**Dark Purple 1, Light Gold 2**	**20**
TYPES:	**Green**	**18**
SUBTYPES:	**Yellow**	**16**
Important Points:	Orange	16
Important Points:	Gold	16
Important Points:	Lime	16

- 10 programs * [6 writeup + 4 viva] = 100
- 30: Reduced to this
- 10: Lab Internals
- 10: Open Ended Lab
- 50: LAB

- 50: THEORY
- 30: CIE
- 20: Exploratory Learning

UNIT 1

Introduction to Operating Systems:
Virtualizing the CPU,
Virtualizing Memory,
Concurrency, Persistence, Design Goals.

Virtualization The Abstraction:
The Process,
Process API ,
Process Creation,
Process States,
Data Structures,
Process API:
- The fork() System Call
- Adding wait() System Call
- exec() System Call
- Other Parts of the API.

Von Neuman model of Computing
- Running Program executes instructions (millions of them)
- Instructions are fetched, decoded, and executed by the PROCESSOR
- Then next instruction

Operating System
- Body of software, that is responsible for running programs
- Allowing you to run many at the same time,
- Allowing programs to share memory
- Enabling programs to interact with devices

The primary way the OS does this is through VIRTUALIZATION.

OS takes a physical resource (processor, memory, disk) and transforms it into a virtual form of itself. Thus, OS is virtual machine.

OS provides interfaces/APIs/standard library/system calls to applications, in order to run programs, access memory, disk.

OS is aka Resource Manager. (Resource: CPU, memory, disk)

[~ OS: controls resources, through virtualization (by APIs) ~]

Virtualization of CPU

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

Let's say we save this file as `cpu.c` and decide to compile and run it on a system with a single processor (or **CPU** as we will sometimes call it). Here is what we will see:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

After 1 second, the code prints the input passed

Let’s run many different instances of this same program

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
```

All 4 programs seems to be running at the same time, even though we have 1 processor. OS creates this illusion of many virtual CPUs.

Turning a single CPU into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call virtualizing the CPU

Virtualization of Memory

Memory is just an array of bytes.

Specifying an address, memory can be read (data stored there). Data can also be written to given address in memory

Memory is accessed all the time when a program is running.
- Program keeps all of its data structures in memory
- accesses them through various instructions (loads, stores)
- Instruction of the program is in memory too

```
int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                   // a1
    assert(p != NULL);
    printf("(%d) address of p: %08x\n",
           getpid(), (unsigned) p);                 // a2
    *p = 0;                                         // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

Figure 2.3: **A Program that Accesses Memory**

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

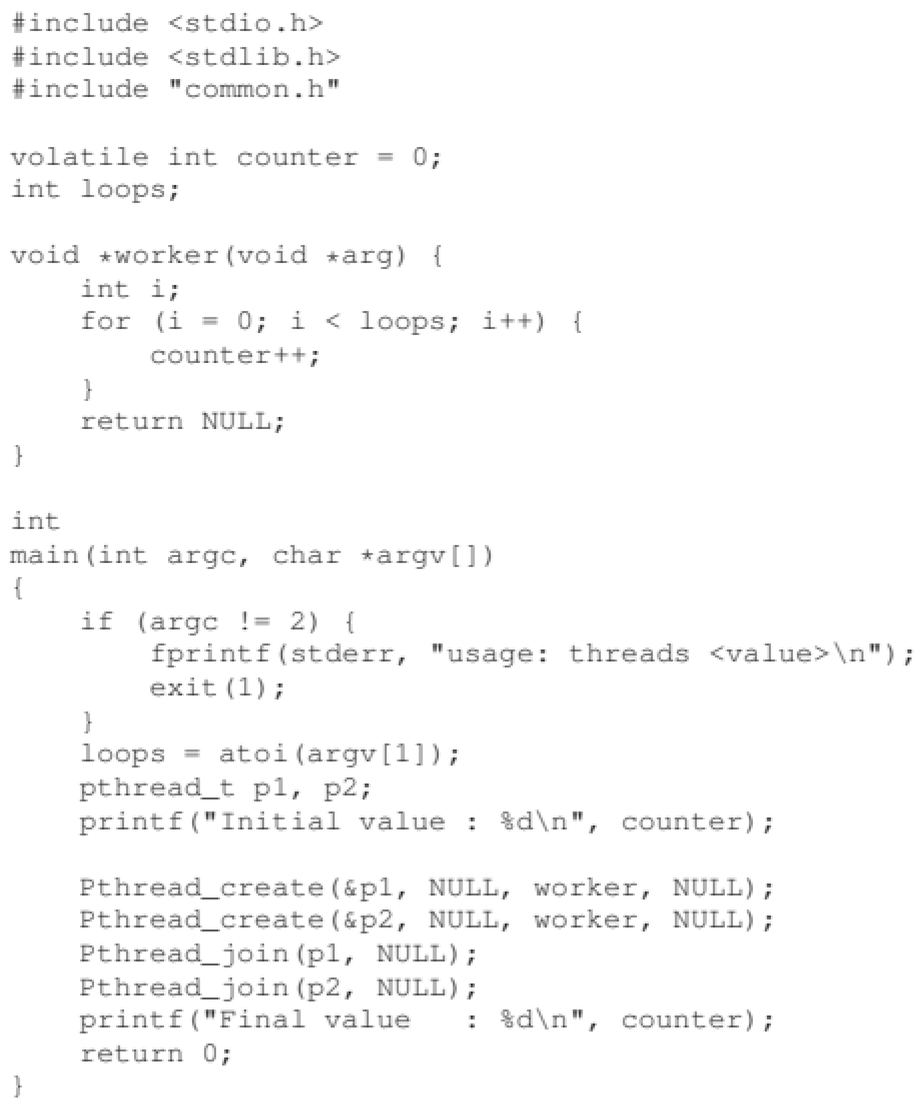
Each process accesses its own private virtual address space
(aka address space), which the OS maps onto the physical memory

It is as if each running program has its own private memory

CONCURRENCY

Concurrency problems arise, when working on many things at once (concurrently) in the same program.

OS is running many processes (concurrency), ./cpu A, ./cpu B,...



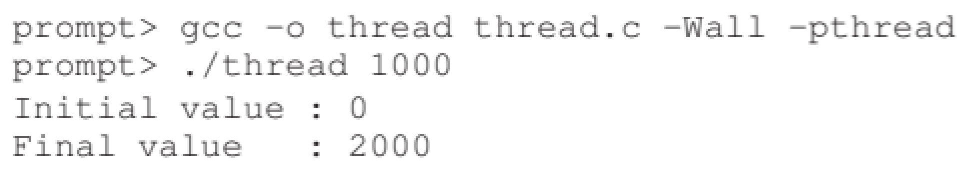
```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Figure 2.5: **A Multi-threaded Program**

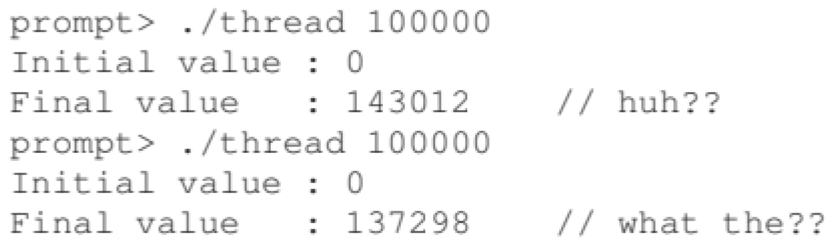
Multithreaded programs exhibit the same problem

Thread is a function running within the same memory space as other functions, with more than one of them active at a time.

In this eg, each thread starts running in a routine called worker()



