1 Linking

Linking is the process of collecting and combining various pieces of code and data into a single file that can be *loaded* (copied) into memory and executed. Programs are translated and linked using a compiler driver: `gcc -Og -o out main.c sum.c`.

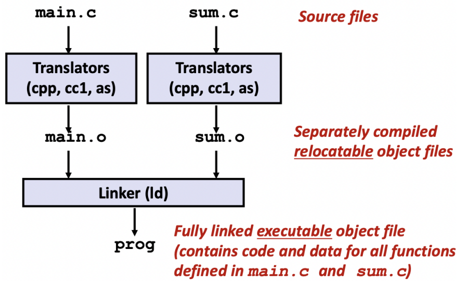


Figure 1. The static linking.

1.1 Why linkers?

- **Modularity.** Program can be written as a collection of smaller source files. Can build libraries of common functions.
- **Time efficiency.** Separate compilation. Change one source file, compile, and then relink.
- **Space efficiency.** Running memory images contain only code for the functions we actually use, rather than a whole library.

1.2 What do linkers do?

1.2.1 Symbol resolution

Programs define and reference *symbols* (global variables and functions). Symbol definitions are stored in object file by assembler in *symbol table*, array of `struct` which includes name, size, and location of each symbol. During symbol resolution, the linker associates each symbol reference with exactly one symbol definition.

1.2.2 Relocation

- Merge separate code and data sections into single section.
- Relocate symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
- Update all references to symbols to reflect their new positions.

1.3 Object files (Modules)

1.3.1 Three kinds of object files

- **Relocatable object file** (`.o` file): Produced from one source (`.c`) file. Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
- **Executable object file** (`.out` file): Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file** (`.so` file): Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time. Called *dynamic link libraries* (DLLs) in Windows.

1.3.2 Executable and linkable format (ELF)

ELF binary is the standard binary format for object files. It is one unified format for `.o`, `.out`, and `.so` files.

- Elf header: word size, byte ordering, file type (`.o`, `exec`, `.so`), machine type
- Segment header table: page size, virtual addresses memory segments (sections), segment sizes.
- `.text`: code
- `.rodata`: read only data, *e.g.*, jump tables
- `.data`: initialized global variables
- `.bss` (better save space): uninitialized global variables. Has section header but occupies no space
- `.symtab`: symbol table, procedure and static variable names, section names and locations
- `.rel .text`: relocation info for code (`.text`)
- `.rel .data`: relocation info for `.data` section. Addresses of pointer data that will need to be modified in the merged executable
- `.debug`: info for symbolic debugging `gcc -g`
- Section header table: offsets and sizes of each section

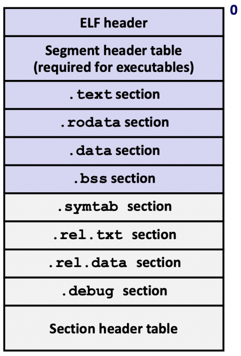


Figure 2. ELF object file format.

1.4 Linker symbols

- Global symbols: Defined by module that can be referenced by other modules, *e.g.*, non-static C functions and global variables
- External symbols: Referenced by module but defined by some other module
- Local symbols: Defined and referenced exclusively by module, *e.g.*, static C functions and global variables

Local linker symbols are *not* local program variables.

1.4.1 Local symbols

- Local non-static C variables: stored on the stack
- Local static C variables: stored in either `.bss` or `.data`

Compiler allocates space in `.data` for each definition of `x`. Creates local symbols in the symbol table with unique names: `x.1` and `x.2`.

```
int f(){
    static int x = 0;
    return x; }
int g(){
    static int x = 1;
    return x; }
```

1.4.2 Global variables

Avoid if you can. Otherwise, use `static` if you can or initialize. Use `extern` if you reference an external global variable.

1.5 Step 1: Symbol resolution

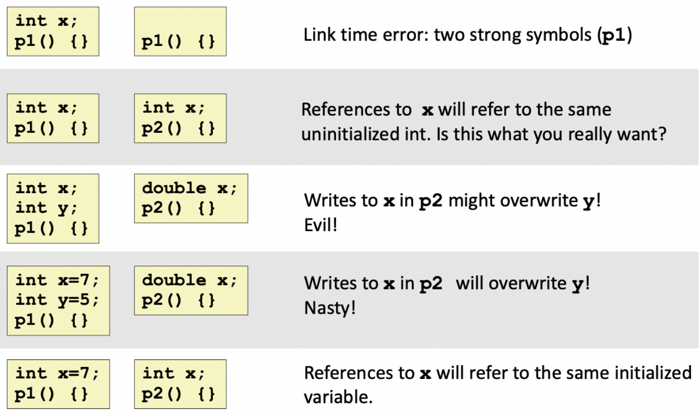
The input to the linker is relocatable object modules. What happens if multiple modules define global symbols with same name?

1.5.1 Global symbols are either *strong* or *weak*

- Strong: procedures and initialized globals
- Weak: uninitialized globals

1.5.2 Linker's symbol rules

1. **Multiple strong symbols are not allowed.** Each item can be defined only once. Otherwise: linker error
2. **Given a strong symbol and multiple weak symbols, choose the strong symbol.** References to the weak symbol resolve to the strong symbol.
3. **If there are multiple weak symbols, pick an arbitrary one.** Can override this with `gcc -fno-common`.



`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (**p1**)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to **x** will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to **x** in **p2** might overwrite **y**! Evil!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to **x** in **p2** will overwrite **y**! Nasty!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to **x** will refer to the same initialized variable.

Figure 3. Linker puzzles.

1.6 Step 2: Relocation

Once the linker has completed the symbol resolution, it knows the exact sizes of the code and data sections in its input object modules. Then linker begin relocation, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consist of two steps:

1. **Relocating sections and symbol definitions.** The linker merges all sections of the same type into a new aggregate section. The linker then assigns run-time memory addresses to the new aggregate section, to each section and to each symbol defined by input modules, so that each instruction and global variable in the program has a unique run-time memory address.
2. **Relocating symbol references within sections**. The linker modifies every symbol reference in the bodies of the code and data sections so that they point the correct run-time addresses.

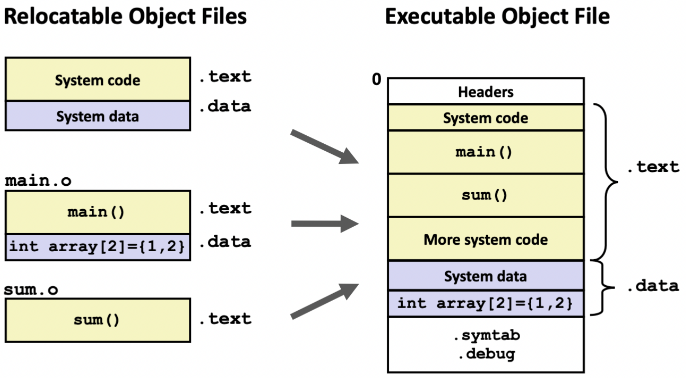


Figure 4. Relocation.

1.6.1 Relocation entries

When an assembler generates an object module, it does not know where the code, data, and externally defined functions or global variables that are referenced be stored in memory. So whenever assembler encounters a reference to an object whose location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in .rel.text and data are placed in .rel.data.

Format of an ELF relocation entry

```
typedef struct {
    long offset; // Offset of the reference to relocate
    long type:32, // Relocation type
      symbol:32; // Symbol table index
    long addend; // Constant part of relocation expression
} Elf64_Rela;
```

ELF defines 32 different relocation types, but we are concerned with only the two most basic types:

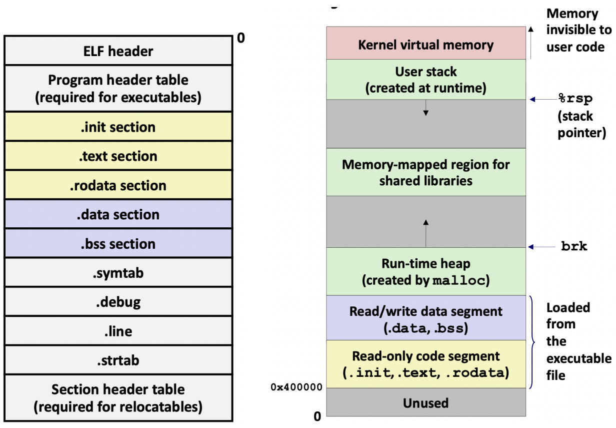
- **R_X86_64_PC32**. Relocate a reference that uses a 32-bit PC-relative address, *i.e.*, offset from current run-time value of PC.
- **R_X86_64_32**. Relocate a reference that uses a 32-bit absolute address. CPU directly uses 32-bit value encoded in the instruction as *effective address*, *e.g.*, the target of call instruction

1.6.2 Relocating symbol references

Pseudocode for the linker's relocation algorithm

```
foreach section s {
foreach relocation entry r {
    rp = s + r.offset; // ptr to ref to be relocated
    if (r.type == R_X86_64_PC32) {
    ra = ADDR(s) + r.offset; // run-time address of ref
    *rp = (unsigned) (ADDR(r.symbol) + r.addend - ra); }
    if (r.type == R_X86_64_32){
    *rp = (unsigned) (ADDR(r.symbol) + r.addend);}
}}
```

1.7 Executable object files



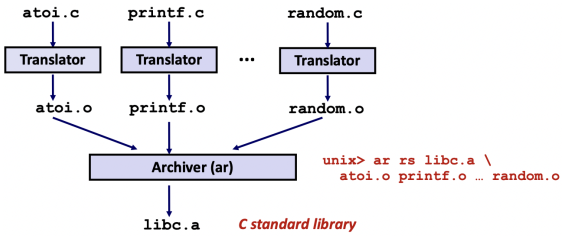
(a) ELF executable object file (b) Runtime memory image

- .text, .rodata, .data: similar to object file
- .init: defines a small function called _init that will be called by the program's initialization code
- No .rel sections since the executable is *fully linked*
- Program header table: describes mapping from contiguous chunks of the executable file to contiguous memory segments

The loader copies the code and data in the executable object file from disk into memory and then runs the program by jumping to its first instruction, or *entry point*.

1.8 Static libraries (.a archive files)

1. Concatenate related relocatable object files into a single file with an index, called an *archive*
2. Enhance linker so that it tries to resolve unresolved external references by looking one or more archives
3. If archive member file resolves reference, link it



```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

Figure 5. Creating static libraries

Archiver allows incremental updates. It recompiles function that changes and replace .o file in archive.

1.8.1 Commonly used libraries

- **libc.a** (C standard library): I/O, memory allocation, ...
- **libm.a** (C math library): sin, cos, log, exp, ...

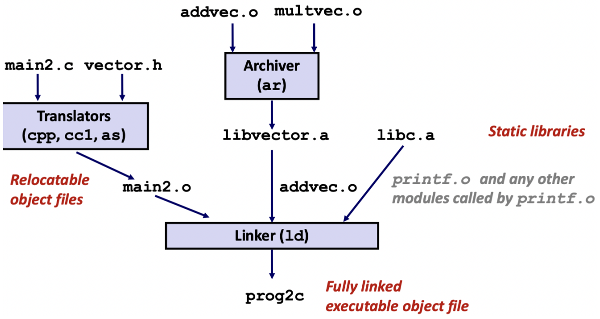


Figure 6. Linking with static libraries

1.8.2 Solving external references

1. Scan .o files and .a files in the command line order
2. For each scan, try to resolve each unresolved reference
3. If any entry in unresolved at the end of scan, error

Command line order matters.

1.9 Shared libraries (.so files)

Static libraries have the following disadvantages:

- Duplication in the stored executables (every function needs libc)
- Duplication in running executable
- Minor bug fixes of system libraries require each application to explicitly relink

Solution is using *shared libraries*: object files that contain code and data that are loaded and linked into an application *dynamically*, at either load-time and run-time. It is also called dynamic link libraries (DLLs) and performed by a *dynamic linker*. A single copy of the .text section of a shared library in memory can be shared by different running processes.

1.9.1 Load-time linking

gcc -shared -o libvector.so addvec.c multvec.c

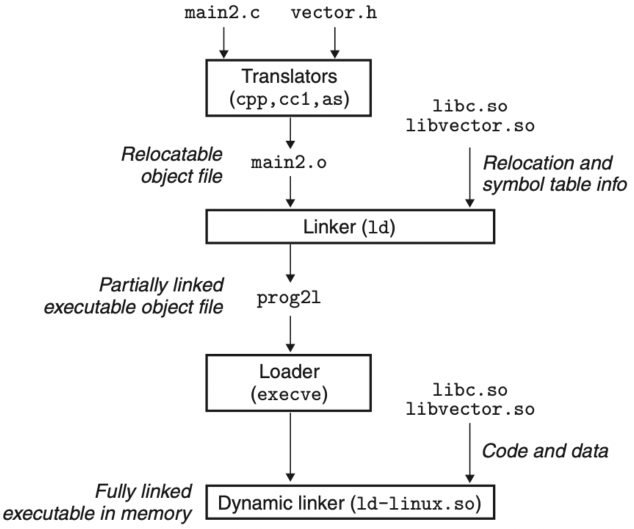


Figure 7. Dynamic linking in load time

1.9.2 Run-time linking

In Linux, this is done by calls to the dlopen().

Code 1. Dynamic linking at run-time

