Historical Development

**Nihilism** (circa 1945)

Individual programs load from punched card decks. Scheduling by signup sheet.

**Batch Processing** (circa 1955)

Groups of jobs submitted in a batch, still executed serially. OS is the program that reads the next job from tape and starts execution.

- CPU wasted during I/O bound jobs
- Short jobs delayed by long jobs

**Multiprogramming** (circa 1965)

Several jobs kept in memory. When running job blocks for I/O, CPU begins executing another job. Required hardware memory protection.

- From user’s point of view, equivalent to batch
- Compute bound jobs are favored

**Timesharing** (circa 1968)

Each user has interactive session with computer. *Time slicing* provides the illusion of a virtual machine.
Historical Development (cont)

Personal Computing (circa 1975)

Single-user interactive computing. Applicable mainly to microprocessor-based computers.

Client-Server model (circa 1982)

Small subset of OS functions contained in supervisor mode “kernel,” other functions migrate to user mode “servers.”

Distributed OS (circa 1985)

Extension of Client-Server model, client and server can be on different (network connected) computers.
OS Resources

Processes

• Process = job or program
• Requires “environment” information for preemption

Memory

• Allocation to processes
• Protection

Files

• Mapping files to disk space
• Directory structure
• Protection

Peripherals

• Printers
• Terminals
• Modems
• Backup devices
• etc.
OS Structure

Monolithic

OS is one large group of functions with little restriction on interaction other than a distinction between kernel mode function and user mode functions.

Layered

OS divided into Layers, each providing a particular service to higher layers. Interaction between layers is strictly controlled.

Virtual Machine

Multiprogramming provided by creating multiple emulated copies of the hardware. Any OS that would run on the bare hardware can be run on one of the virtual machines.

Client-Server

Small kernel provides communication services to client and server processes.
Hardware support for OS Features

The Interrupt Mechanism

- Scheduling needed because a multiprocessing system allows processes to be interrupted (suspended) and restarted at any time.

- Interrupts occur due to hardware events (completion of disk I/O, etc.) or software events (expiration of a timer, reception of a message, etc.).

- Instruction cycle is modified to include a check for pending interrupts after each instruction execution.

- *Interrupt vector* - an area of system memory containing a pointer to the *interrupt handler* for each type of interrupt.

- Interrupt handler first preserves processor state for the suspended process, then performs actions necessary to service the interrupt.

- Different type of interrupts may have different priorities: interrupt handlers may themselves be interruptable!
Hardware support for OS Features

Peripheral Handling: Direct Memory Access

- High speed peripheral devices have the potential to swamp the CPU with interrupts.
- Transfers to or from such devices can be grouped into blocks.
- The device driver (OS component that services the device) provides the memory addresses of the start/end of the memory area the device reads from/writes to, then gives the device permission to directly access that range of locations without further CPU involvement (hence Direct Memory Access or DMA).
- All the work to complete the transfer occurs between the device and the memory controller, with the CPU free to deal with other tasks.
- Cycle stealing - since the memory controller can handle only one request at a time, some memory requests from the CPU may be delayed by requests from the device performing DMA.
Other Hardware Support for the OS

• Dual mode operation
  - Monitor mode for execution of OS (also called Supervisor mode or Kernel mode)
  - User mode for execution of user programs

• Time slicing - prevents a "runaway" process from dominating CPU usage. Hardware timer needed to implement this feature.

• Memory protection
  - CPU provides Base Register and Limit Register.
  - When a process runs on the CPU, these registers contain the addresses of the range of memory locations that the process is allowed to access
  - All memory references are compared against the hardware registers.
  - References outside the allowed range cause a trap. (Segmentation fault, in Unix.)

• I/O protection - making I/O instructions "privileged", the OS manages shared access to I/O devices.
Processes

Process - an executing program plus an environment containing all information needed by the OS to reliably suspend and resume the execution of the program.

Process States

New - process is being initialized

Ready - the process is ready to run, but currently suspended

Running - the process is actually executing

Blocked - (also called "waiting") the process is suspended waiting on the completion of some external event

Terminated - the program has completed; the process is waiting to be "recycled" by the system.

Process Management

![Diagram showing process management]

Currently executing
P2 - others are ready or blocked
Process Control Block

The information needed to manage the execution of a process is stored in an OS data structure called the *process control block*. The OS maintains a copy of this structure for each process.

Contents:

- Process state
- Program counter
- Contents of all CPU registers, including status registers
- Scheduling information
- Memory management information
- Accounting information - disk quotas, time quotas, security
- I/O status information - open files, devices allocated to the process, etc.

Why do we need all of this?