```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```



```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

Reason for this behaviour:
Because of how instructions are executed, which is one at a time.
Shared Counter takes three instructions,
- load the value of the counter from memory into a register
- increment it
- store it back into memory

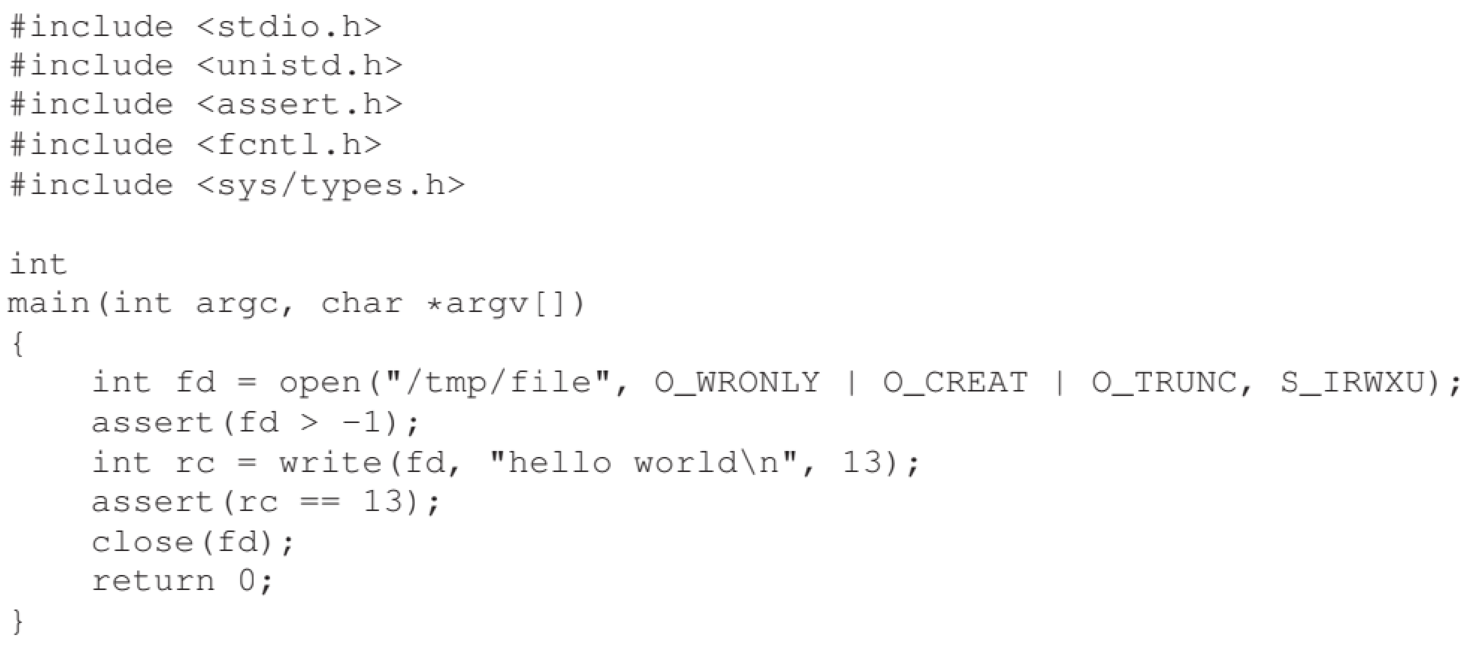
These three instructions do not execute atomically (all at once), strange things can happen (race condition, both threads trying to access the same shared memory)

PERSISTENCE

- System Memory (DRAM), store values in volatile manner
- Data in memory is lost when power goes off
- Hard Drive / SSD are repository for persistent storage (through I/O)

File System: Software in the OS that manages the disk

- OS does not create virtualization for disk
- Instead, Files in disk is shared between different processes
- prgm.c (shared between processes: file editor, compiler)



```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

Figure 2.6: **A Program That Does I/O**

Open(): Opens the file and creates it
Write(): Writes data to file
Close(): Closes the file

These system calls are routed to a part called File System

Write protocols: Journaling, Copy-on-Write to improve performance

DESIGN GOALS

what OS actually does: it takes physical resources (CPU, memory, or disk) and virtualizes them. It handles tough and tricky issues related to concurrency. And it stores files persistently.

We need to have GOALS to design and implement, and make necessary trade-offs

Abstraction

- Build up abstraction to make system convenient and easy to use
- APIs and Systems calls

- Write program in High level language (C)
- Write code in Assembly
- Build processors using Logic Gates
- Transistors

Performance

- Provide high performance
- Minimize the overheads
- Virtualization and other features without excessive overheads
- Overhead: in time (more instructions), in space (more memory)

Protection

- Protection between the applications, OS and applications
- Programs shouldn't interfere with other programs, and the OS
- ISOLATING processes from one another is the key to protection

Reliability

- OS must run non stop, because all applications running will fail

Energy Efficiency (green world)
Security (extension of protection)
Mobility (on mobiles)

Virtualization The Abstraction

Virtualizing the CPU
technique of Time Sharing,
allows users to run concurrent processes

To implement virtualization of CPU
- Mechanisms (low level machinery)
- Policies (high level intelligence) (algorithms making some decision)

Time Sharing Mechanism
Context Switch is a time sharing mechanism in which OS stops a program and starts another program

Scheduling Policy
Policy that determines which program to run, given a set of possible programs. Decision is made using historical information, knowledge

The Process
- Fundamental abstraction that OS provides
- It is a running program
- Program: static data, bunch of instructions, sitting on the disk
- Process: OS takes the program and gets it running

We can summarize the process by taking an inventory of system it accesses or affects during the course of its execution.

Machine state that comprises a process is,

Memory
- Instructions, Data the program reads/writes is in memory
- Address Space: Memory that the process can address

Register
- Registers are read or updated as the instruction is executed
- Program Counter (PC): Instruction being executed
- Stack Pointer: Manages stack for function parameters, local variable

I/O Information
- For programs accessing disk storage
- Info can be list of files the process currently has open

Process API

Create
- To create new processes
- Command into a shell, Double click an application icon
- OS is invoked to create a new process to run that programs

Destroy
- To destroy processes forcefully
- Many processes will run and exit by themselves when complete
- However, user may wish to wish to kill them
- So, interface to halt a process is useful

Wait
- It is useful to wait for a process, to stop running
- Synchronization, Dependency Management

Miscellaneous Control
- There are methods to suspend a process, then resume it after