```c
int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main() {
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;
    /* Dynamically load the shared library that contains
     addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1); }
    /* Get a pointer to the addvec() function we just
     loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1); }
    /* Now we can call addvec() just like any other
     function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);
    /* Unload the shared library */
    if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1); }
}
```

1.10 Library interpositioning

Library interpositioning is a powerful linking technique that allows programmers to intercept calls to arbitrary functions. Given target function, create a wrapper function whose prototype is identical to the target function. Then trick the system into calling the wrapper function instead of the target function. Interpositioning can occur at compile time, link time, or load/run time.

1.10.1 Applications

- Security
- Debugging
- Monitoring and profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - Generating address traces

2 Exceptional control flow

Processors do only one thing: read and execute a sequence of instructions, one at a time. This sequence is the CPU's *control flow*. The abrupt changes to the flow is caused by instructions such as jumps, calls, and returns. Such instructions allow programs to react to changes in internal program state.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program, for example:

- User hits Ctrl-C at the keyboard
- Instruction divides by zero
- Data arrives from a disk or a network adapter

Modern systems react to these situations by making abrupt changes in the control flow, which is called *exceptional control flow* (ECF).

Mechanism	Level	Implemented by
Exceptions	Low	Hardware and OS software
Process context switch	High	Os software and hardware timer
Signals	High	OS software
Nonlocal jumps	High	C runtime library

Table 2. ECF exists at all levels of a computer system.

2.1 Exceptions

An *exception* is a transfer of control to the OS *kernel* in response to some event, *e.g.*, divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C. As shown in Fig. 8, a change in the processor's state (an event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After finishing processing exception, one of below happens:

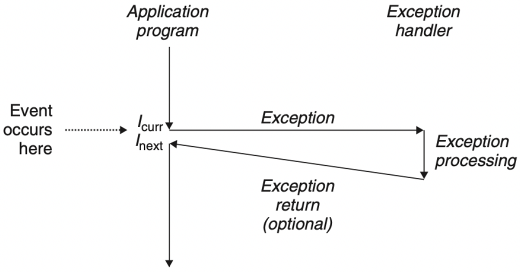


Figure 8. Anatomy of an exception.

- Handler returns control to current instruction I_{curr}.
- Handler returns control to next instruction I_{next}, *i.e.*, the instruction that would have executed after I_{curr} if event not occurred
- Handler aborts the interrupted program

2.1.1 Exception handling

Each type of possible exception in a system is assigned a unique nonnegative integer *exception number*. At the system boot time, OS allocates and initializes a jump table called *exception table*, so that entry k contains the address of the handler for exception k. The starting address of exception table is contained in special CPU register called the *exception table base register*.

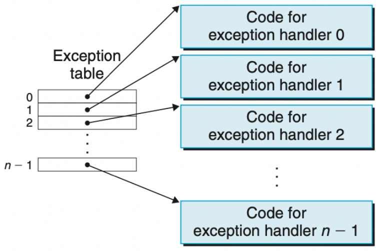


Figure 9. Exception table.

At run time, the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an *indirect procedure call* through entry k.

1. Processor pushes a return address on the stack, either current or next instruction depending on the class of exception
2. Processor pushes an additional processor state on the stack that will be necessary to restart the interrupted program when the handler returns, *e.g.*, current condition codes
3. When control is transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack
4. Exception handlers run in *kernel mode*, which means they have complete access to all system resources

2.1.2 Classes of exceptions

- **Asynchronous**: caused by events external to the processor
- **Synchronous**: caused by events that occur as a result of executing an instruction

2.2 System calls

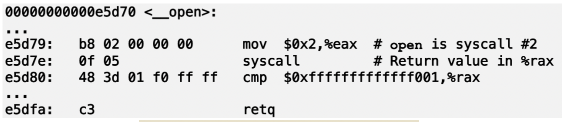
System call is user process calling a system (kernel).

Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

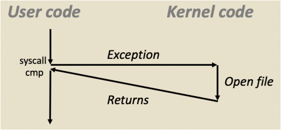
Table 3. List of system calls

2.2.1 Example: Opening file

User calls `open(filename, options)`. It calls `__open` function, which invokes system call instruction `syscall`.



```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```

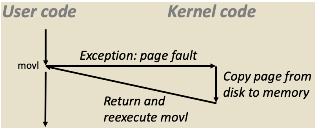


Class	Cause	Example	Sync	Return behavior
Interrupt	Signal from I/O device	Ctrl-C, arrival of packet or data	Async	Next
Trap	Intentional exception	**System calls**, breakpoints	Sync	Next
Fault	Potentionally recoverable error	Page and protection faults, floating point	Sync	Current or abort
Abort	Nonrecoverable error	Illegal instruction, parity error	Sync	Abort

Table 1. Classes of exceptions.

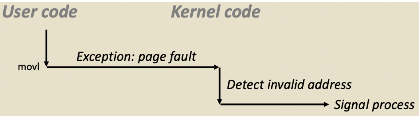
2.2.2 Example: Page fault

User writes to memory location, but that page of user's memory is currently on disk.



2.2.3 Example: Invalid memory reference

Send SIGSEGV signal to user process. User process exit with segmentation fault.



2.3 Processes

A *process* is an instance of a running program. Process provides each program with two key abstractions:

- **Logical control flow**: Seems to have exclusive use of the CPU. Provided by kernel mechanism called *context switching*
- **Private address space**: Seems to have exclusive use of main memory. Provided by kernel mechanism called *virtual memory*.

2.3.1 Multiprocessing

Traditionally, single processor executes multiple processes concurrently. Register values for processes saved in memory and loaded. Recently, we use mutlicore processors. There are multiple CPUs on a single chip, sharing main memory and some of the caches. So, each can execute a separate process. Scheduling of processors onto cores done by kernel.

2.3.2 Concurrent processes

Each process is a logical control flow. Two processes run *concurrently* if their flows overlap in time. Otherwise, they're *sequential*.

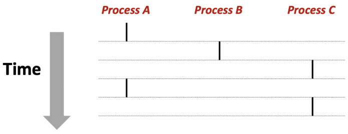


Figure 10. A & B, A & C are concurrent. B & C is sequential.

Control flows for concurrent processes are physically disjoint in time. However, user views they are running in parallel.

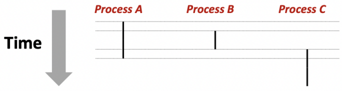


Figure 11. User view of concurrent processes.

2.3.3 Context switching

Processes are managed by a shared chunk of memory-resident OS code called *kernel*. The kernel is not a separate process, but rather runs as part of some existing process. Control flow passes from one process to another via a *context switch*.

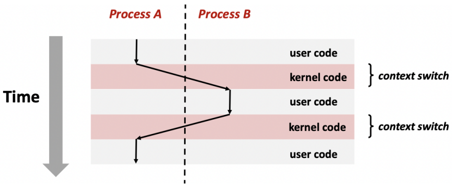


Figure 12. Context switch.

2.4 Process control

2.4.1 System call error handling

On error, Linux system-level functions typically return -1 and set global variable errno to indicate cause. So check return status of every system-level function, except a few that return void.

Code 6. Error handling

```c
/* Error-reporting function */
void unix_error(char *msg) {
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
/* Error-handling wrappers */
pid_t Fork(void) {
    pid_t pid;
    if ((pid = fork()) < 0) unix_error("Fork error");
    return pid;
}
pid = Fork();
```

2.4.2 Obtaining process IDs

- pid_t getpid(void) returns PID of current process
- pid_t getppid(void) returns PID of parent process

2.4.3 State of processes

- **Running**: either executing, or waiting to be executed and will eventually be scheduled (*i.e.*, chosen to execute) by the kernel
- **Stopped**: execution is suspended and will not be scheduled until further notice
- **Terminated**: stopped permanently

2.4.4 Terminating processes

Process becomes terminated for one of three reasons:

- Receiving a signal whose default action is to terminate
- Returning from the main routine
- Calling the exit function. void exit(int status) terminates with an exit status of status. Normal returns status is 0 and nonzero indicates error. exit is called once and never returns.

2.4.5 Creating processes

Parent process creates a new running child process by calling fork. int fork(void)

- Returns 0 to the child process
- Returns child's PID to parent process
- Child gets an identical copy of the parent's virtual address space
- Child gets identical copies of the parent's open file descriptors
- Child has a different PID with the parent

2.4.6 Process graphs

A *process graph* captures the partial ordering of statements in a concurrent program.

- Each vertex is the execution of a statement
- $a \rightarrow b$ means a happens before b
- Edges labeled with current value of variables
- Graph begins with a vertex with no incoming edges

Any topological sort of the graph corresponds to a feasible total ordering. It is when all edges point from left to right.

2.4.7 Process groups

Every process belongs to exactly one process group.

- getpgrp(): return process group of current process
- setpgid(): change process group of a process

2.4.8 Fork examples

Code 7. Fork example 1

```
int main() {
    pid_t pid;
    int x = 1;
    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x); exit(0); }
    /* Parent */
    printf("parent: x=%d\n", --x); exit(0);
}
/* Output: parent: x=0  child: x=2 */
```

- Call one, return twice
- Concurrent execution: cannot predict execution order
- Duplicate but separate address spaces
  - x has a value of 1 when fork returns in parent and child
  - Changes in x are independent
- Shared open files: stdout is the same in both parent and child

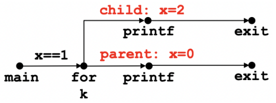


Figure 13. Process graph of fork example.

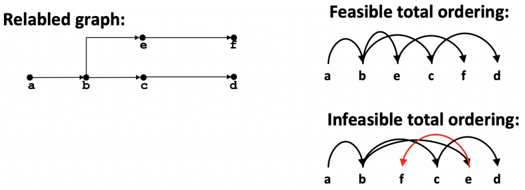
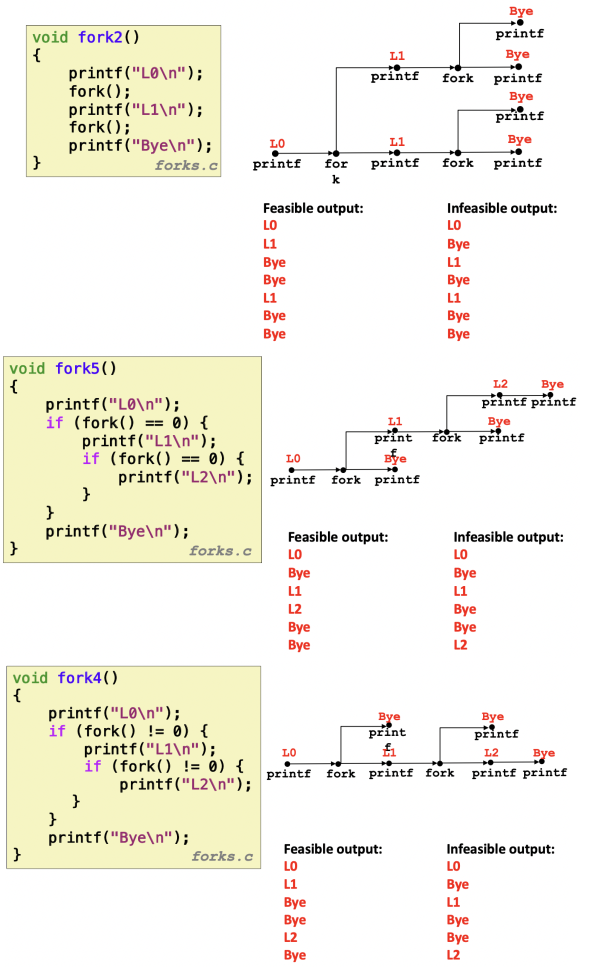


Figure 14. Feasible ordering of graph.



```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}                                forks.c
```

Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}                                forks.c
```

Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}                                forks.c
```

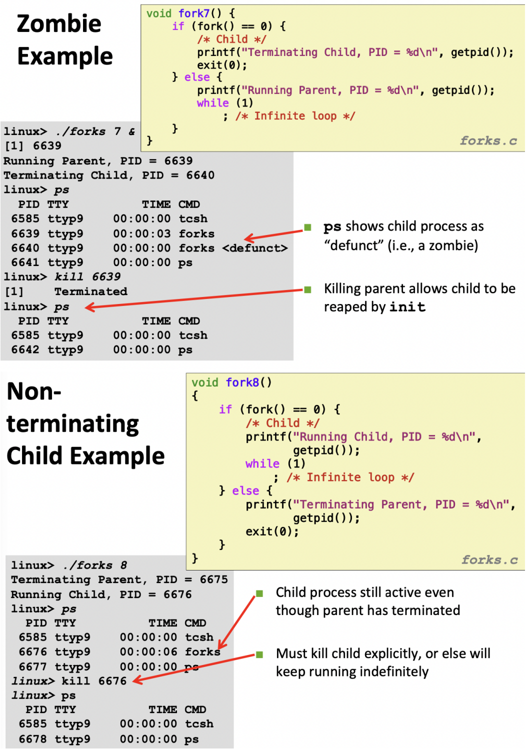
Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

Figure 15. Fork examples.

2.4.9 Reaping child processes

When process terminates, it still consumes system resources, called a *zombie*. *Reaping* is performed by parent on terminated child. Parent is given exit status information. Kernel then deletes zombie child process. If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid==1). So, we only need explicit reaping in long-running processes, *e.g.*, shells and servers.



Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}                                forks.c
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps** shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by **init**

Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}                                forks.c
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

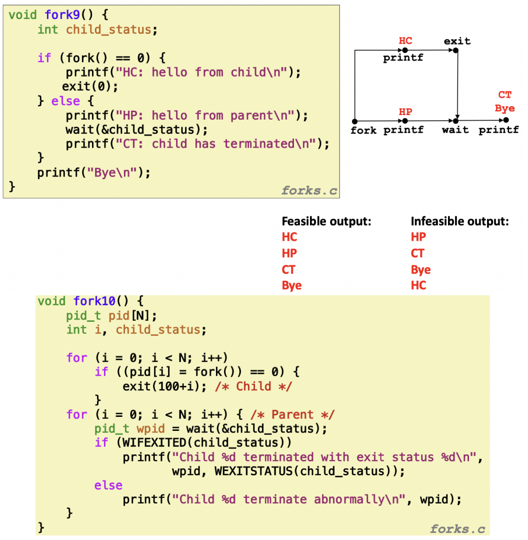
- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

Figure 16. Zombie and non-terminating child examples

2.4.10 wait: Synchronizing with children

int wait(int *child_status).

- Suspends current process until one of its children terminates
- Return value is the pid of the child process that terminated
- If child_status!=NULL, then the integer it points to will be set to a value that indicates the reason the child terminated and the exit status



```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
       exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}                                forks.c
```

Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

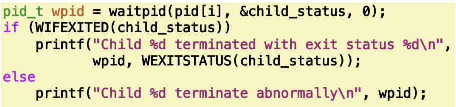
```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}                                forks.c
```

Figure 17. Wait example

If multiple children completed, it will take in arbitrary order. We can use macros WIFEXITED and WEXITSTATUS to get information about exit status.

2.4.11 waitpid: Waiting for a specific process

pid_t waitpid(pid_t pid, int &status, int options) suspends current process until specific process terminates.



```
pid_t wpid = waitpid(pid[i], &child_status, 0);
if (WIFEXITED(child_status))
    printf("Child %d terminated with exit status %d\n",
           wpid, WEXITSTATUS(child_status));
else
    printf("Child %d terminate abnormally\n", wpid);
```

Figure 18. Waitpid example

2.4.12 execve: Loading and running programs

int execve(char *filename, char *argv[], char *envp[]) loads and runs executable file filename in the current process with argument list argv and envrionment variable list envp. By convention, argv[0] == filename. envp is in format of name=value.

It overwrites code, data, and stack. IT reatins PID, open files, and signal context. It is called once and never returns except error.

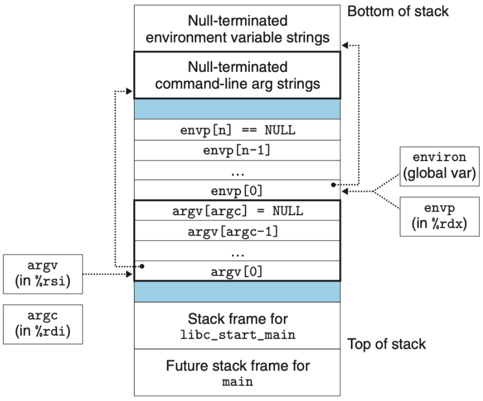
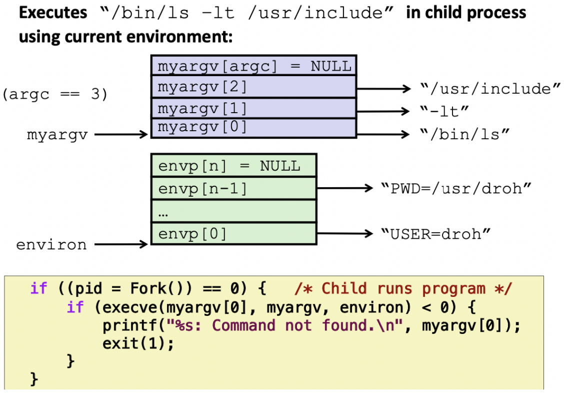


Figure 19. Organization of user stack when new program starts.



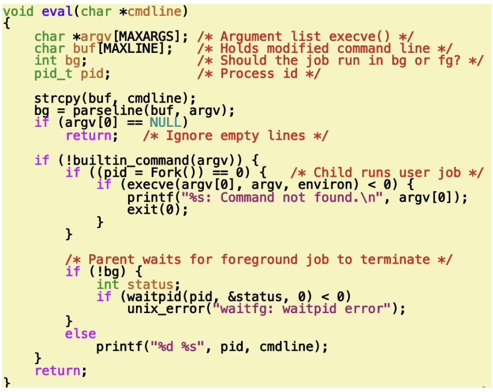
Executes "/bin/ls -lt /usr/include" **in child process using current environment:**

```
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Figure 20. execve example

2.5 Shell

Shell is application program that runs programs on behalf of user.



```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

Figure 21. Simple shell eval function

The example shell correctly waits for and reaps foreground jobs. But what about background jobs?

- Will become zombie when they terminate
- Will never be reaped because shell will not terminate
- Will create a memory leak that could run kernel out of memory

2.5.1 Process states

ps w shows the state of the processes.

No.	Letter	Meaning
First	S	sleeping
	T	stopped
	R	running
Second	s	session leader
	+	foreground group

Table 4. STAT legend

2.6 Signals

A *signal* is a message that notifies a process that an event of some type has occurred in the system. It is sent from kernel to process.

ID	Name	Default action	Event
2	SIGINT	Terminate	User typed Ctrl-C
9	SIGKILL	Terminate	Kill program
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

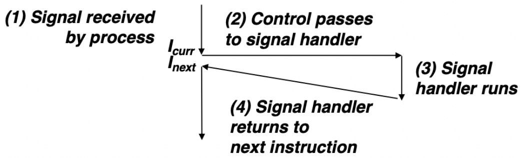
Table 5. List of signals

Kernel *sends* a signal to a *destination process* by updating some state in the context of the process. Kernel sends a signal because:

- Kenrel has detected a system event such as divide-by-zero or the termination of a child process
- Another process has invoked the kill call to request the kernel to send a signal to the destination process

A destination process *receives* a signal when it is forced by the signal to react in some way to the delivery of the signal.

- **Ignore** signal: do nothing
- **Terminate** to process
- **Catch** signal by executing a user-level function: *signal handler*



A signal is *pending* if sent but not yet received. Signals are not queued. There can be at most one pending signal of certain type. A process can *block* receipt of certain signals. Blocked signals can be delivered, but will not be received until signal is unblocked.

Kernel maintains pending and blocked bit vectors in the context of each process.

- Kernel sets/clears bit k in pending when a signal of type k is delivered/received
- blocked can be set and cleared using the sigprocmask function

2.6.1 Sending signals

/bin/kill sends arbitrary signal to a process or process group.

- /bin/kill -9 24818: send SIGKILL to process 24818
- /bin/kill -9 -24817: send SIGKILL to processes in group 24817

Typing Ctrl-C/Ctrl-Z causes the kernel to send a SIGINT/SIGTSTP to every job in the foreground process group.

2.6.2 Receiving signals

Suppose kernel is returning from an exception handler and is ready to pass control to process pnb. Kernel computes pnb = pending & ~ blocked, the set of pending nonblocked signals of process *p*.

- If pnb==0, pass control to next instruction in logical flow for *p*
- Else, repeat for all nonzero *k* in pnb:
  - Choose least nonzero bit *k* in pnb and force processes *p* to receive signal *k*
  - The receipt of the signal triggers some action by *p*
  - Pass control to next instruction in logical flow for *p*

Each signal type has a predefined default action, which is one of:

- The process terminates
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal

2.6.3 Installing signal handlers

The signal function modifies the default action associated with the receipt of signal signum:

handler_t *signal(int signum, handler_t *handler).

- SIG_IGN: ignore signals of type signum
- SIG_DFL: revert to the default action of type signum
- Otherwise, handler is the address of a user-level signal handler.

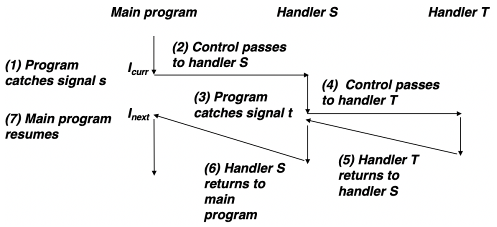
Code 8. Signal handling example

```
/* Install the SIGINT handler */
if (signal(SIGINT, sigint_handler) == SIG_ERR)
    unix_error("signal error");
pause(); // Wait for the receipt of a signal
```

Signal handler is a separate logical flow, not process, that runs concurrently with the main program.

2.6.4 Nested signal handlers

Handlers can be interrupted by other handlers.



2.6.5 Blocking and unblocking signals

To avoid nested signal handling, block signals. Kernel **implicitly** blocks any pending signals of type currently being handled. **Explicit blocking** is done by sigprocmask function. Supporting functions are:

sigemptyset	Create empty set
sigfillset	Add every signal number to set
sigaddset	Add signal number to set
sigdelset	Delete signal number from set

Code 9. Temporarily blocking signals

```
sigset_t mask, prev_mask;
Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);
/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
/* Code region that will not be interrupted by SIGINT */
/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

2.6.6 Async-Signal-Safety

Function is async-signal-safe if either *reentrant*, *i.e.*, all variables stored on stack frame, or non- interruptible by signals.

- Safe: _exit, write, wait, watipid, sleep, kill
- Not safe: printf, malloc, exit

write is the only async-signal-safe output function.

2.6.7 Example: Wait all child process

Put wait in a loop to reap all terminated children. **This is wrong**. Parent process don't have to wait for all child to terminate. It needs to do its own work. Since we install handler to SIGCHLD, just handle processes that gave parent SIGCHLD.

```
void child_handler2(int sig){
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
    }
    errno = olderrno;
}
```

2.6.8 Example: Synchronizing flows

mask_one corrects the synchronization error that assumes parent runs before child. Without it, parent would not receive SIGCHLD since it did not added child yet.

```
void handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;
    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid);
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    errno = olderrno;
}

int main(int argc, char **argv){
    int pid;
    sigset_t mask_all, mask_one, prev_one;
    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */
    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD. Child may make its children! */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid);
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}
```

2.6.9 Example: Explicitly waiting for signal

Handlers for program explicitly waiting for SIGCHLD to arrive. Code below is a still **nonsense** (bur correct) since it immediately returns child process after fork, and it waits for SIGCHLD instead of adding child process to the job list. The parent process explicitly wait for the global variable pid to change by the sigchld_handler. This is still inaccurate, but similar to a shell waiting for a foreground job.

```
volatile sig_atomic_t pid;
void sigchld_handler(int s) {
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}
void sigint_handler(int s) { }

int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);
    /* Parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
    /* Wait for SIGCHLD to be received (wasteful!) */
    while (!pid) ;
    /* Do some work after receiving SIGCHLD */
    printf(".");
    }
    exit(0);
}
```

Let's think of other options than waiting. First approach is using pause, which waits until any signal comes. However it could go around (*race*) when the process get the SIGCHLD signal just after entering the while but just before pause. Then pause will wait forever, because the process already got signal!

```
while(!pid) pause();
```

Second approach is using wait, which never going to stuck you. However, it is too *slow*. The child may end just after 1ms. But process should still wait for 1s.

```
while(!pid) sleep(1);
```

Solution is using: int sigsuspend(const sigset_t *mask).