- In order to suspend and resume the execution of a process, we must exactly restore the state of the hardware.
- Statistical information about the execution of processes is used for scheduling and system performance tuning
- Some information is needed to maintain the association between the process and the memory that belongs to it.
#ifndef _SYS_PROC_H
#define _SYS_PROC_H

#pragma ident"@(#)proc.h 1.8796/01/18 SMI" /* SVr4.0 11.65 */

#include <sys/time.h>
#include <sys/thread.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/timer.h>
#ifdef (i386) || defined(__i386)
#include <sys/segment.h>
#endif
#ifdef __cplusplus
extern "C" {
#endif

/*
 * One structure allocated per active process. It contains all
 * data needed about the process while the process may be swapped
 * out. Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped
 * out.
 */
typedef struct proc {
  /*
   * Fields requiring no explicit locking
   */
  struct vnode *p_exec; /* pointer to a.out vnode */
  struct as *p_as; /* process address space pointer */
  struct plock *p_lockp; /* ptr to proc struct's mutex lock */
  int p_pad; /* for backwards compatibility */
  kmutex_t p_crlock; /* lock for p_cred */
  struct cred *p_cred; /* process credentials */
  /
  * Fields protected by pidlock
  */
  int p_swapcnt; /* number of swapped out lwps */
  char p_stat; /* status of process */
  char p_wcode; /* current wait code */
  int p_wdata; /* current wait return value */
  pid_t p_ppid; /* process id of parent */
  struct proc *p_link; /* forward link */
  struct proc *p_parent; /* ptr to parent process */
  struct proc *p_child; /* ptr to first child process */
  struct proc *p_sibling; /* ptr to next sibling proc on chain */
  struct proc *p_psibling; /* ptr to prev sibling proc on chain */
struct proc *p_sibling_ns; /* ptr to siblings with new state */
struct proc *p_child_ns; /* ptr to children with new state */
struct proc *p_next; /* active chain link next */
struct proc *p_prev; /* active chain link prev */
struct proc *p_nextofkin; /* gets accounting info at exit */
struct proc *p_orphan;
struct proc *p_nextorph;
struct proc *p_pglink; /* process group hash chain link next */
struct proc *p_ppglink; /* process group hash chain link prev */
struct sess *p_sessp; /* session information */
struct pid *p_pidp; /* process ID info */
struct pid *p_pgidp; /* process group ID info */

/*
 * Fields protected by p_lock
 */
kcondvar_t p_cv; /* proc struct's condition variable */
kcondvar_t p_flag_cv;
kcondvar_t p_lwpexit; /* waiting for some lwp to exit */
kcondvar_t p_holdlwps; /* process is waiting for its lwps */
  /* to be held. */
kcondvar_t p_flock; /* asleep in closef */
u_short p_prwant; /* /proc: process wanted for locking */
u_int p_flag; /* protected while set. */
  /* flags defined below */
clock_t p_utime; /* user time, this process */
clock_t p_stime; /* system time, this process */
clock_t p_cutime; /* sum of children's user time */
clock_t p_cstime; /* sum of children's system time */
caddr_t *p_segacct; /* segment accounting info */
caddr_t p_brkbase; /* base address of heap */
u_int p_brksize; /* heap size in bytes */
/*
 * Per process signal stuff.
 */
k_sigset_t p_sig; /* signals pending to this process */
k_sigset_t p_ignore; /* ignore when generated */
k_sigset_t p_siginfo; /* gets signal info with signal */
struct sigqueue *p_sigqueue; /* queued siginfo structures */
struct sigqhdr *p_sigqhdr; /* hdr to sigqueue structure pool */
u_char p_stopsig; /* jobcontrol stop signal */
/*
 * Per process lwp and kernel thread stuff
 */
int p_lwptotal; /* total number of lwps created */
int p_lwpcnt; /* number of lwps in this process */
int p_lwprcnt; /* number of not stopped lwps */
int p_lwpbcount; /* number of blocked lwps. kept */
  /* consistent by sched_lock() */
int p_zombcnt; /* number of zombie LWFs */
kthread_t *p_tlist; /* circular list of threads */
kthread_t *p_zomblist; /* circular list of zombie LWFs */
/*
 * XXX Not sure what locks are needed here.
 */
k_sigset_t p_sigmask; /* mask of traced signals (/proc) */
k_fltset_t p_fltmask; /* mask of traced faults (/proc) */
struct vnode *p_trace; /* pointer to primary /proc vnode */
struct vnode *p_plist; /* list of /proc vnodes for process */