Status
- There are interfaces to get some status information about a process

Process Creation

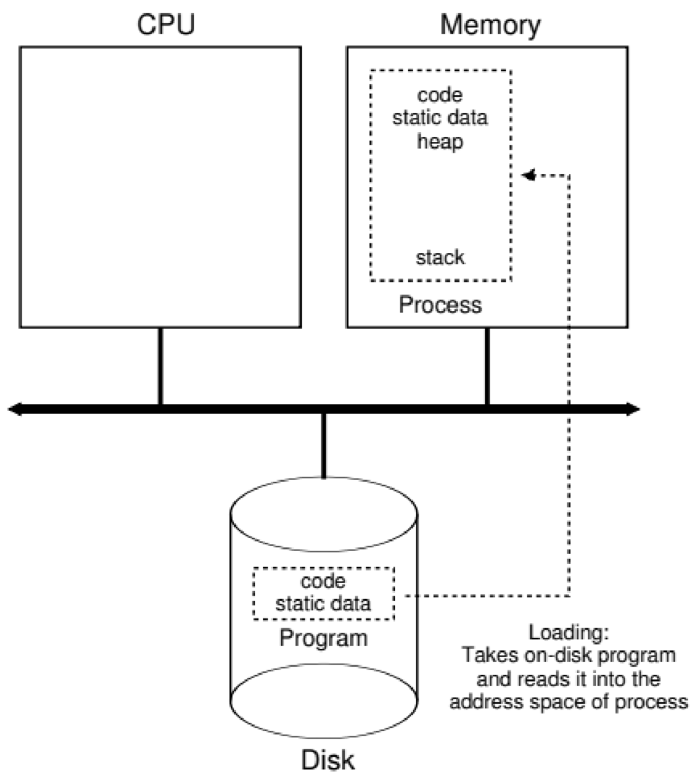


Figure 4.1: **Loading: From Program To Process**

- OS loads the code and static data from disk to memory
- It is stored in the “address space” of the process
- Programs initially reside on disk (harddrive, ssd,) in executable form
- Loading a program requires OS to read those bytes into memory

- Loading process was done eagerly
- (all data is loaded at once before running the program)
- Loading process is done lazily
- (data or code are loaded as they are needed for program execution)

- Once loaded, some memory is to be allocated for runtime stack
- Programs use stack: local var, fn parameters, return addresses
- also stack for arguments (argc and argv array)
- OS also allocates memory for heap (for dynamic memory)

- OS will do initialisation tasks related to I/O
- 3 default fd for a process: Input, Output, Error

Starts the program running at the entry point (main())
OS transfers control to CPU

Process States

Different states a process can be in at a given time

Running

- Process is running on a processor
- It is executing instructions

Ready

- Process is ready to run
- But OS has chosen not to run it at that moment

Blocked

- Process has performed some operation, that makes it
- not ready to run until some other event takes place
- Eg: Process initiates an I/O request to disk becomes blocked

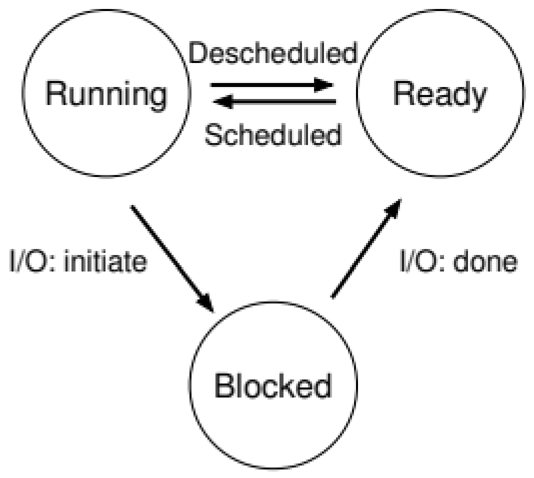


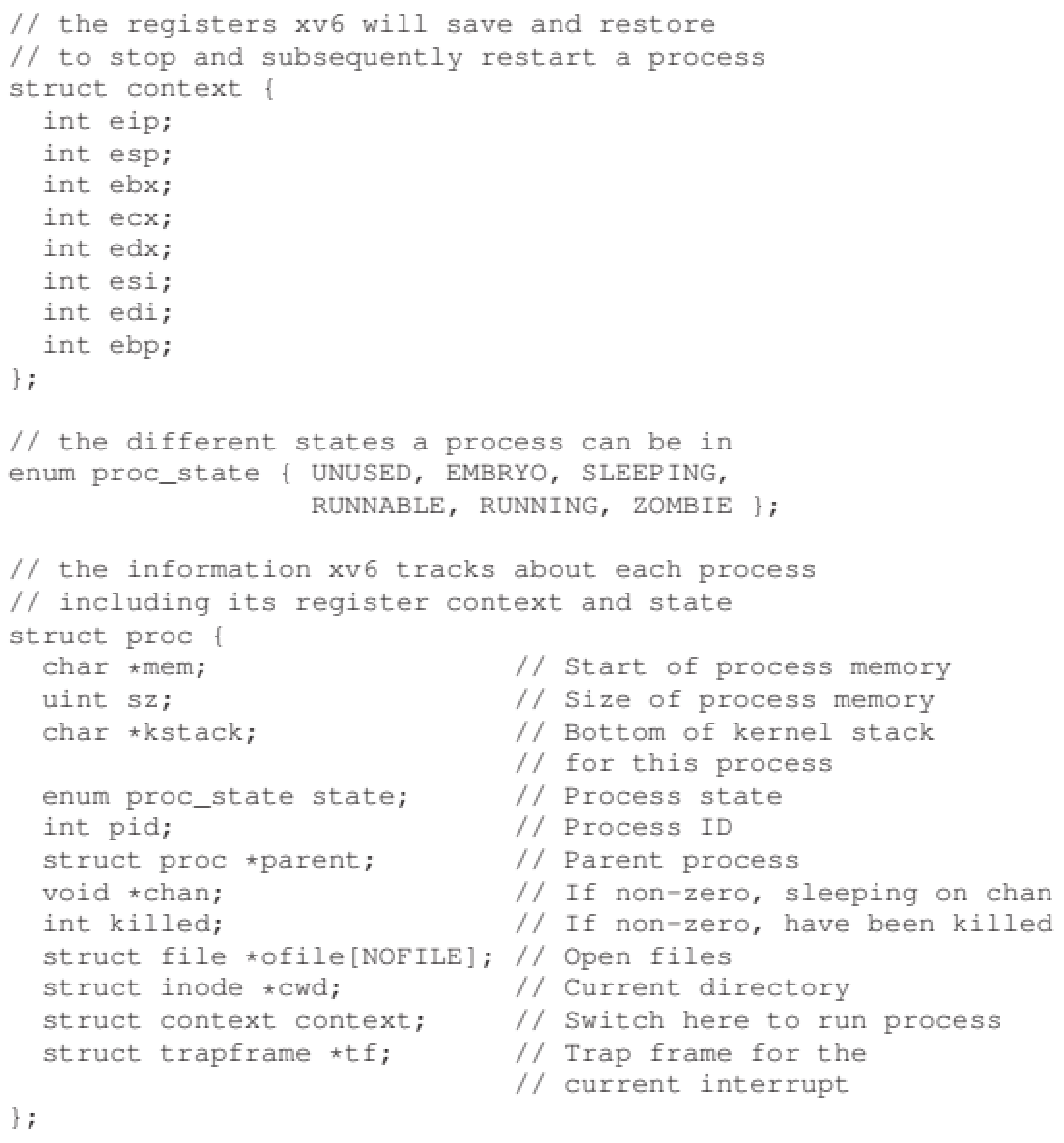
Figure 4.2: **Process: State Transitions**

Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process$_0$ initiates I/O
4	Blocked	Running	Process$_0$ is blocked,
5	Blocked	Running	so Process$_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process$_1$ now done
9	Running	–	
10	Running	–	Process$_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Data Structures