```
while(!pid) sigsuspend(&prev);
```

It is equivalent to uninterreuptable (*atomic*) version of:

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Since we blocked mask and the operations are atomic, pid cannot change before pause.

3 Virtual Memory (VM)

3.1 Physical addressing versus virtual addressing

Physical addressing	Virtual addressing
Simple systems like camera Directly access physical	Most of the devices MMU translates virtual to physical

3.2 Why VM?

- Use main memory efficiently: Use DRAM as a cache for parts of a virtual address space
- Simplifies memory management: Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel info and code

3.2.1 Address spaces

Linear	Ordered set of contiguous integer	$\{0, 1, \cdots\}$
Virtual	Set of $N = 2^n$ virtual addresses	$\{0, 1, \cdots, N-1\}$
Physical	Set of $M = 2^m$ physical addresses	$\{0, 1, \cdots, M-1\}$

Table 6. List of address spaces

3.3 VM as a tool for caching

Conceptually, VM is an array of contiguous bytes stored on disk. Since it is typically larger than physical memory, VM provides mechanism for using the DRAM as cache.

3.3.1 Pages

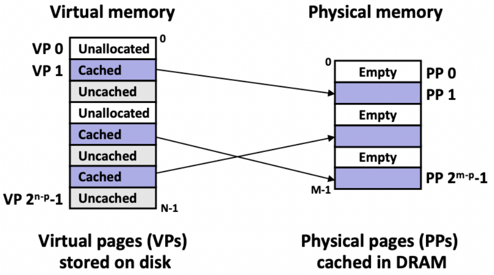


Figure 22. Virtual pages

The memory is splitted into *page*, a chunk of caches. Size of page is $P = 2^p$ bytes. In Linux system, $p = 12$, which means that we need 12 bits to access particular bytes in page. The number of virtual pages is $2^{n-p} - 1$, excluding the first VP which is unallocated.

3.3.2 DRAM cache organization

DRAM is about 10x slower than SRAM, and disk is about 10,000x slower than DRAM. DRAM cache organization is driven by the enormous miss penalty, *i.e.*, the cost of accessing something that is not close to CPU. As a result,

- Large page size, typically 4KB: increase the hit rate
- Fully associative: Any VP can be placed in any PP
- Highly sophisticated, expensive replacement algorithms
- Write-back rather than write-through

3.3.3 Enabling data structure: Page table

Replacement. If there's a miss, a system determines where the VP is stored on disk, select a victim page in physical memory, and copy the VP from disk to DRAM, replacing the victim page. This is provided by a combination of OS software, MMU, and data structure stored in physical memory known as a *page table* that maps virtual pages to physical pages.

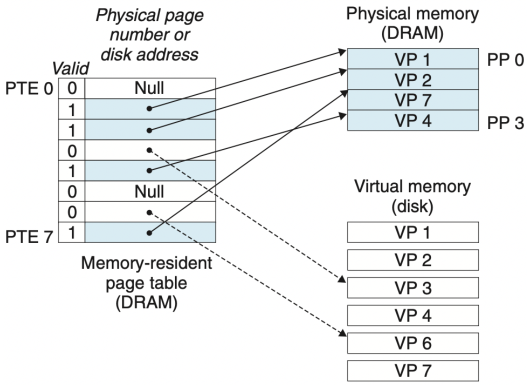


Figure 23. Page table

A *page table* is an array of *page table entries* (PTEs). Each page in the virtual address space has a PTE at a fixed offset in the page table. Valid bit indicates whether the VP is cached in DRAM:

- 1 (cached): the address indicates the start of the corresponding PP in DRAM where the VP is cached
- 0 (unallocated): null address indicates that VP not allocated
- 0 (uncached): the address points to the start of the VP on disk

3.3.4 Handling page faults

- Page hit: reference to VM word that is in physical memory
- Page fault: reference to VM word that is not in physical memory

Page fault is also an exception and handled by:

1. Page fault handler selects a victim to be evicted.
2. If victim is modified, the kernel copies it back to disk.
3. Update victim PTE: point virtual memory, change valid bit.
4. Kernel copies referenced page from disk to physical memory, and update PTE: point physical memory, change valid bit
5. Handler returns. It restarts the faulting instruction, which resends the faulting virtual address to the address translation hardware. But now it is cached in main memory, so page hit.

Waiting until the miss to copy the page to DRAM is known as *demand paging*.

3.3.5 Allocating pages

When OS allocates new page of VM, *e.g.*, as a result of malloc, VP is allocated by creating room on disk and updating PTE to point the created page on disk.

3.3.6 Locality to the rescue!

Will the large miss penalties destroy the program performance? Although the total number of pages that program references during an entire run might exceed the physical memory, the program will tend to work on smaller set of *active pages* known as the *working set* or resident set. After an initial overhead where the working set is paged to the memory, subsequent references to the working set will result in hits. However, if the working set size exceeds the size of physical memory, then the program will cause *thrashing*, where pages are swapped in and out continuously.

3.4 VM as a tool for memory management

OS provide a separate page table, and thus a separate virtual address space, for each process. Note that multiple VPs can be mapped to the same shared PP. The combination of demand paging and separate virtual address spaces has a profound impact on memory management:

- **Simplifying linking**. Each program has similar virtual address space. Code, data, and heap always start at the same addresses
- **Simplify loading**

3.5 VM as a tool for memory protection

PTE can be extended with additional permission bits, so that MMU can check these bits on each access.

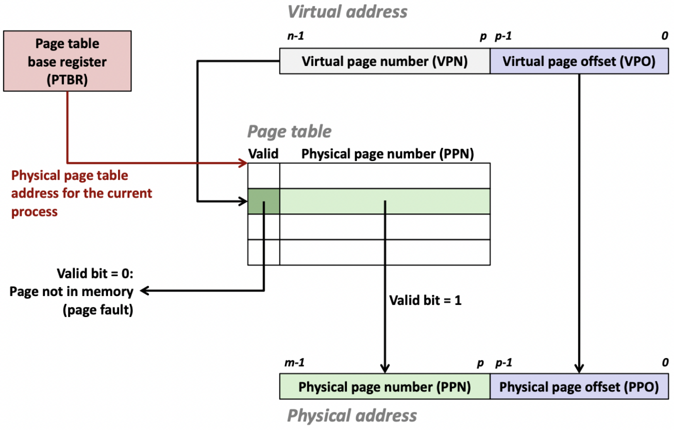
3.6 Address translation

Say virtual address space $V = \{0, \cdots, N-1\}$, physical address space $P = \{0, \cdots, M-1\}$, address translation is the mapping $MAP : V \to P \cup \{\emptyset\}$.

3.6.1 Address translation with a page table

The n-bit virtual address has two components: a p-bit VP offset (VPO) and an $(n-p)$-bit VP number (VPN). A control register in CPU, page table base register (PTBR) points to the current page table. The MMU adds VPN to PTBR to get PTE address (PTEA).

- If page hit, the corresponding physical address (PA) is the concatenation of the PP number (PPN) from PTE and VPO.
- If page fault, *i.e.*, valid bit is zero, MMU triggers page fault exception. See 3.3.4.



3.6.2 Integrating VM and cache

Consider L1 cache in CPU chip. If there's PTEA or PA hit, get it from L1 cache. If miss, get it from memory, and store in L1 cache.

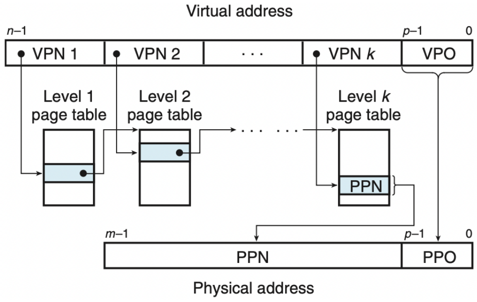
3.6.3 Speeding up translation with a TLB

Let's eliminate even the cost looking L1 cache by including a small cache of PTEs in MMU called a translation lookaside buffer (TLB). TLB maps VPN to PPN instead of storing a whole PTE. VPN is consisted of t-bits TLB index (TLBI) and TLB tag (TLBT). TLB is consisted of multiple sets containing the tag and PTE. TLBI decides the set and compare TLBT with the tags in the set.

3.6.4 Multi-level page tables

Suppose a 4KB (2^{12}) page size, 48-bit address space, and 8-byte PTE. Then we will need $2^{48-12} \times 2^3 = 2^{39}$ bytes!

→ Multi-level page table. Consequently, multiple VPNs. For example, level-1 table points to a level-2 table. Level-2 table points to a page. Since most of the level-2 table would be empty, most of the level-1 table would be null. So, most of the level-2 tables are not allocated. Usually the number of levels is 4.



4 Malloc

4.1 Dynamic memory allocation

Dynamic memory allocators manage an area of process VM known as heap. Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*.

- **Explicit allocator**: application allocates and frees space
- **Implicit allocator**: application allocates, but does not free space, *e.g.*, garbage collection in Java

4.1.1 The `malloc` package

`void *malloc(size_t size)`

- Successful: Returns a pointer to a memory block of at least `size` bytes aligned to 16-byte (x86-64). If `size==0`, returns NULL
- Unsuccessful: Returns NULL and sets `errno`

`void free(void *p)`

- Returns the block pointed at by p
- p must come from a previous call to `malloc` or `realloc`

Other functions are:

- `realloc`: changes the size of a previously allocated block
- `sbrk`: used internally by allocators to grow or shrink the heap

4.2 Performance goals

- Handling arbitrary request sequences
- Making immediate responses to requests
- Using only the heap
- Aligning blocks (alignment requirement)
- Not modifying allocated blocks

Within the constraints, an allocator attempts to meet the often conflicting performance goals:

- **Throughput**: number of request it completes per unit time
- **Memory utilization**: peak utilization $U_k = \frac{\max_{i \le k} P_i}{H_k}$ where H_k denote the current size of the heap and P_i denote the sum of payloads of currently allocated blocks

4.2.1 Fragmentation

Poor memory utilization is caused by *fragmentation*.

4.2.2 Internal fragmentation

Occurs if payload is smaller than block size. It is caused by

- Overhead of maintaining heap data structures
- Padding for alignment
- Explicit policy decisions

It depends on pattern of previous requests, so is easy to measure.

4.2.3 External fragmentation

Occurs when there is enough heap memory, but no single free bock is large enough. It depends on pattern of future requests, so is hard to measure.

4.3 Implementation

4.3.1 Knowing how much to free

Use *header/footer* to keep the length of the block.

4.3.2 Keeping track of free blocks

- Implicit list using length: links all blocks
- Explicit list: links free blocks using pointers
- Segregated free list: different free lists for different size classes
- Block sorted by size: balanced tree

4.4 Implicit list

Header and footer (boundary tags) indicates size and tag. Footer allows a bidirectional coalescing.

4.4.1 Finding a free block (placement policy)

- **First fit**: Search list from beginning, choose first that fits
  - Can take linear time in total number of blocks
  - Cause splinters at beginning of list
- **Next fit**: First fit, search starts where previous search finished
  - Faster than first fit: avoid re-scanning
  - Fragmentation is worse
- **Best fit**: fits with the fewest bytes left over
  - Keep fragments small: higher memory utilization
  - Typically slower than first fit

4.4.2 Allocating in free block

Split the remaining space when allocating free block.

4.4.3 Freeing a block

Only clearing tag leads to false fragmentation → Coalescing!

- Immediate coalescing: coalesce each time freeing
- Deferred coalescing: coalesce when the amount of external fragmentation reaches some threshold

4.5 Explicit list

Include next, prev pointers after header tag. Maintain the list of free blocks. Only free blocks, so we can use payload area.

4.5.1 Insertion policy

Where in the free list do you put a newly freed block?

- LIFO: beginning of the free list
  - Pro: simple and constant time
  - Con: fragmentation
- Address-ordered policy: blocks are always in address order
  - Pro: lower fragmentation
  - Con: require search

4.6 Segregated free list

Each size class of blocks has its own free list. Separate classes for each small size, two-power size classes for larger sizes.

- Higher throughput
- Better memory utilization: First-fit search of segregated free list approximates a best-fit search of entire heap.

4.6.1 Allocating

1. Search appropriate free list
2. If an appropriate block is found: split block and place fragmentation on appropriate list
3. If no block is found, try next larger class

If no block is found, request heap memory from OS using `sbrk()`. Extend chunk size, and put remaining free block to appropriate list.

4.6.2 Freeing

Coalesce and place on appropriate list.

Cost	Implicit	Explicit	Segregated
Allocate	Linear to all	Linear to free	Log time
Free	Constant	Constant	Constant
Memory	Depends	Better	Best

Table 7. Comparison on allocators

5 System-level I/O

5.1 Unix I/O

A Linux file is a sequence of bytes. All I/O devices, even the kernel, are represented as files. Mapping of files to devices allows kernel to export simple interface called Unix I/O:

- Opening and closing files: open(), close()
- Reading and writing a file: read(), write()
- Changing the current file position (seek): lseek()

5.1.1 File types

Each file has a type indicating its role in the system.

- **Regular file**: Contains arbitrary data
  - **Text files** are with only ASCII or Unicode character. Text file is a sequence of text lines, terminated by newline char (0xa).
  - Binary files are everything else.
  - Kernel doesn't know the difference!
- **Directory**: Index for a related group of files
  - Consists of an array of links mapping a filename to a file
  - Contains at least two entries: link to itself (.), and link to the parent directory (..)
- **Socket**: For communicating with a process on another machine

5.1.2 Directory hierarchy

All files are organized as a hierarchy anchored by root directory named / (slash). Kernel maintains *current working directory* (cwd) for each process. Locations of files in hierarchy are denoted by pathnames.

- Absolute: path from root and starts with /
- Relative: path from cwd

5.1.3 Opening and closing files

Opening/closing a file informs the kernel that you are ready/finished accessing that file. open returns a identifying integer *file descriptor*. fd==-1 indicates that an error occurred.

```c
int fd; /* file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) exit(1);
int retval; /* return value */
if ((retval = close(fd)) < 0) exit(1);
```

Each process created by a Linux shell begins life with three open files associated with a terminal:

- 0: standard input (stdin)
- 1: standard output (stdout)
- 2: standard error (stderr)

5.1.4 Reading and writing files

Reading/writing a file copies bytes CFP→memory/memory→CFP, and then updates file position. Writing a file copies bytes from memory to the current file position, and then updates file position. Returns a number of bytes read/written. nbytes<0 indicates that an error occurred.

```c
char buf[512];
int fd;
int nbytes;
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) exit(1);
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) exit(1);
```

Short counts (nbytes < sizeof(buf)) are possible and are not errors. Short count can occur when encountering EOF on reads, reading from terminal, or reading and writing network sockets. But never occur when reading and writing from disk files.

5.1.5 Example

```c
int main(void) {
    char c;
    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

5.2 Metadata

Metadata is data about data. Per-file metadata maintained by kernel. Users can access with the stat and fstat functions.