struct proc *p_rlink; /* linked list for server */
kcondvar_t p_srwcharn_cv;
u_int p_stksize; /* process stack size in bytes */
/
* Microstate accounting, resource usage, and real-time profiling */
hrt ime_t p_mstart; /* hi-res process start time */
hrt ime_t p_mterm; /* hi-res process termination time */
hrt ime_t p_mreal; /* elapsed time sum over defunct lwps */
hrt ime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
struct lrusage p_ru; /* lrusage sum over defunct lwps */
struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
int p_rprof_timerid; /* interval timer's timeout id */
u_int p_defunct; /* number of defunct lwps */
/
* profiling. A lock is used in the event of multiple lwp's
* using the same profiling base/size.
*/
kmutex_t p_pflock; /* protects user pr_base in lwp */
/
* The user structure
*/
struct user p_user; /* (see sys/user.h) */

u_int p_swrss; /* resident set size before last swap */
struct aio *p_ai o; /* pointer to async I/O struct */
struct timerstr *p_itimer; /* posix timers */
k_sigset_t p_notifsigs; /* signals in notification set */
kcondvar_t p_notifcv; /* notif cv to synchronize with aslwp */
int p_alarmid; /* alarm's timeout id */
}

proc_t;

#define PROC_T /* headers relying on proc_t are OK */

/* active process chain */
extern proc_t *practive;

/* Well known processes */

extern proc_t *proc_sched; /* memory scheduler */
extern proc_t *proc_init; /* init */
extern proc_t *proc_pageout; /* pageout daemon */
extern proc_t *proc_fsflush; /* filesystem sync-er */

#ifdef __cplusplus
}
#endif
#endif /* _SYS_PROC_H */
Operations on Processes

Creation

Typically, a process must be created by another process. At boot time, a special initial process is constructed and started which is the "ancestor" of all other processes.

There are several styles of process creation:

- Child shares parent’s address space vs. child has its own address space.
- Child has access to parent’s resources vs. child must request its own resources
- Parent suspends until child completes vs. parent continues execution
- Child runs same program as parent vs. child runs some other program

Example: running a program from a Unix GUI system

- A process called the window manager is responsible for the GUI
- User double-clicks an icon on the "desktop"
- Window manager spawns a new process by executing the fork( ) system call and gives it the pathname of the executable associated with the icon
- Child process executes the exec( ) system call to switch to the executable image of the requested program
- Parent continues execution
Operations on Processes (continued)

Termination

Processes terminate for several reasons:

• The program that the process executes has completed and requested termination via a system call.
• A fatal error was encountered.
• The process’s parent chooses to terminate it.
• The parent itself has been terminated.

In all cases, several things must happen:

• Open files belonging to the process must be closed
• Devices that are allocated to the process must be returned to the free pool (ex: tape drives)
• Memory belonging to the process is returned to the system
• The OS data structures must be updated to reflect the removal of the process control block for the process.

Context Switch

There are various activities that must occur when one process is suspended and another is allowed to execute.

• Copying info about hardware state to suspended processes PCB and updating other info kept there
• Copying PCB info from resuming process to hardware, etc.
• OS data structure updated to reflect changed process status
The availability of multiple processors and the cost of context switching leads to a need for more efficient ways to support cooperation between processes.

A thread is an independently schedulable entity which shares an address space and system resources with other threads belonging to the same process. (Also called lightweight processes.)

- Each thread runs sequentially
- Per thread program counters and stacks
- Global variables, open descriptors, signals etc. are shared

Threads allow the combination of sequential programming with blocking system calls (which are easier to program than the non-blocking alternative) with performance gains due to parallel execution.
Interprocess Communication (IPC)

Many operating systems provide a mechanism that allows two processes to exchange information.

Possible implementations:

- Shared memory based implementation, e.g. mailboxes
- Message passing implementation

Requirements:

- Processes must be uniquely identified (we need this for other reasons, too)
- Methods for processes to initiate and accept connections
- Methods for producing and consuming information
- Methods for processes to gracefully terminate connections

Other issues:

- Communication must be "secure" (i.e. only intended recipient can access transferred information)
- Capacity of links (how many messages can be "in transit" at once)
Berkeley Unix IPC

socket—an interprocess communication endpoint supporting operations similar to those supported by files opened with the open system call.

\[
s = \text{socket (domain, type, protocol)};
\]

domain—the “communication domain” to be used by the socket. One of:

- **AF_UNIX**—sockets used for processes on the same computer
- **AF_INET**—processes on different computers.

type what kind of service the process wants on the socket. Choices are:

- **SOCK_STREAM**—two way byte stream
- **SOCK_DGRAM**—communication consists of individual messages.

protocol—what protocol is to be used for communication over the socket. 0 is the Internet Protocol (IP).
Connecting Sockets

In order for some other process to connect to us, it needs to know what machine we are on and also how to specify the process that the socket it is trying to connect to. We must request a “network visible process" address for our socket. This address is called a \textit{port number}, and we get it by doing a \textit{bind} operation:

\begin{verbatim}
status = bind (socket, name, namelen);
\end{verbatim}

\textbf{socket} a socket descriptor returned by a previous call to the socket system call.