- OS is a program, and it has DS to track various information
- To track state of each process (process List)



```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent 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
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.3: **The xv6 Proc Structure**

The register “context” will hold the contents of a stopped process’s registers. The registers will be saved to memory, so they can be restored later to resume running

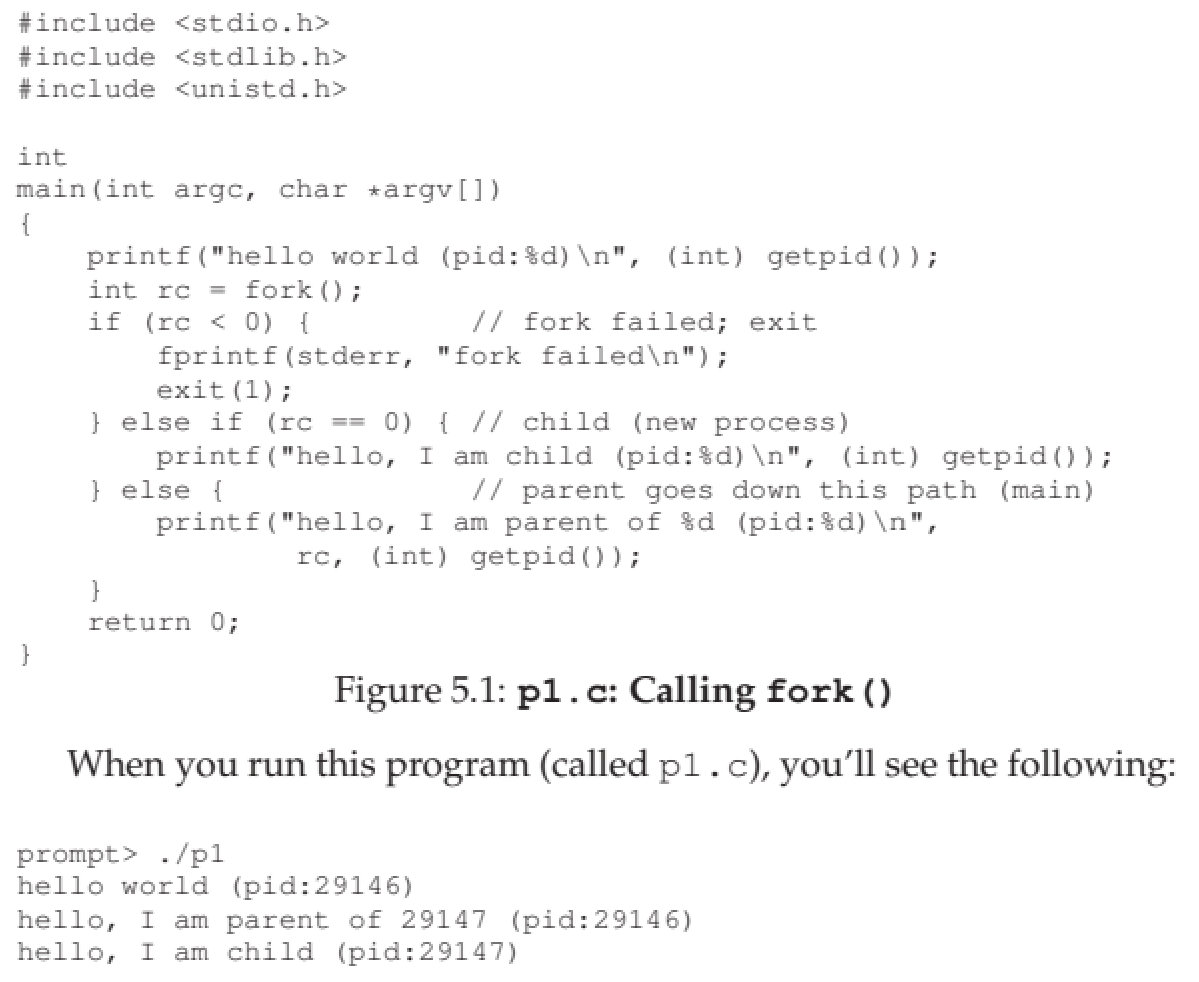
Process state: R, R, B. In addition,
Initial state when process is created
Final State when it has exited but not cleaned (Zombie state)

Process Control Block (PCB): Individual structure that stores information about a process

Process Creation API

fork() system call

Used to create a new processes
Process created is an almost exact copy of the calling process



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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}
```