5.3 File sharing

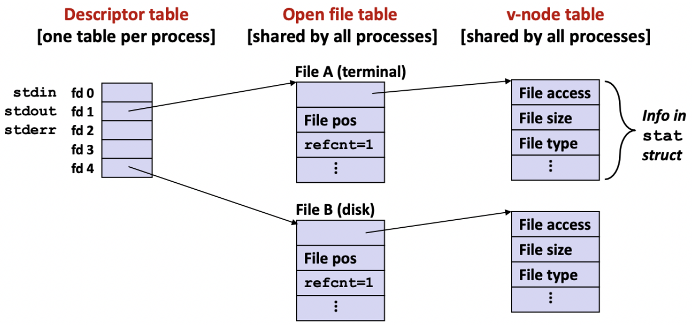


Figure 24. How the kernel represents open files

File sharing is two distinct descriptors sharing the same disk file through two **distinct** open file table entries, *e.g.*, by open twice.

Processes share files using **same** open file table entries. A child process inherits parent's open files. After fork, child process's descriptor table is same as parent's, and refcnts are incremented.

5.3.1 I/O redirection

Linux shells provide I/O redirection operators that allow users to associate standard input and output with disk files. For example, ls > foo.txt causes the shell to load and execute the ls program, with standard output redirected to disk file foo.txt.

Shell implement it by calling dup2(oldfd, newfd) function. It copies per-process descriptor table entry oldfd to entry newfd.

1. Open file to which stdout should be redirected. Happens in child executing shell code, before exec.
2. Call dup2. Change refcnt accordingly.

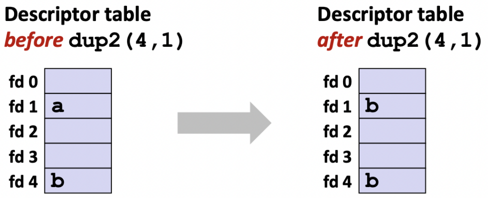


Figure 25. I/O redirection

5.4 Standard I/O

C standard library libc.so contains a collection of higher-level standard I/O functions.

- Opening and closing files: fopen, fclose
- Reading and writing bytes: fread, fwrite
- Reading and writing text lines: fgets, fputs
- Formatted reading and writing: fscanf, fprintf

5.4.1 Streams

Standard I/O models open files as *streams*, abstraction for a file descriptor and a buffer in memory. C programs begin life with three open streams: stdin, stdout, and stderr.

	Unix I/O	Standard I/O
Pros	Most general and lowest overhead Provides functions for accessing metadata Async-signal-safe and can be used safely in signal handlers	Buffering increases efficiency by reducing system calls Short counts automatically handled
Cons	Dealing with short counts is tricky No buffering → Lower efficiency	No functions for accessing metadata Not asnyc-signal-safe, not appropriate for signal handlers Not appropriate for I/O in network sockets

Table 8. Comparison on I/O

```
#include <stdio.h>
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
int main() {
    fprintf(stdout, "Hello, world\n");
}
```

5.4.2 Buffered I/O

Applications often read/write one character at a time. Implementing as Unix I/O calls is expensive. Solution is *buffered read*.

- Use Unix read to grab block of bytes
- User input functions take one byte at a time from buffer
- Refill buffer when empty

Buffer is flushed to output fd on newline char, call to fflush, exit, or return from main.

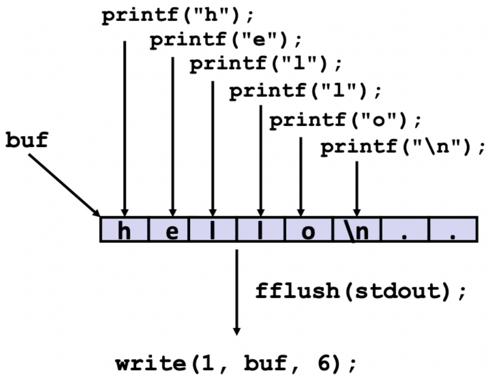
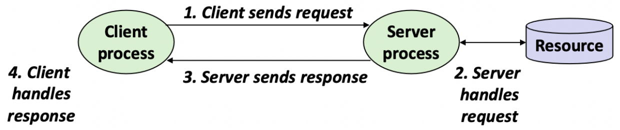


Figure 26. Buffered write

6 Network programming

6.1 A client-server transaction



6.2 Computer netwroks

A *network* is a hierarchical system of boxes and wires organized by geographical proximity.

- LAN (local area network): spans a building or campus
- WAN (wide area network): spans country or world

An *internetwork* (internet) is an interconnected set of networks. The Global IP Internet (uppercase "I") is the unique and united form of an internet (lowercase "i").

6.2.1 Internet protocol

Protocol is set of rules of how hosts and routers should cooperate when they transfer data from network to network. Internet protocol defines:

- **Naming scheme**. Defines a uniform format of *host address*.
- **Delivery mechanism**. Defines a standard transfer unit (*packet*) consisting of *header* and *payload*.
  - Header: contains info such as packet size, source and destination address
  - Payload: contains data bits sent from source host

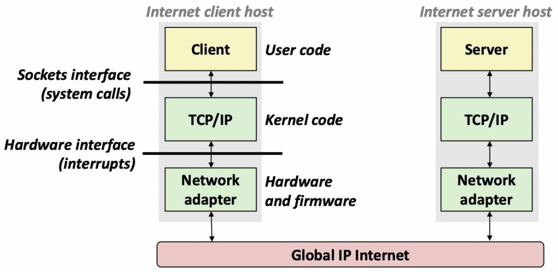
6.2.2 Global IP Internet

Based on TCP/IP protocol family.

- IP (internet protocol): Provides basic naming scheme and unreliable delivery of packets (datagrams) from host-to-host
- UDP (unreliable datagram protocol): Uses IP to provide unreliable datagram delivery from process-to-process
- TCP (transmission control protocol): Uses IP to provide reliable byte streams from process-to-process over connections

Via a mix of Unix file I/O and functions from sockets interface.

6.2.3 Hardware and software organization of an internet



6.3 Programmer's view of the Internet

6.3.1 IP address

Hosts are mapped to a set of 32-bit IP addresses

- Stored in an IP address struct
- Always stored in memory in network byte order (big-endian)
- Dotted decimal notation: each byte in address is represented by its decimal value and separated by a period

6.3.2 Domain naming system (DNS)

The set of IP addresses is mapped to a set of identifiers called Internet domain names in DNS.

- Each host has a locally defined domain name *localhost* which always maps to the *loopback address* 127.0.0.1
- Use hostname to determine real domain name of local host
- Mapping is many-to-many

6.3.3 Connection

Clients and servers communicate by sending streams of bytes over *connections*. Each connection is:

- Point-to-point: connects a pair of processes
- Full-duplex: data can flow in both directions at the same time
- Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent

A *socket* is endpoint of a connection. Socket address is an IPaddress:port pair. A *port* is a 16-bit integer that identifies a process.

- Ephemeral port: Assigned automatically by client kernel when client makes a connection request.
- Well-known port: Associated with service provided by a server

Popular services have permanently assigned well-known ports and corresponding well-known service names:

- echo server: 7/echo
- ssh servers: 22/ssh
- email server: 25/smtp
- Web servers: 80/http

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*).

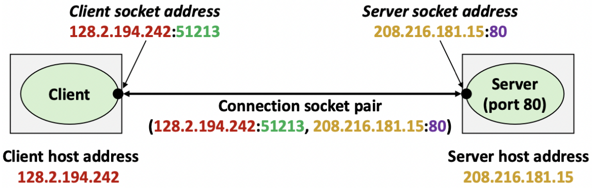


Figure 27. 51213 is an ephemeral port allocated by the kernel.

6.4 Socket

To the kernel, a socket is an endpoint of communication. To an application, a socket is a file descriptor that lets the application read/write from/to the network (networks are modeled as files!). Clients and servers communicate with each other by reading from and writing to socket descriptors. The main distinction between regular file I/O and socket I/O is how the application *opens* the socket descriptors.

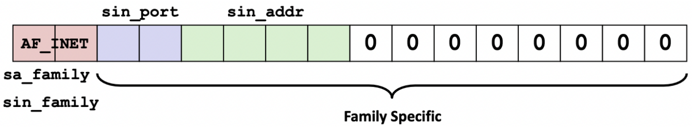
6.4.1 Socket address structures

Generic socket address:

```
struct sockaddr {
  uint16_t  sa_family; /* Protocol family */
  char sa_data[14]; /* Address data. */
};
```

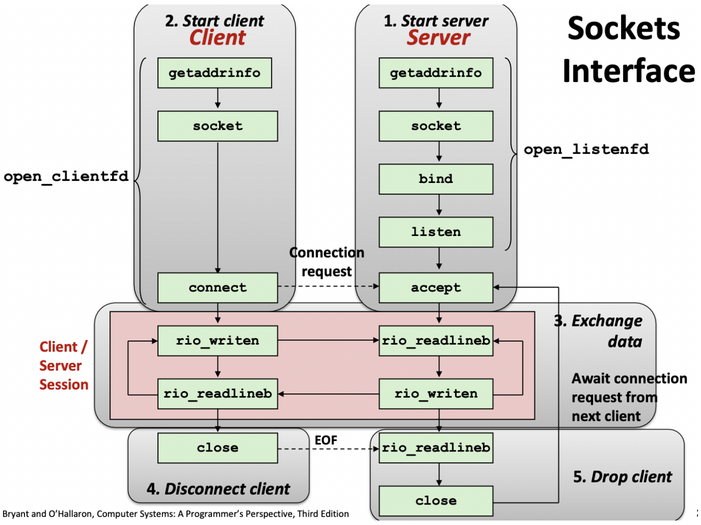
Internet-specific socket address:

```
struct sockaddr_in  {
  uint16_t sin_family; /* Protocol family (always AF_INET ) */
  uint16_t sin_port; /* Port num in network byte order */
  struct in_addr sin_addr; /* IP addr in network byte order */
  unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



Must cast to (struct sockaddr *) for functions that take socket address arguments.

6.5 Sockets interface



6.5.1 getaddrinfo

Convert strings of hostnames, host addresses, ports, and service names to socket address structures.

```
int getaddrinfo(const char *host, // Hostname or address
    const char *service, // Port or service name
    const struct addrinfo *hints, // Input params
    struct addrinfo **result); // Output linked list
```

Given host and service, getaddrinfo returns result that points to a linked list of addrinfo structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to socket function. Also points to a socket address struct that can be passed directly to connect and bind functions.

Clients walk the list, trying each socket address in turn, until the calls to socket and connect succeed. Servers walk the list until calls to socket and bind succeed.

6.5.2 getnameinfo

getnameinfo is the inverse of getaddrinfo, converting a socket address to the corresponding host and service.

6.5.3 socket

int socket(int domain, int type, int protocol)

Clients and servers use the socket function to create a socket descriptor. It is protocol specific. Use getaddrinfo to generate the parameters automatically, so that code is protocol independent.

6.5.4 bind

int bind(int sockfd, SA *addr, socklen_t addrlen);

A server uses bind to ask the kernel to associate the server's socket address with a socket descriptor. Use getaddrinfo to supply parameters.

6.5.5 listen

int listen(int sockfd, int backlog);

By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection. A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client. It converts sockfd from an active socket to a listening socket that can accept connection requests from clients.

6.5.6 accept

int accept(int listenfd, SA *addr, int *addrlen);

Servers wait for connection requests from clients by calling accept. Returns a *connected descriptor* that can be used to communicate with the client via Unix I/O routines.

6.5.7 connect

int connect(int clientfd, SA *addr, socklen_t addrlen);

A client establishes a connection with a server by calling connect. If successful, then clientfd is now ready for reading and writing.

6.5.8 Listening vs. connected descriptors

- Listening: Created once and exists for lifetime of the server. End point for client connection requests.
- Connected: End point of the connection between client and server
- New descriptor is created each time the server accepts a connection request from a client

6.5.9 Example

Helpers

```c
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;
    // Get a list of potential server addresses
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM; // Open a connection
    hints.ai_flags = AI_NUMERICSERV; // using numeric port
    hints.ai_flags |= AI_ADDRCONFIG;
    Getaddrinfo(hostname, port, &hints, &listp);
    // Walk the list for one that we can successfully connect to
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) continue; // failed, try next
        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; // Success
        Close(clientfd); // Connect failed, try another
    }
    Freeaddrinfo(listp); // Clean up
    if (!p) return -1;
    else return clientfd;
}
int open_listenfd(char *port) {
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;
    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM; // Accept connect.
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; // any addr
    hints.ai_flags |= AI_NUMERICSERV; // using port no.
    Getaddrinfo(NULL, port, &hints, &listp);
    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0) continue; // failed, try next
        // Eliminates "Address already in use" error from bind
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval, sizeof(int));
        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }
    Freeaddrinfo(listp); // Clean up
    if (!p) return -1;
    // Make it a listening socket ready to accept conn. requests
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd); return -1; }
    return listenfd;
}
```

echoclient.c

```c
int main(int argc, char **argv) {
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;
    host = argv[1];
    port = argv[2];
    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
       Rio_writen(clientfd, buf, strlen(buf));
       Rio_readlineb(&rio, buf, MAXLINE);
       Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

echoserveri.c

```c
void echo(int connfd);
int main(int argc, char **argv) {
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; // Enough room for any
    char client_hostname[MAXLINE], client_port[MAXLINE];
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
        Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

echo.c

```c
 void echo(int connfd) {
    size_t n;
    char buf[MAXLINE];
    rio_t rio;
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```

6.5.10 Testing servers using telnet

linux> telnet <host> <portnumber>

The telnet program is invaluable for testing servers that transmit ASCII strings over Internet connections

6.6 Web servers

Clients and servers communicate using the HyperText Transfer Protocol (HTTP).