\textbf{name} a structure describing the port we want to bind to. \textit{bind()} expects a structure of type sockaddr, but for the AF_INET domain, we use a sockaddr_in structure, as defined in /usr/include/netinet/in.h

\begin{verbatim}
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
\end{verbatim}

\textbf{sin_family}—“protocol family” to use, same as domain

\textbf{sin_port}—the port address we are requesting. Anything > 1024 is recognized.

\textbf{in_addr}—the IP address to respond to if more than one exists for the machine we’re on. INADDR_ANY responds to any address the machine recognizes.

\textbf{sin_zero}—padding.

\textbf{namelen}—size of the structure passed in name.
Waiting for Connections

Once a socket is created and bound to a port, a server will typically wait for other processes to connect to it using the `listen()` system call.

\[
\text{listen(socket, n);} \\
\]

This notifies the system that we are willing to accept up to \( n \) connection requests on `socket`. If multiple requests are made before we handle them, they are queued.

*Listen* does not block; it merely notifies the OS of our address. In order to wait for connections, we use the `accept()` system call.

\[
\text{new_sock = accept (socket, addr, addrlen);}
\]

`socket`--the socket that we did the listen call on

`addr`--information about the connect that is created. Can be NULL if you aren’t interested.

`addrlen`--length in bytes of the structure passed in `addr`.

`new_sock`--socket created by the accept. The original socket is still available for new connect attempts.
Client Side Connection

There is a bit less work involved on the client side. Socket() is used to create a socket, then connect() is used to connect to an existing socket at the other end:

\[
\text{status} = \text{connect (socket, name, namelen);} \\
\]

socket--a socket created by a call to socket.

name--structure describing the host and port to connect to. Of type sockaddr_in. The in_addr field can be filled in by using gethostbyname().

namelen--the length of the structure passed to name.

\[
\text{host_info} = \text{gethostbyname(hostname);} \\
\]

hostname--a character string containing the name of the host to connect to.

host_info--a structure of type hostent (defined in /usr/include/netdb.h). The h_addr field can be copied to the sin_addr field of the sockaddr_in structure passed to connect.
Sending/Receiving via sockets

Sockets created with type `SOCK_STREAM` behave just like files opened with the `open()` system call. The `read()` system call is used to receive bytes, the `write()` system call is used to send bytes.

```
nbytes = read (new_sock, buf, size);
```

new_sock--socket created by an accept or socket/connect.
buf--buffer to hold the incoming bytes.
size--maximum number of bytes to put into buf.
nbytes--number of bytes actually read.

```
nbytes = write (new_sock, buf, size);
```

new_sock--socket created by an accept or socket/connect.
buf--buffer containing string to write.
size--number of bytes contained in buf.
nbytes--number of bytes actually written.
Socket Summary

Server

lsoc=socket();
bind (lsoc, port);
listen(lsoc)

Network

Port

Client

comm=socket()
connect (comm, name)
write (comm, buf)
read (comm, buf)
close (comm)

comm=accept
read(comm, buf)
write(comm, buf)
close (comm)
Scheduling Criteria

- Fairness - processes should receive an equal share of the processor if at all possible. (Modulo priority assignments)
- Efficiency - percentage of time CPU is busy doing useful work.
- Response time - performance for interactive tasks.
- Waiting time - how long a process waits in the "ready queue" to be allowed to run (note that this is different from waiting for I/O to complete - the process is not "ready" in that case)
- Turnaround time - submission to completion time for batch jobs.
- Throughput - number of jobs completed per hour. (Also a batch measure.)
Scheduling Algorithms

**Round Robin**

Preemptive. Access to processor rotates around a circular queue of processes.

**Priority Scheduling**

Preemptive. Each process is assigned a priority. Runnable process with highest priority is given access to processor. Priorities can be separated into classes, with each class having a separate queue.

**Shortest Job First**

Non-preemptive. Run the job with the shortest completion time next. Provably optimal for batch turnaround.

**Deadline Based Scheduling**

Preemptive. Each process has a real-time deadline. Scheduler allows process in greatest danger of missing deadline to run.

**Multilevel Queue Scheduling**

Processes are divided into classes, each with their own queue. Various approaches can be used to determine how much of the processor time is allocated to each queue.