Figure 5.1: **p1.c: Calling fork()**

When you run this program (called p1.c), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
```

Process is running (PID = 29146)
fork() system call to create a new process by OS
Now TWO exact copies of the process exist
Both are returned from fork()
So child process doesn't start running at main()
Rather at, int rc = fork()

Parent: int rc = fork() = PID of child = 29147
Child: int rc = fork() = 0
This differentiation in return value is useful to handle both the cases

Output is not deterministic
CPU Scheduler determines which process runs at a given time

wait() system call

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

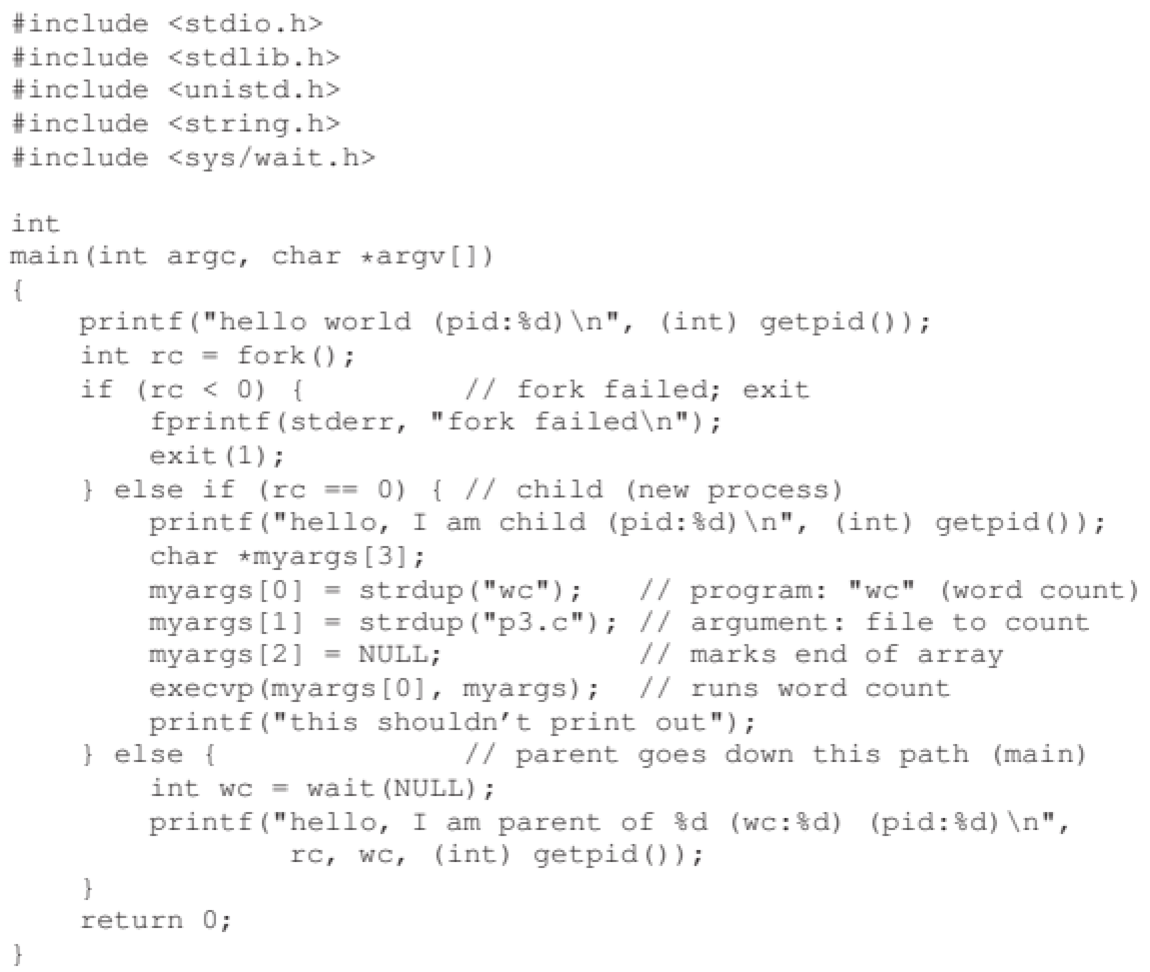
Figure 5.2: **p2.c: Calling fork() And wait()**

- It is useful for PARENT to wait for CHILD process to finish
- Makes the output deterministic
- PARENT calls wait() which delays its execution until child finishes
- When CHILD is done, wait() returns to the parent

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

exec() system call

- Useful to run a program that is different from the calling program
- fork runs copies of same program, exec runs different program
- 6 variants: execl(), execle(), execlp(), execv(), execvp()



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

Figure 5.3: **p3.c: Calling fork(), wait(), And exec()**

Given the name of the EXECUTABLE (wc) and arguments (p3.c) exec() loads "wc" code into memory overwriting the current code. Then OS runs this process "wc p3.c"

Thus OS doesn't create new process, rather transforms the current program (p3) into a different program (wc)

After exec(), its almost as if p3.c never ran
Successful call to exec() never returns

* execl("wc", /myfile.c, "cmd line arguments");

UNIT 2

Advanced Multiprocessor Scheduling

Background Multiprocessor Architecture,
Synchronization,
Cache Affinity,
Single- Queue Scheduling,
Multi-Queue Scheduling,
Linux Multiprocessor Schedulers,
Real time scheduling,
Deadline scheduling and rate monotonic scheduling,
Priority inversion

Advanced Multiprocessor Scheduling

Background: Multiprocessor Architecture

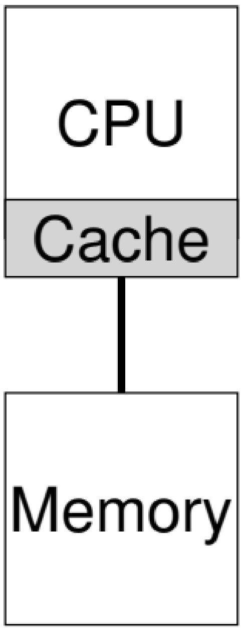


Figure 10.1: **Single CPU With Cache**

In a single CPU system,
- There is a hierarchy of hardware caches
- Caches are small, fast memories
- They hold copies of popular data from the main memory
- Caches are based on temporal and spatial locality
- Temporal: When a data is accessed, it is likely to be accessed again
- Spatial: When a data at x is accessed, data near x will be accessed

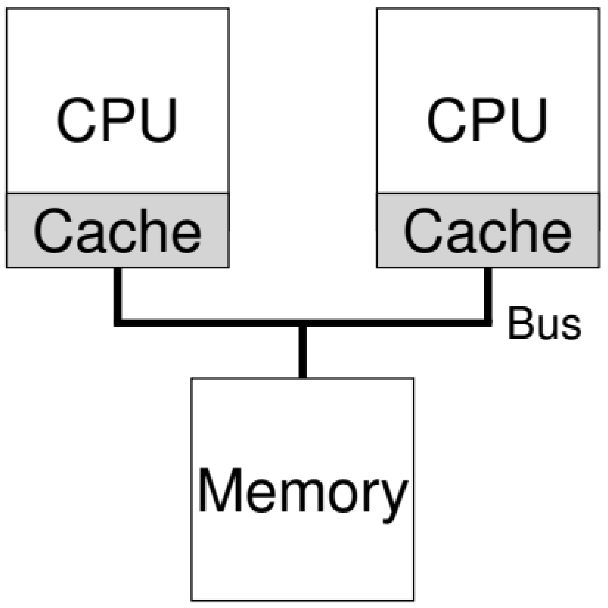


Figure 10.2: **Two CPUs With Caches Sharing Memory**

Caching with multiple CPUs can lead to data inconsistency issues.

Cache Coherence Problem

CPU 1 reads data at address A from main memory
CPU 1 stores it in its cache
CPU 1 updates the value
But it doesn't write it to main memory yet
CPU 2 reads data at address A from main memory (old value)

Solution: Hardware Monitoring
Hardware ensures that the "right thing" happens to maintain a single shared memory view.

Bus snooping

Monitoring the memory updates on the bus bw caches and memory.
When a CPU sees an update, it either:
- Invalidates its cache copy (removes it).
- Updates its cache copy with the new value.

Synchronization