1. Client and server establish TCP connection
2. Client requests content
3. Server responds with requested content
4. Client and server close connection (eventually)

6.6.1 Web content

Web servers return content to clients, a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type, *e.g.*, HTML, and PNG. The content returned in HTTP responses can be either static or dynamic.

- Static: stored in files and retrieved in response to HTTP request
- Dynamic: produced on-the-fly in response to HTTP request

6.6.2 URLs

Unique name for a file: URL (Universal Resource Locator).

http://www.cmu.edu:80/index.html

- Clients use prefix (http://www.cmu.edu:80) to infer protocol, where the server is, and port
- Servers use suffix (/index.html) to determine if request is for static or dynamic content

6.6.3 HTTP requests

HTTP request is a request line followed by zero or more request headers.

Request line: <method> <uri> <version>

- <method> is one of GET, POST, OPTIONS, HEAD, PUT, DELETE, or TRACE
- <uri> is typically URL for proxies, URL suffix for servers
- <version> is HTTP version of request

Request headers: <header name>: <header data>

6.6.4 HTTP responses

HTTP response is a response line followed by zero or more response headers, possibly followed by content, with blank line separating headers from content.

Response line: <version> <status code> <status msg>

- <version> is HTTP version of the response
- <status code> is numeric status
- <status msg> is text: OK, Moved, or Not found

Response headers: <header name>: <header data>

6.7 Tiny web server

6.7.1 Tiny operations

- Accept connection from client
- Read request from client (via connected socket)
- Split into <method> <uri> <version>
- If <method> is not GET, then return error
- If URI contains cgi-bin then serve dynamic content
- Otherwise serve static content

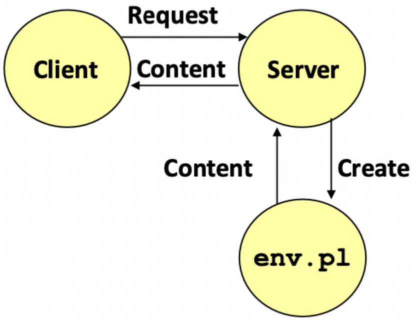


Figure 28. Serving dynamic content

6.7.2 Serving static content

```
void serve_static(int fd, char *filename, int filesize)
    {
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];
    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf,
     filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf,
     filetype);
    Rio_writen(fd, buf, strlen(buf));
    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE,
     srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

6.7.3 Serving dynamic content

1. The server creates a child process and runs the program identified by the URI in that process
2. The child runs and generates the dynamic content
3. The server captures the content of the child and forwards it without modification to the client

Common Gateway Interface (CGI) defines a simple standard for transferring information between the client (browser), the server, and the child process. Because the children are written according to the CGI spec, they are often called CGI programs.

http://add.com/cgi-bin/adder?15213&18213

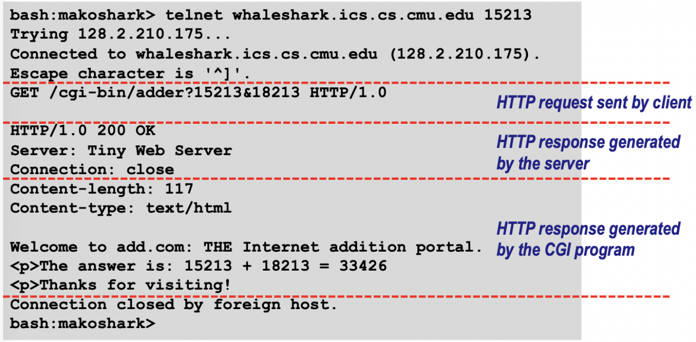
- adder is the CGI program on the server that will do the addition
- argument list starts with ?
- arguments separated by &
- spaces represented by + or %20

Server pass the arguments to child in environment variable QUERY_STR
For example, QUERY_STRING = “15213&18213”

Child generates output on stdout. Server uses dup2 to redirect it to its connected socket.

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
    {
    char buf[MAXLINE], *emptylist[] = { NULL };
    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
```

```
    Rio_writen(fd, buf, strlen(buf));
    if (Fork() == 0) { /* Child */
        /* Real server would set all CGI vars here */
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO); /* Redirect stdout to client */
        Execve(filename, emptylist, environ); /* Run CGI program */
     }
     Wait(NULL); /* Parent waits for and reaps child */
}
```



```
bash:makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
GET /cgi-bin/adder?15213&18213 HTTP/1.0

HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 117
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 15213 + 18213 = 33426
<p>Thanks for visiting!
Connection closed by foreign host.
bash:makoshark>
```

Figure 29. Dynamic content result

7 Concurrent programming

7.1 Iterative servers

Iterative servers process one request at a time.

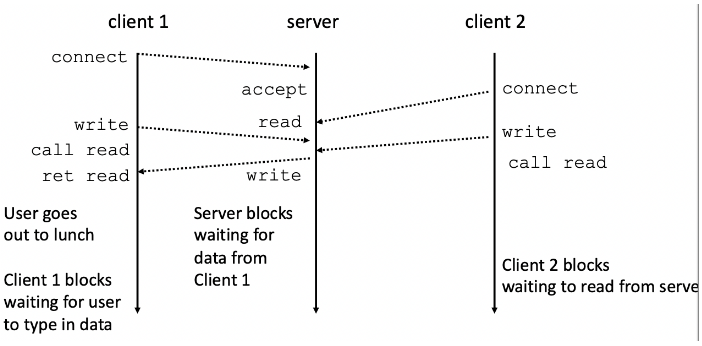
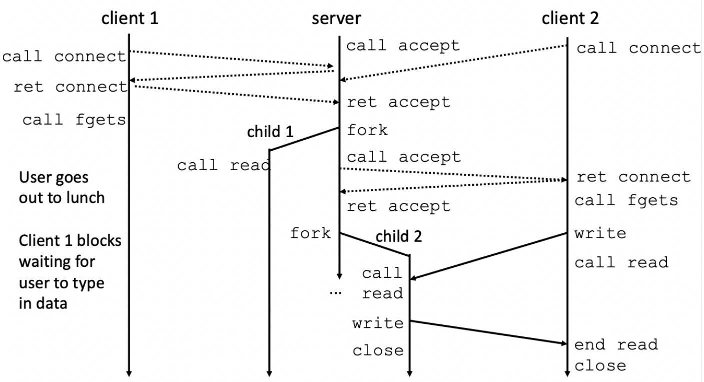


Figure 30. Fundamental flaw of iterative servers. Server waits until client 1 is closed.

Solution: use concurrent servers instead! Allow server to handle multiple clients simultaneously.

7.2 Process-based concurrent server

Spawn separate process for each client.



- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

```
int main(int argc, char **argv) {
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen=sizeof(clientaddr);
    signal(SIGCHLD, sigchld_handler);
    listenfd = open_listenfd(port);
    while (1) {
       connfd = accept(listenfd, (SA *) &clientaddr, &clientlen);
       if (fork() == 0) {
           close(listenfd); // Child closes its listening socket
        }
        echo(connfd); // Child services client
        close(connfd); // Child closes connection with client
        exit(0); // Child exits
        Close(connfd); // Parent closes connected socket (important!)
    }
}
/* Listening server process must reap zombie children to avoid fatal memory leak */
void sigchld_handler(int sig) {
    while (waitpid(-1, 0, WNOHANG) > 0) ;
    return;
}
```

7.2.1 Pros and cons

- + Handle multiple connections concurrently
- + Clean sharing model
- + Simple and straightforward
- – Additional overhead for process control
- – Nontrivial to share data between processes

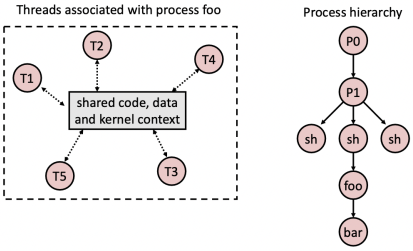
7.3 Thread-based concurrent server

Multiple threads can be associated with a process.

- Each thread has its own logical control flow
- Each thread shares the same code, data, and kernel context
- Share common virtual address space
- Each thread has its own thread id (TID)

Two threads are (logically) concurrent if their flows overlap in time. Otherwise, they are sequential. True concurrency is only possible in multi-core processor.

7.3.1 Threads vs. Processes



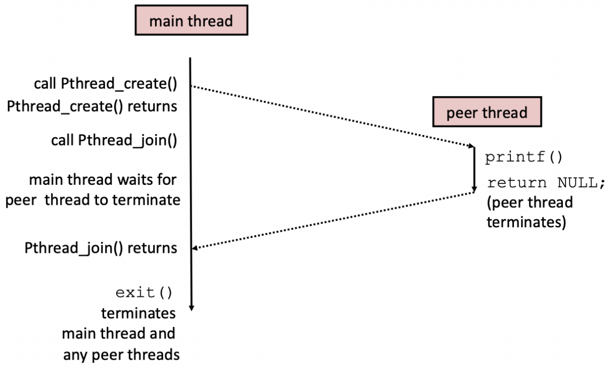
Threads share code and some data. Processes (typically) do not. Threads are less expensive than processes.

7.3.2 Posix threads (Pthreads) interface

Standard interface for functions that manipulate threads from C.

- Creating and reaping threads: pthread_create(), pthread_join()
- Determining your thread ID: pthread_self()
- Terminating threads: pthread_cancel(), pthread_exit(), exit() (terminates all threads), RET (terminates current thread)

```c
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
int main() {
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
     Pthread_join(tid, NULL);
    exit(0);
}
```



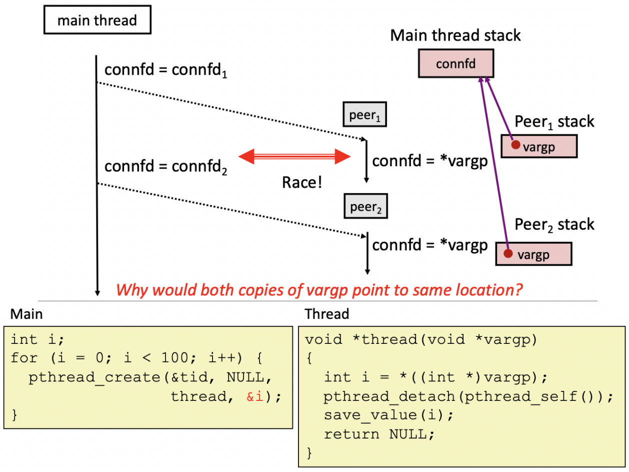
Spawn new thread for each client.

```c
int main(int argc, char **argv) {
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr); pthread_t tid;
    int listenfd = open_listenfd(port);
    while (1) {
        int *connfdp = malloc(sizeof(int));
        *connfdp = accept(listenfd, (SA *) &clientaddr, &clientlen);
        pthread_create(&tid, NULL, echo_thread, connfdp);
    }
}
void *echo_thread(void *vargp) {
    int connfd = *((int *)vargp); pthread_detach(pthread_self());
    free(vargp);
    echo(connfd);
    close(connfd);
    return NULL;
}
```

7.3.3 Pros and cons

- + Easy to share data structures between threads
- + Threads are more efficient than processes
- – Unintentional sharing

7.3.4 Unintended sharing



Why would both copies of vargp point to same location?

Main

```c
int i;
for (i = 0; i < 100; i++) {
  pthread_create(&tid, NULL,
                 thread, &i);
}
```

Thread

```c
void *thread(void *vargp)
{
  int i = *((int *)vargp);
  pthread_detach(pthread_self());
  save_value(i);
  return NULL;
}
```

8 Synchronization

8.1 Sharing

Say variable x is shared if and only if multiple threads reference instance of x. Which variables in threaded C program are shared?

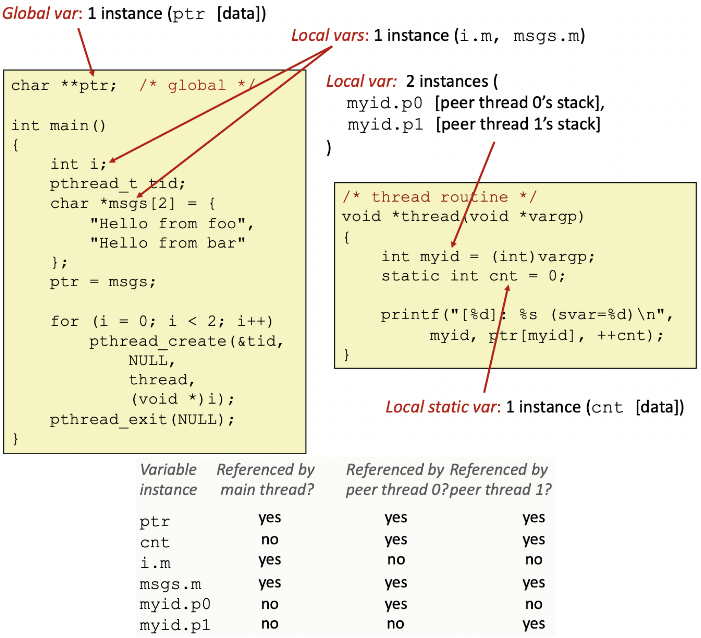
8.1.1 Threads memory model: Conceptual model

- Multiple threads run within the same context of a single process
- Each thread has its own separate thread context

This model is not strictly enforced. Any thread can read and write the stack of any other thread.

8.1.2 Mapping variable instances to memory

- Global/Local static variable: VM contains exactly one instance
- Local variables: Each thread stack contains one instance



```c
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

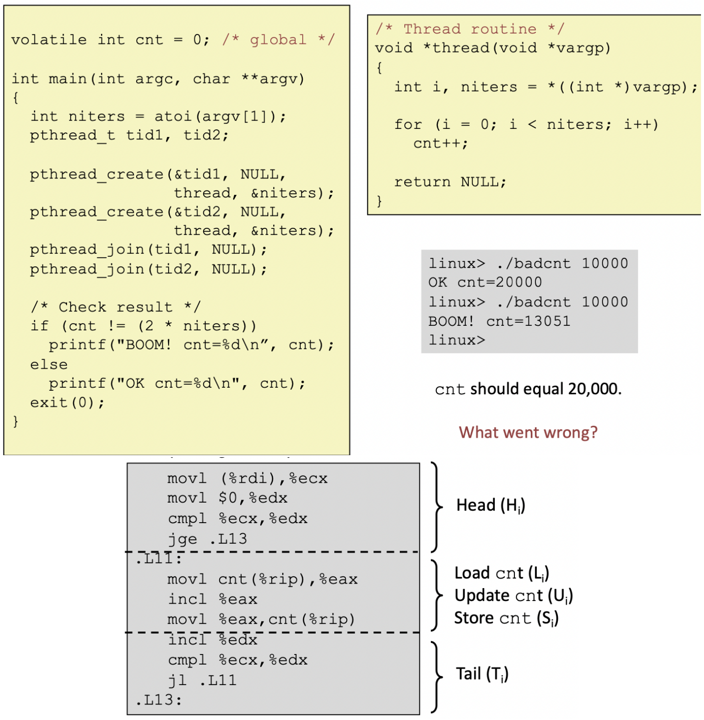
```c
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++cnt);
}
```

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

ptr, cnt, msgs are shared but i, myid are not shared.

8.2 Mutual exclusion

8.2.1 Improper synchronization example



```c
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
  int niters = atoi(argv[1]);
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL,
                 thread, &niters);
  pthread_create(&tid2, NULL,
                 thread, &niters);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  /* Check result */
  if (cnt != (2 * niters))
    printf("BOOM! cnt=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}
```

```c
/* Thread routine */
void *thread(void *vargp)
{
  int i, niters = *((int *)vargp);

  for (i = 0; i < niters; i++)
    cnt++;

  return NULL;
}
```

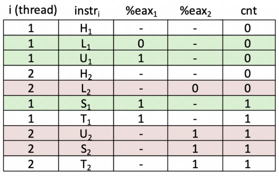
```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?

```
    movl  (%rdi),%ecx
    movl  $0,%edx
    cmpl  %ecx,%edx
    jge  .L13
.L11:
    movl  cnt(%rip),%eax
    incl  %eax
    movl  %eax,cnt(%rip)
    incl  %edx
    cmpl  %ecx,%edx
    jl  .L11
.L13:
```

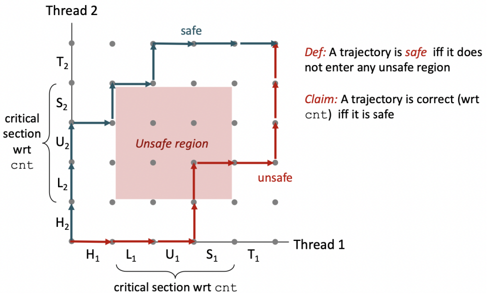
Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.



i (thread)	instr$_i$	%eax$_1$	%eax$_2$	cnt
1	H$_1$	-	-	0
1	L$_1$	0	-	0
1	U$_1$	1	-	0
2	H$_2$	-	-	0
2	L$_2$	-	0	0
1	S$_1$	1	-	1
1	T$_1$	1	-	1
2	U$_2$	-	1	1
2	S$_2$	-	1	1
2	T$_2$	-	1	1

8.2.2 Process graphs

A *progress graph* depicts the discrete execution state space of concurrent threads. Each axis corresponds to the sequential order of instructions in a thread. Each point corresponds to a possible *execution state*. A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.



Def: A trajectory is *safe* iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe

Figure 31. L, U, and S form a critical section with respect to the shared variable cnt.

To enforce safe trajectory, we must need to guarantee mutually exclusive access to critical regions.

8.3 Semaphores

Semaphore is non-negative global integer synchronization variable. It is manipulated by:

- P(s): [while (s == 0) wait(); s–;] (test)
- V(s): [s++;] (increment)

OS kernel guarantees that operations between [] are executed indivisibly: Only one P or V operation at a time can modify s.

8.3.1 Proper synchronization

- Binary semaphore: semaphore whose value is always 0 or 1
- Mutex: binary semaphore used for mutual exclusion
  - P operation: locking the mutex
  - V operation: unlocking the mutex

```
int sem_init(sem_t *sem, 0, unsigned int val);}
int sem_wait(sem_t *s); // P(s)
int sem_post(sem_t *s); // V(s)
void P(sem_t *s); // Wrapper fuc for sem_wait  void V(sem_t *s); /. Wrapper func for sem_post

volatile int cnt = 0; // Counter
sem_t mutex; // Semaphore that protects cnt
sem_init(&mutex, 0, 1);  // mutex = 1

/* Surround critical section with P and V */
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

Semaphore creates a forbidden region that encloses unsafe region that cannot be entered by any trajectory.

9 Attack Lab

9.1 Buffer overflow

When exceeding the memory size allocated for an array. It can cause security vulnerabilities.

Vulnerable buffer example

```
void echo(){
    char buf[4]; // Way too small!
    gets(buf); // Runs until EOF so no way to specify limit on number of characters
    puts(buf); }
 void call_echo() {
    echo(); }
```

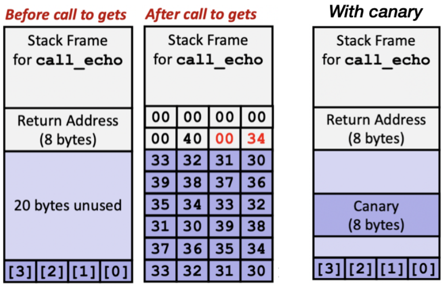


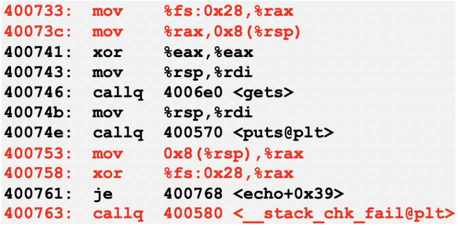
Figure 32. Buffer overflowed and corrupted the return pointer.

Buffer overflow can corrupt the return pointer and may corrupt state (cause segmentation fault) or cause undesired action.

9.1.1 Code injection attack

Return address can be corrupted to point exploit code. To prevent,

- **Avoid vulnerabilities in code.** Use methods that limits string lengths, *e.g.*, fgets, strncpy instead of gets, strcpy.
- **System-level protections.** At the start of the program, allocate random amount of space on stack, making difficult for hacker to predict the beginning of the inserted code.
- **Non-executable code segments.** Mark stack as non-executable.
- **Stack canaries**. Place special value beyond buffer and check for corruption before exiting function. Add -fstack-protector.



```
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
```

Figure 33. Disassembly of canary.

9.2 Code injection attack

Set return address to the address of touch1.

1. disas getbuf to see how much rsp is decreased
2. Get the start address of function by disas touch1
3. Put bytes to fill the size, and then put the start address of function in little endian (gh ef cd ab 00 00 00 00)

Set the assembly operations in bytes in the return address.

```
pushq $touch2 address$
movq $COOKIE$, %rdi
retq
```

10 Shell Lab

```c
/* Misc manifest constants */
#define MAXLINE 1024 /* max line size */
#define MAXARGS 128  /* max args on a command line */
#define MAXJOBS 16   /* max jobs at any point in time */
#define MAXJID 1 << 16 /* max job ID */

/* Job states */
#define UNDEF 0 /* undefined */
#define FG 1    /* running in foreground */
#define BG 2    /* running in background */
#define ST 3    /* stopped */

/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *     FG -> ST  : ctrl-z
 *     ST -> FG  : fg command
 *     ST -> BG  : bg command
 *     BG -> FG  : fg command
 * At most 1 job can be in the FG state.
 */

/* Global variables */
extern char **environ;   /* defined in libc */
char prompt[] = "tsh> "; /* command line prompt */
int nextjid = 1;         /* next job ID to allocate */
char sbuf[MAXLINE];      /* for composing sprintf messages */

struct job_t
{                          /* The job struct */
    pid_t pid;             /* job PID */
    int jid;               /* job ID [1, 2, ...] */
    int state;             /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE]; /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
/* End global variables */

/* Function prototypes */
void eval(char *cmdline);
int builtin_cmd(char **argv);
void do_bgfgkl(char **argv);
void do_export(char **argv);
void waitfg(pid_t pid);

void sigchld_handler(int sig);
void sigtstp_handler(int sig);
void sigint_handler(int sig);

/* Safe API print functions for signal handler */
ssize_t sio_puts(char s[]);
ssize_t sio_putl(long v);
void sio_error(char s[]);

/* Here are helper routines */
int parseline(const char *cmdline, char **argv);
void sigquit_handler(int sig);

void clearjob(struct job_t *job);
void initjobs(struct job_t *jobs);
int maxjid(struct job_t *jobs);
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t pid);
pid_t fgpid(struct job_t *jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
struct job_t *getjobjid(struct job_t *jobs, int jid);
int pid2jid(pid_t pid);
void listjobs(struct job_t *jobs);

void unix_error(char *msg);
void app_error(char *msg);
typedef void handler_t(int);
handler_t *Signal(int signum, handler_t *handler);
```

```c
/* The shell’s main routine */
int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];
    int emit_prompt = 1; /* emit prompt (default) */

    /* Install the signal handlers */
    Signal(SIGINT, sigint_handler);   /* ctrl-c */
    Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
    Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */
    Signal(SIGQUIT, sigquit_handler);

    /* Initialize the job list */
    initjobs(jobs);

    /* Execute the shell’s read/eval loop */
    while (1)
    {
        /* Read command line */
        if (emit_prompt) {
            printf("%s", prompt);
            fflush(stdout);
        }
        if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
            app_error("fgets error");
        if (feof(stdin)) { /* End of file (ctrl-d) */
            fflush(stdout);
            exit(0);
        }
        /* Evaluate the command line */
        eval(cmdline);
        fflush(stdout);
        fflush(stdout);
    }
    exit(0); /* control never reaches here */
}

/* eval - Evaluate the command line that the user has just typed in. If the user has requested a built-in command (quit, jobs, bg or fg) then execute it immediately. Otherwise, fork a child process and run the job in the context of the child. If the job is running in the foreground, wait for it to terminate and then return. Note: each child process must have a unique process group ID so that our background children don’t receive SIGINT (SIGTSTP) from the kernel when we type ctrl-c (ctrl-z) at the keyboard. */
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    pid_t pid;

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if(!builtin_cmd(argv)){
        sigset_t mask, prev;
        sigemptyset(&mask);
        sigaddset(&mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &mask, &prev); // block SIGCHLD
        if((pid = fork()) == 0){ // child process
            setpgid(0, 0);
            sigprocmask(SIG_SETMASK, &mask, NULL); // unblock SIGCHLD
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(1);
            }
        }
        // parent process
        addjob(jobs, pid, bg ? BG : FG, cmdline);
        sigprocmask(SIG_SETMASK, &prev, NULL); // unblock SIGCHLD
        if(!bg) waitfg(pid);
        else printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    return;
}
```

```
/* parseline - Parse the command line and build the argv array.
     Characters enclosed in single quotes are treated as a
     single argument. Return true if the user has requested a BG
      job, false if the user has requested a FG job. */
int parseline(const char *cmdline, char **argv)
{
    static char array[MAXLINE]; /* holds local copy of command
     line */
    char *buf = array; /* ptr that traverses command line */
    char *delim;  /* points to first space delimiter */
    int argc; /* number of args */
    int bg; /* background job? */

    strcpy(buf, cmdline);
    buf[strlen(buf) - 1] = ’ ’; /* replace trailing ’\n’ with
     space */
    while (*buf && (*buf == ’ ’)) /* ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    if (*buf == ’\’’) {
        buf++;
        delim = strchr(buf, ’\’’);
    } else delim = strchr(buf, ’ ’);

    while (delim) {
        if (*buf == ’$’) {
            buf++;
            char * copy = malloc(strlen(buf)-1);
            strncpy(copy, buf, strlen(buf)-1);
            if(getenv(copy)!=NULL) argv[argc++] = getenv(copy);
            *delim = ’\0’;
            buf = delim + 1;
            while (*buf && (*buf == ’ ’)) /* ignore spaces */
                buf++;
            if (*buf == ’\’’){
                buf++;
                delim = strchr(buf, ’\’’);
            } else delim = strchr(buf, ’ ’);
            free(copy);
        } else {
            argv[argc++] = buf;
            *delim = ’\0’;
            buf = delim + 1;
            while (*buf && (*buf == ’ ’)) /* ignore spaces */
                buf++;
            if (*buf == ’\’’) {
                buf++;
                delim = strchr(buf, ’\’’);
            } else delim = strchr(buf, ’ ’);
        }
    }

    argv[argc] = NULL;
    if (argc == 0) /* ignore blank line */
        return 1;