Even with cache coherence, programs/OS need to ensure correct access to shared data.

Mutual exclusion primitives (locks) are needed for synchronization and to ensure correctness when updating shared data.

Without locks, concurrent access to shared data can lead to problems
Example: Shared Linked List
Two threads on different CPUs try to remove an element from a shared linked list. Both threads may try to remove the same element, causing issues.

Solution: Locking
Use locks (mutex). Locking ensures only one thread can access the shared data at a time.

Performance Issues
Locking can lead to performance problems as the number of CPUs grows. Access to data structures can become slow.

Cache Affinity: **A Multiprocessor Scheduling Issue**

- When a process runs on CPU, it builds up state in the CPU's caches

- If the process is run again on the same CPU,
  it will run faster due to the existing cache state.

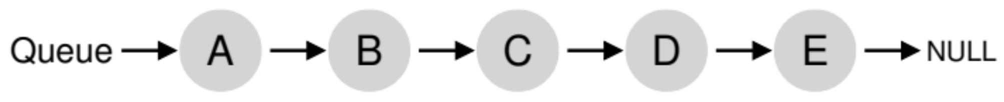
- If the process is run on a different CPU,
  it will run slower due to cache reloads.

Implication for Scheduling
A multiprocessor scheduler should consider cache affinity. Preferably, keep a process on the same CPU to maximize performance.

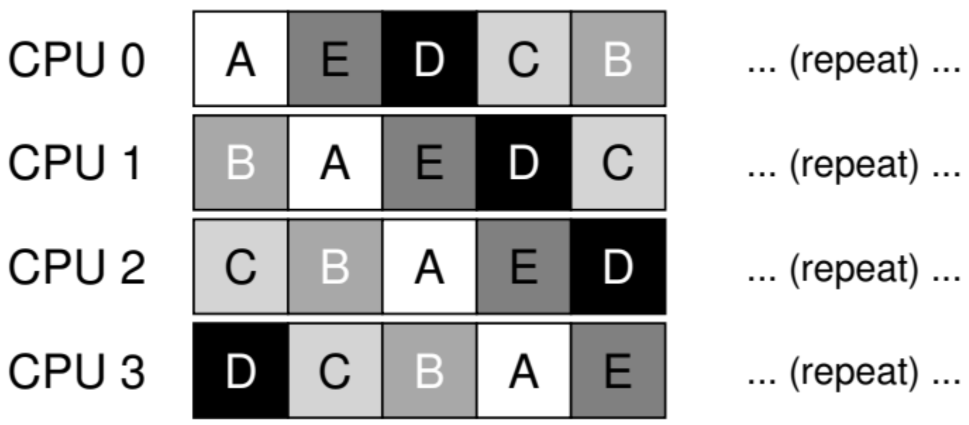
Cache coherence protocols ensure correctness,
Cache affinity optimizes performance.

In essence, cache affinity is about minimizing cache reloads by scheduling processes to run on the same CPU where their cache state is already present, thereby improving performance.

Single Queue Scheduling



Over time, assuming each job runs for a time slice and then another job is chosen, here is a possible job schedule across CPUs:


CPU 0 A E D C B ... (repeat) ...
CPU 1 B A E D C ... (repeat) ...
CPU 2 C B A E D ... (repeat) ...
CPU 3 D C B A E ... (repeat) ...

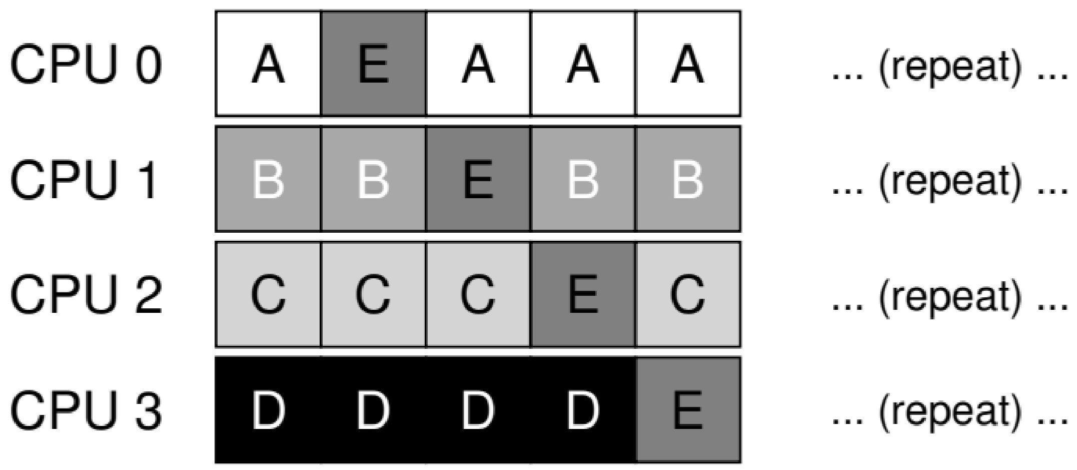
It simplifies multiprocessor scheduling by using a single job queue

+ Easy to implement
+ Can adapt existing single-CPU scheduling policies

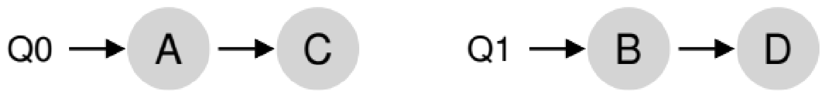
Lack of scalability: Requires locks to ensure correctness. Locks can reduce performance as CPUs grow

Poor cache affinity: Jobs may switch CPUs since they are picked from a shared queue, degrading cache performance

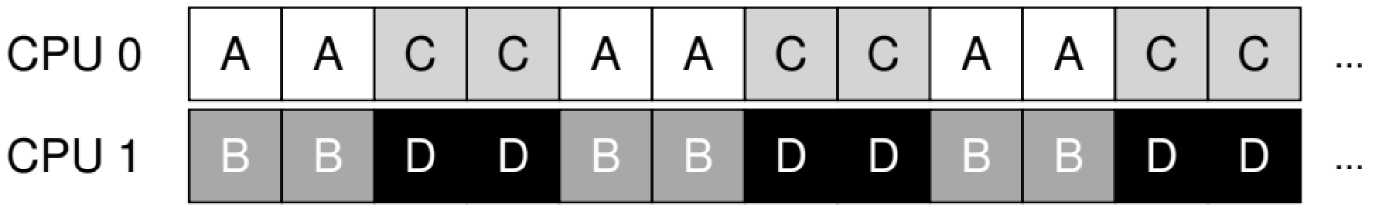
Addressing Cache Affinity
Introduce affinity mechanisms to keep jobs on the same CPU if possible. Balance load by moving some jobs between CPUs


CPU 0 A E A A A ... (repeat) ...
CPU 1 B B E B B ... (repeat) ...
CPU 2 C C C E C ... (repeat) ...
CPU 3 D D D D E ... (repeat) ...

Multi-Queue Scheduling



Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:



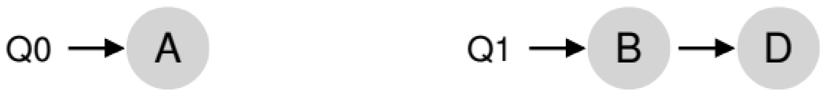
- It overcomes the problems in SQMS
- It uses multiple queues (one job queue for each CPU)
- Each queue follows a scheduling discipline (RR, FIFO,..)

- When a job enters the system, it is placed on exactly one queue (based on some heuristic)

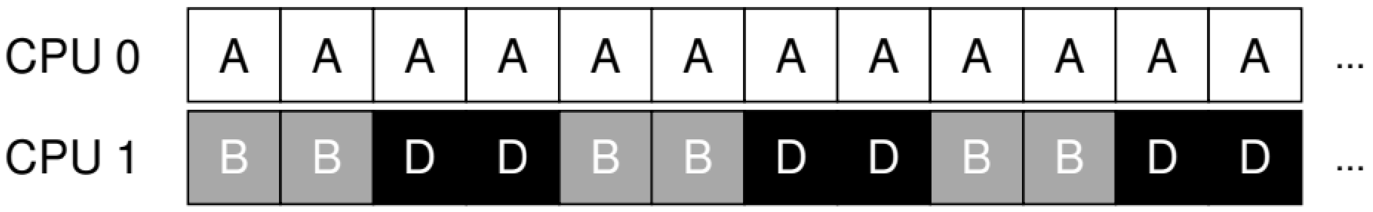
- This avoids the problems of synchronization found in SQMS

+ Scalable: As CPU grows, Queues will grow
+ Provides Cache Affinity

- Load Imbalance



If we then run our round-robin policy on each queue of the system, we will see this resulting schedule:



A gets 2x CPU time, while B and D gets 1x CPU time

Migration:
When Queue 1 is empty, the OS will move jobs to it

Work Stealing:
When Queue 1 is lesser than other Queues, Queue 1 will peek at the other queue and if it has more jobs, it will steal those jobs. But too often peeking will cause performance overhead

Linux Multiprocessor Schedulers

O(1) scheduler
- Multiple Queues
- Priority Based Scheduler (like MLFQ)
- Changing priority of a job over time

Completely Fair Scheduler (CFS)
- Multiple Queues
- Proportional-share approach

BF Scheduler (BFS)
- Single Queue
- Proportional-share approach
- Earliest Eligible Virtual Deadline First (EEVDF)

Real time scheduling,
Deadline scheduling and rate monotonic scheduling,
Priority inversion

UNIT 3

Threads API
Thread Creation, Thread Completion, Locks, Condition Variables.

Locks
The Basic Idea, Pthread Locks, Building A Lock, Evaluating Locks, Controlling Interrupts, Test And Set, Building A Working Spin Lock, Evaluating Spin Locks, Compare-And-Swap, Load-Linked and Store-Conditional, Fetch-And-Add, Using Queues, Different OS, Different Support, Two-Phase Locks

Lock-based Concurrent Data Structures
Concurrent Counters, Concurrent Linked Lists, Concurrent Queues, Concurrent Hash Table

UNIT 4

Processes in Linux:

Processes, Lightweight Processes, and Threads, Process Descriptor, Process Switch, Creating Processes, Destroying Processes.

Process Scheduling:

Scheduling Policy, The Scheduling Algorithm, Data Structures Used by the Scheduler.

Process Descriptor

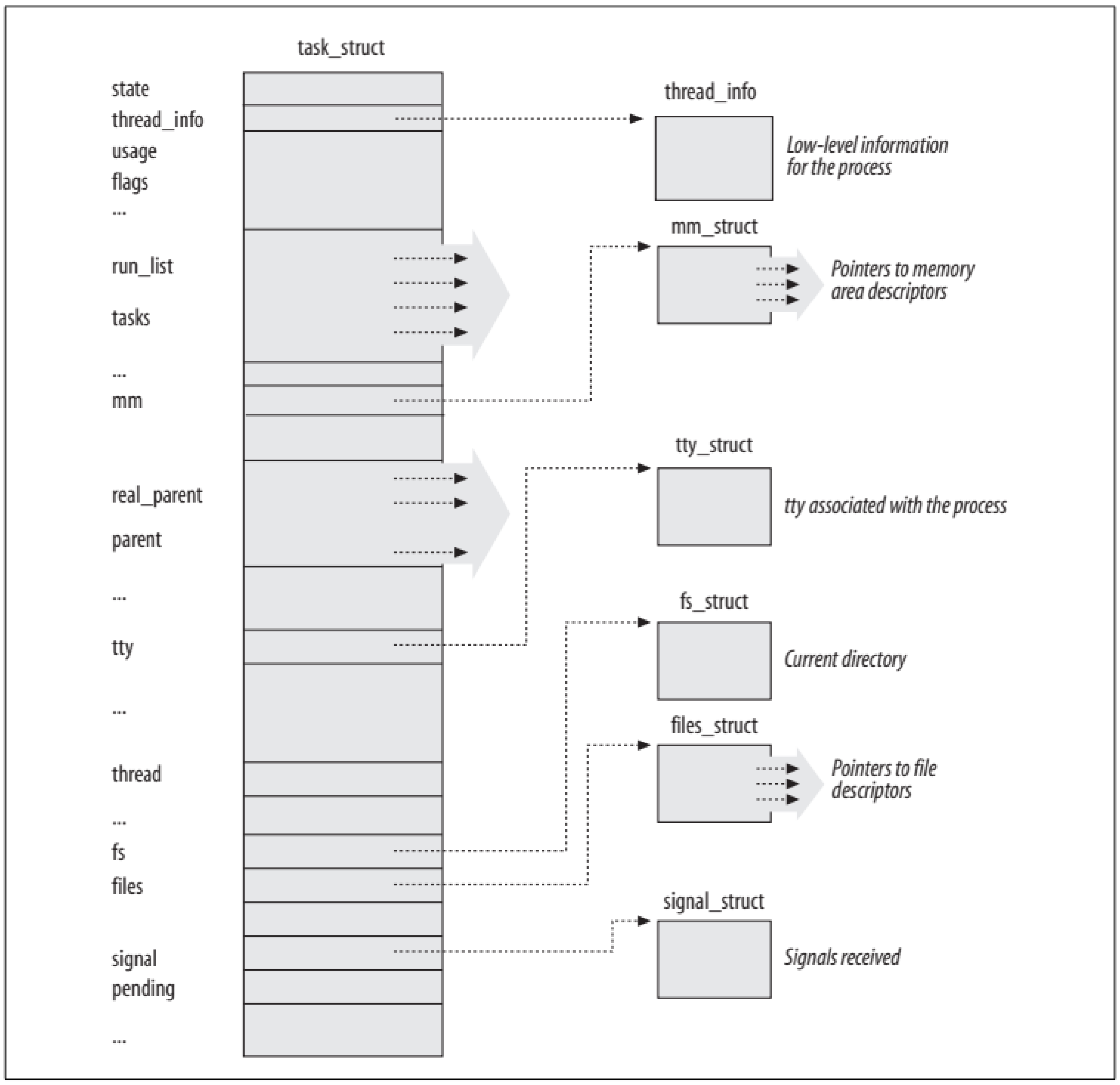
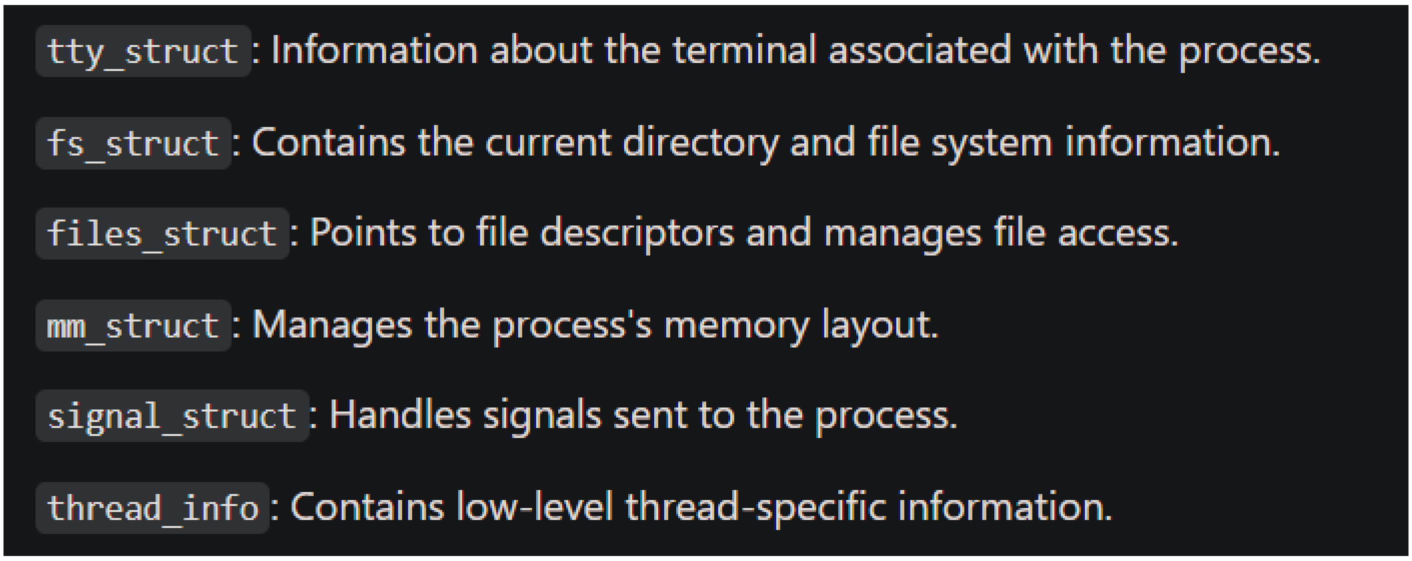


Figure 3-1. The Linux process descriptor

The Linux kernel uses a complex data structure called the process descriptor to manage and keep track of all aspects of a process.

The data structure: `task_struct` stores essential process information
- Process priority, current state, scheduling details,
- Pointers to: address space, files, signal handlers,
- Parent/Child relationship



`tty_struct`: Information about the terminal associated with the process.

`fs_struct`: Contains the current directory and file system information.

`files_struct`: Points to file descriptors and manages file access.

`mm_struct`: Manages the process's memory layout.

`signal_struct`: Handles signals sent to the process.

`thread_info`: Contains low-level thread-specific information.

Process State

TASK_RUNNING:
The process is either currently executing on a CPU
The process is ready to execute.

TASK_INTERRUPTIBLE:
The process is asleep
The process is waiting for a specific condition (signal) to wake it up so it can resume running.

TASK_UNINTERRUPTIBLE:
The process is asleep
The process cannot be woken up by signals.

TASK_STOPPED:
The process has been stopped, usually due to receiving a signal like `SIGSTOP` or `SIGTSTP`.

TASK_TRACED:
The process is being traced by a debugger.
It is stopped and monitored, typically due to a debugging operation.

2 terminal states after a process has finished execution:

EXIT_ZOMBIE:
The process has terminated, but its parent has not yet collected its exit status via `wait` call. The process descriptor remains in this state until the parent collects the exit status.

EXIT_DEAD:
The process has been removed from the system after the parent has collected its exit status. The kernel can now clean up and discard the process descriptor.