    /* should the job run in the background? */
    if ((bg = (*argv[argc - 1] == ’&’)) != 0) {
        argv[--argc] = NULL;
    }
    return bg;
}

/* builtin_cmd - If the user has typed a built-in command then
     execute it immediately. */
int builtin_cmd(char **argv)
{
    if(!strcmp(argv[0], "quit")) exit(0);
    if(!strcmp(argv[0], "jobs")){ listjobs(jobs); return 1; }
    if(!strcmp(argv[0], "bg")) { do_bgfgkl(argv); return 1; }
    if(!strcmp(argv[0], "fg")) { do_bgfgkl(argv); return 1; }
    if(!strcmp(argv[0], "kill")) { do_bgfgkl(argv); return 1; }
    if(!strcmp(argv[0], "export")) { do_export(argv); return 1;
     }
    return 0; /* not a builtin command */
}



/*
 * do_bgfgkl - Execute the builtin bg, fg and kill commands
 */
void do_bgfgkl(char **argv)
{
    struct job_t * job;
    char * stop;
    if (argv[1] == 0) {
        printf("%s command requires PID or %%jobid argument\n",
     argv[0]);
        return;
    }
    if(argv[1][0] == ’%’){
        int jid = strtol(argv[1]+1, &stop, 10);
        if (stop == argv[1]+1) {
            printf("%s: argument must be a PID or %%jobid\n",
     argv[0]);
            return;
        }
        job = getjobjid(jobs, jid);
        if (job == NULL) {
            printf("%s: No such job\n", argv[1]);
            return;
        }
    } else {
        int pid = strtol(argv[1], &stop, 10);
        if (stop == argv[1]) {
            printf("%s: argument must be a PID or %%jobid\n",
     argv[0]);
            return;
        }
        job = getjobpid(jobs, pid);
        if (job == NULL) {
            printf("(%s): No such process\n", argv[1]);
            return;
        }
    }

    kill(-job->pid, SIGCONT);

    if(!strcmp(argv[0], "bg")){
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline)
     ;
    }
    if(!strcmp(argv[0], "fg")){
        job->state = FG;
        waitfg(job->pid);
    }
    if(!strcmp(argv[0], "kill")) kill(-job->pid, SIGKILL);

    return;
}

/* do_export - Execute the builtin export commands */
void do_export(char **argv)
{
    char * name = strtok(argv[1], "=");
    char * value = strtok(NULL, "=");
    setenv(name, value, 1);
    return;
}

/* waitfg - Block until process pid is no longer the foreground
     process */
void waitfg(pid_t pid)
{
    while (pid == fgpid(jobs)) sleep(1);
    return;
}

/* sigint_handler - The kernel sends a SIGINT to the shell
     whenver the user types ctrl-c at the keyboard.  Catch it
     and send it along to the foreground job. */
void sigint_handler(int sig)
{
    pid_t pid;
    if ((pid = fgpid(jobs)) == 0) return;
    kill(-pid, sig);
    return;
}
```

```c

/* sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever the user types ctrl-z at the keyboard. Catch it and suspend the foreground job by sending it a SIGTSTP. */
void sigtstp_handler(int sig)
{
    pid_t pid;
    if ((pid = fgpid(jobs)) == 0) return;
    kill(-pid, sig);
    return;
}


/* sigchld_handler - The kernel sends a SIGCHLD to the shell whenever a child job terminates, or stops because it received a SIGSTOP or SIGTSTP signal. The handler reaps all available zombie children, but doesn't wait for any other currently running children to terminate. */
void sigchld_handler(int sig)
{
    int olderrno = errno;
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        struct job_t *job = getjobpid(jobs, pid);
        if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        } else if (WIFSIGNALED(status)) {
            if (WTERMSIG(status) == SIGINT){
                sio_puts("Job [");
                sio_putl(job->jid);
                sio_puts("] (");
                sio_putl(pid);
                sio_puts(") terminated by signal ");
                sio_putl(WTERMSIG(status));
                sio_puts("\n");
            }
            deletejob(jobs, pid);
        } else if (WIFSTOPPED(status)) {
            sio_puts("Job [");
            sio_putl(job->jid);
            sio_puts("] (");
            sio_putl(pid);
            sio_puts(") stopped by signal ");
            sio_putl(WSTOPSIG(status));
            sio_puts("\n");
            job->state = ST;
        }
    }
    /* other handlers can't overwrite the value of errno */
    errno = olderrno;
    return;
}

/* sigquit_handler - The driver program can gracefully terminate the child shell by sending it a SIGQUIT signal. */
void sigquit_handler(int sig)
{
    printf("Terminating after receipt of SIGQUIT signal\n");
    exit(1);
}

/* Signal - wrapper for the sigaction function. Different versions of Unix can have different signal handling semantics. Some older systems restore the action to default after catching signal, and some systems don't block signals of the type being handled. So use this. */
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;
    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* restart syscalls if possible */
    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

11 Malloc lab

```c
#define ALIGNMENT 8
#define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)

#define WSIZE 4 /* Word and header/footer size (bytes) */
#define DSIZE 8 /* Double word size (bytes) */
#define INITCHUNK (1<<6)
#define CHUNKSIZE (1<<12) /* Extend heap by this amount (bytes) */
#define LISTSIZE 20

#define MAX(x, y) ((x) > (y) ? (x) : (y))
/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))
#define PUT_ADD(p, bp) (*(unsigned int *)(p) = (unsigned int)(bp))
/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

#define PREV_ADD(bp) ((char *)(bp))
#define NEXT_ADD(bp) ((char *)(bp) + WSIZE)
#define PREV(bp) (*(char **)(bp))
#define NEXT(bp) (*(char **)(NEXT_ADD(bp)))

static void* extend_heap(size_t words);
static void* coalesce(void* bp);
static void append(void* bp, size_t asize);
static void delete(void* bp);
static void* place(void* bp, size_t asize);

static char* heap_listp;
char* segfree_list[LISTSIZE];

/* mm_init - initialize the malloc package. */
int mm_init(void) {
    for (int i = 0; i < LISTSIZE; i++) {
        segfree_list[i] = NULL;
    }
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1)
        return -1;
    PUT(heap_listp, 0); /* Alignment padding */
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); /* Epilogue header */
    /* Extend the empty heap with a free block of CHUNK bytes */
    if (extend_heap(INITCHUNK) == NULL)
        return -1;
    return 0;
}
static void* extend_heap(size_t words)
{
    char* bp;
    size_t size;
    size = ALIGN(words);

    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0)); /* Free block header */
    PUT(FTRP(bp), PACK(size, 0)); /* Free block footer */
```

```c
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */

    append(bp, size);

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}

static void* coalesce(void* bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) { /* Case 1 */
        return bp;
    }
    else if (prev_alloc && !next_alloc) { /* Case 2 */
        delete(bp);
        delete(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc) { /* Case 3 */
        delete(bp);
        delete(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else { /* Case 4 */
        delete(bp);
        delete(NEXT_BLKP(bp));
        delete(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    append(bp, size);
    return bp;
}

static void append(void* bp, size_t asize) {
    void* prev_p = NULL;
    void* curr_p = NULL;
    int lidx = 0;

    while (lidx < LISTSIZE - 1) {
        if (asize <= 1)
            break;
        asize >>= 1;
        lidx++;
    }
    prev_p = segfree_list[lidx];
    while (prev_p != NULL) {
        if (asize <= GET_SIZE(HDRP(prev_p)))
            break;
        curr_p = prev_p;
        prev_p = PREV(prev_p);
    }

    if (prev_p == NULL && curr_p == NULL) {
        PUT_ADD(PREV_ADD(bp), NULL);
        PUT_ADD(NEXT_ADD(bp), NULL);
        segfree_list[lidx] = bp;
    }
    else if (prev_p == NULL && curr_p != NULL) {
        PUT_ADD(PREV_ADD(bp), NULL);
        PUT_ADD(NEXT_ADD(bp), curr_p);
        PUT_ADD(PREV_ADD(curr_p), bp);
    }
    else if (curr_p == NULL) {
        PUT_ADD(PREV_ADD(bp), prev_p);
        PUT_ADD(NEXT_ADD(prev_p), bp);
        PUT_ADD(NEXT_ADD(bp), NULL);
        segfree_list[lidx] = bp;
    }
    else {
        PUT_ADD(PREV_ADD(bp), prev_p);
        PUT_ADD(NEXT_ADD(prev_p), bp);
        PUT_ADD(NEXT_ADD(bp), curr_p);
        PUT_ADD(PREV_ADD(curr_p), bp);
    }
    return;
}
static void delete(void* bp) {
    size_t size = GET_SIZE(HDRP(bp));
    int lidx = 0;

    while (lidx < LISTSIZE - 1) {
        if (size <= 1)
            break;
        size >>= 1;
        lidx++;
    }

    if (PREV(bp) == NULL && NEXT(bp) == NULL) {
        segfree_list[lidx] = NULL;
    } else if ((PREV(bp) == NULL && NEXT(bp) != NULL)) {
        PUT_ADD(PREV_ADD(NEXT(bp)), NULL);
    } else if ((NEXT(bp) == NULL)) {
        PUT_ADD(NEXT_ADD(PREV(bp)), NULL);
        segfree_list[lidx] = PREV(bp);
    } else {
        PUT_ADD(NEXT_ADD(PREV(bp)), NEXT(bp));
        PUT_ADD(PREV_ADD(NEXT(bp)), PREV(bp));
    }
    return;
}

static void* place(void* bp, size_t asize) {
    size_t csize = GET_SIZE(HDRP(bp));

    delete(bp);

    if ((csize - asize) > (2 * DSIZE)) {
        if (asize < 100) {
            PUT(HDRP(bp), PACK(asize, 1));
            PUT(FTRP(bp), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(bp)), PACK(csize - asize, 0));
            PUT(FTRP(NEXT_BLKP(bp)), PACK(csize - asize, 0));
            append(NEXT_BLKP(bp), csize - asize);
        }
        else {
            PUT(HDRP(bp), PACK(csize - asize, 0));
            PUT(FTRP(bp), PACK(csize - asize, 0));
            PUT(HDRP(NEXT_BLKP(bp)), PACK(asize, 1));
            PUT(FTRP(NEXT_BLKP(bp)), PACK(asize, 1));
            append(bp, csize - asize);
            return NEXT_BLKP(bp);
        }
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }

    return bp;
}

/* mm_malloc - Allocate a block by incrementing the brk pointer. Always allocate a block whose size is a multiple of the alignment. */
void* mm_malloc(size_t size) {
    size_t asize; /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    size_t tsize;
    int lidx = 0;
    char* bp = NULL;
    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE) {
        asize = 2 * DSIZE;
```

```c
    }
    else {
        asize = ALIGN(size + DSIZE);
    }
    tsize = asize;
    /* Search the free list for a fit */
    while (lidx < LISTSIZE && bp == NULL) {
        if (((tsize <= 1) && (segfree_list[lidx] != NULL)) || (
     lidx == LISTSIZE - 1)) {
            bp = segfree_list[lidx];
            while (bp != NULL) {
                if (asize > GET_SIZE(HDRP(bp)))
                    bp = PREV(bp);
                else
                    break;
            }
        }
        tsize >>= 1;
        lidx++;
    }

    /* No fit found. Get more memory and place the block */
    if (bp == NULL) {
        extendsize = MAX(asize, CHUNKSIZE);
        if ((bp = extend_heap(extendsize)) == NULL)
            return NULL;
    }
    bp = place(bp, asize);
    return bp;
}

/* mm_free - Freeing a block does nothing. */
void mm_free(void* ptr) {
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    append(ptr, size);
    coalesce(ptr);
}

/*  mm_realloc - Implemented simply in terms of mm_malloc and
     mm_free */
void* mm_realloc(void* ptr, size_t size) {
    void* new_ptr = ptr;
    int extendsize = 0;
    int sizesum = 0;
    size_t copysize = size;

    if (size == 0)  return NULL;
    if (copysize <= DSIZE) {
        copysize = 2 * DSIZE;
    } else {
        copysize = ALIGN(size + DSIZE);
    }

    if (GET_SIZE(HDRP(ptr)) >= copysize)
        return ptr;

    if (GET_ALLOC(HDRP(NEXT_BLKP(ptr))) && GET_SIZE(HDRP(
     NEXT_BLKP(ptr)))) {
        new_ptr = mm_malloc(copysize - DSIZE);
        memcpy(new_ptr, ptr, copysize);
        mm_free(ptr);
    } else {
        sizesum = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(
     ptr)));
        if (sizesum < (int)copysize) {
            extendsize = MAX((int)copysize - sizesum, CHUNKSIZE)
     ;
            if (extend_heap(extendsize) == NULL)
                return NULL;
        }
        delete(NEXT_BLKP(ptr));
        PUT(HDRP(ptr), PACK(sizesum + extendsize, 1));
        PUT(FTRP(ptr), PACK(sizesum + extendsize, 1));
    }
    return new_ptr;
}
```