Relationship among Processes

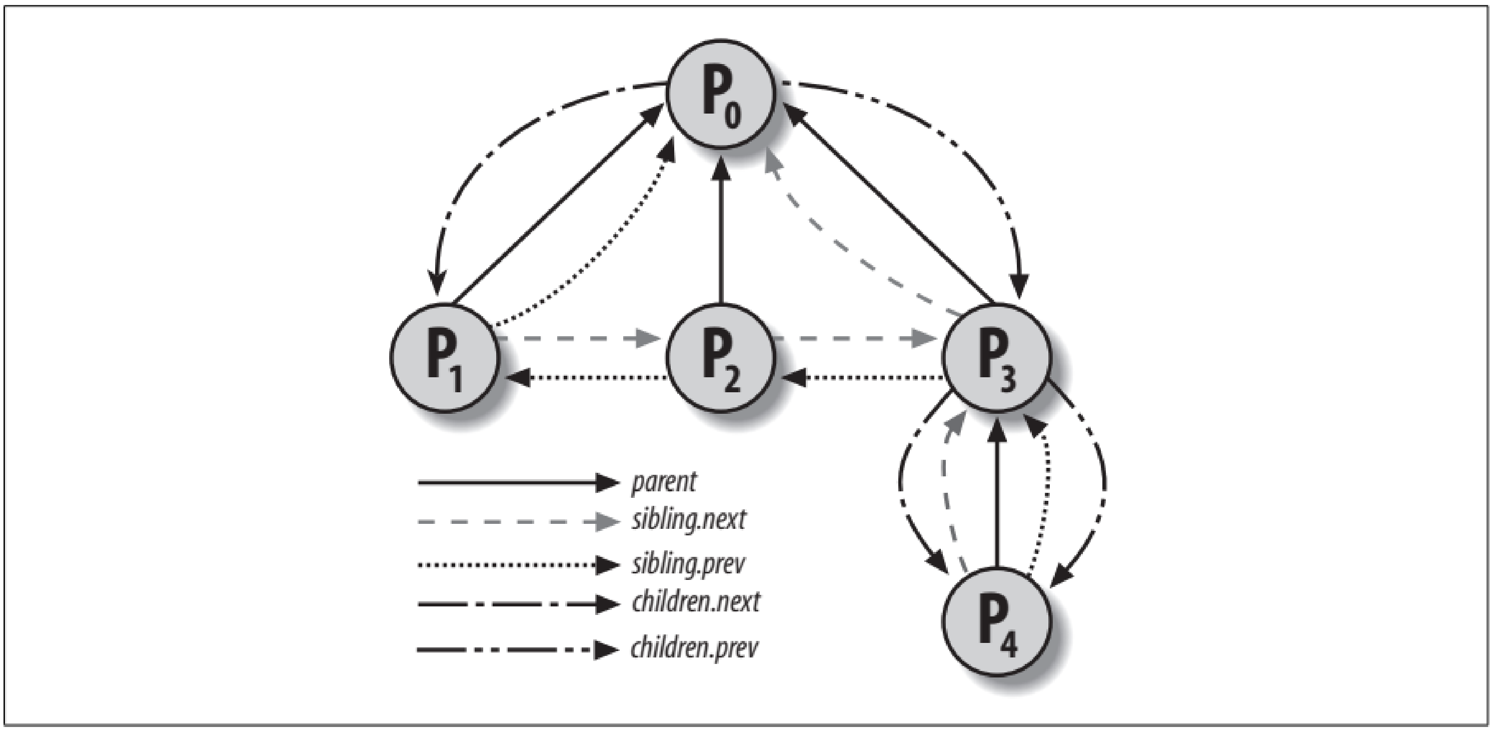


Figure 3-4. Parenthood relationships among five processes

Process Hierarchy:

Parent Process: The process that creates another process.
Child Process: A process created by another process.
Sibling Processes: Processes that share the same parent.

Special Processes:

Process 0 and 1 are created by the kernel;
Process 1, known as init, is the ancestor of all other processes.

Process Descriptor Fields:

These fields help represent the parent/child relationships and other process relationships.

`real_parent:`

Points to the process that originally created the process `P`.
If the original parent no longer exists it points to process 1 (`init`).

`Parent:`

Points to the current parent process of `P`.
This is the process that will be notified if `P` terminates.
It usually matches `real_parent`

`Children`:
A list containing all child processes created by `P`.
This list allows efficient management of descendant processes of `P`.

`Sibling`:
Pointers to the next and previous processes of `P` in the list of that share the same parent. This maintains linked list of sibling processes

x

UNIT 5

System Calls:
POSIX APIs and System Calls,
Parameter Passing,
Kernel Wrapper Routines.

Kernel Synchronization:
How the Kernel Services Requests,
Synchronization Primitives,
Synchronizing Accesses to Kernel Data Structures,
Examples of Race Condition Prevention

POSIX APIs and System Calls

API: Function definition to obtain a service
System Call: Explicit request to the kernel

Parameter Passing

System calls require parameters to be passed from User Mode to Kernel Mode.

Due to the mode switch, stack cannot be used to store parameters.
Instead: stored in CPU registers (eax, ebx, ecx, edx, esi, edi, ebp)

The kernel copies these parameters onto the Kernel Mode stack before invoking the system call service routine.

Key Points:
- Parameters must fit in a 32-bit register
- Larger parameters are passed by reference.
- No more than six parameters can be passed directly
- Additional parameters are passed through pointer
- Service routines return values by writing them to the eax register.

Example Service Routines:
sys_write(unsigned int fd, const char *buf, unsigned int count)

Kernel Wrapper Routines

Functions whose purpose is to issue a system call
They return integer value 0 or -1 indicating the status

Linux provides macros _syscall0 to _syscall6 to simplify declarations of kernel wrapper routines

Key Features:
- Macros generate wrapper routines for system calls with 0-6 params
- Cannot be used for > 6 parameters and non standard return values
- Each macro requires 2 + 2 × n parameters
- First 2: return type, name of system call
- Other 2 x n: data type, name of parameter of system call
- Generates assembly code for system call invocation

Example: _syscall3 for write()

```
_syscall3(int,write,int,fd,const char *,buf,unsigned int,count)
```

Generated Assembly Code:
Loads parameters into CPU registers (ebx, ecx, edx)
Invokes system call with int $0x80
Checks return code and sets errno if error
Returns error code (-1) or success value

They allow kernel threads to invoke system calls without using library functions. These macros simplify the process of generating wrapper routines.