Processes typically do not move between queues.
Multilevel Feedback Queue Scheduling

Like Multilevel Queue, but processes can be moved from queue to queue depending on their CPU-burst behavior.

Typically these systems weight their priorities toward interactive processes rather than CPU-bound processes to optimize response time.

If a process in a high-priority queue begins to have longer CPU bursts, it may be moved to a lower priority queue.

To prevent "starvation," processes in low priority queues with long average wait times may be moved up to a higher priority queue.

Highly configurable, but complicated.

Swapping

Swapping manages the "degree of multiprogramming," i.e. the number of processes that are currently active in the system. If some processes are swapped out, a second level of scheduling is necessary. (Medium-term scheduling.)
Scheduling Examples

Shortest Job First

In the pre-emptive case, what we really need to know is which job has the shortest next CPU burst (i.e. shortest stretch of continuous computation before experiencing an I/O wait or termination).

Suppose we have the following situation:

<table>
<thead>
<tr>
<th>Process</th>
<th>Next CPU burst Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>4</td>
</tr>
<tr>
<td>P₁</td>
<td>16</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td>8</td>
</tr>
</tbody>
</table>

The processes will run in the following order: P₂, P₀, P₃, P₁.

The average waiting time is:

\[(0 + 2 + 6 + 14) / 4 = 5.5\]

For FCFS, the average waiting time would have been

\[(0 + 4 + 20 + 22) / 4 = 11.5\]

If we allow pre-emption, we must also take into account the arrival time of the processes.
Scheduling Examples

In practice, in a multiprogramming environment, we don’t know the length of the next CPU burst for a process; the best we can do is predict it based on past performance.

If we keep track of the entire burst history of a process, processes that go through phases (e.g. doing a lot of I/O early, then doing a long computation on the data) can be inaccurately categorized.

We can age the burst length estimate as follows: let $t_n$ be the length of the $n$th CPU burst for a process, and let $\tau_{n+1}$ be the predicted value of the next CPU burst. Then for $0 \leq \alpha \leq 1$ define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

This is an exponential average. The parameter $\alpha$ controls how much emphasis we place on older versus newer information. Typical value is $1/2$.

If $\alpha = 1$, then only the most recent burst is considered. As $\alpha$ decreases toward 0, previous bursts are given more and more weight.
Performance Evaluation

Given a choice among scheduling algorithms, how do we decide which to use?

**Deterministic Analysis**

Uses specific cases of job arrival and burst times and compares algorithms the way we did in our examples.

Not a generally valid method; specific cases can’t be generalized.

**Queueing Model Analysis**

A computer system can be viewed as a connected network of queues and processes that service the queues.

Service times and queue arrival times can be modeled with appropriate probability distributions.

A branch of mathematics called Queueing Theory allows us to mathematically determine the properties (such as waiting time, throughput, etc.) of such a system.

Limitation: actual arrival and service time distributions may not match the probability distributions very well.

**Simulation**

Construct a program that simulates the operation of the computer/scheduling algorithm.

Run the simulation with recorded or synthesized workloads and compare the performance of different systems. Level of detail of simulation is limited.
System example: Unix Processes

A Unix process image has three parts:

- text segment - the executable code for the program
- data segment - any data which is initialized when the process starts plus an indicator for the kernel of how much space to reserve for uninitialized data (bss)
- stack segment - dynamically allocated by the kernel, space for procedure call frames and locally declared data space.

System keeps process management info in two structures:

- Process structure
  - Scheduling information
  - Process identification
  - Memory management information
  - Synchronization
  - Signal management
  - Resource accounting
  - Timer management

- user (u) structure
  - User- and kernel-mode execution states
  - System call state
  - Descriptor table
  - Accounting information
  - Resource controls
  - Kernel execution stack
Unix Scheduling

Round-Robin with Multilevel feedback

Select the runnable process with the highest priority.
If there is a tie, select the process that has been waiting longest.
Kernel recomputes the priority of all processes at least once per second, using a formula based on recent CPU usage.
CPU usage is *aged*, so that a process that received a large share of CPU in the past is not penalized forever.
Separate queues are maintained for processes executing in kernel mode and processes executing in user mode.

\[
\text{new priority} = \text{base} + \frac{\text{CPU usage}}{2}
\]

\[
\text{base} = 50, \text{ but can be increased with } \text{nice}().
\]
Suppose OS provides mailboxes for communication between processes. A Mailbox consists of an index variable and a set of slots for messages.

To deposit a message in a mailbox, a process must:

- Copy the message into the slot indicated by Index
- Increment index to the next available slot.

### Scenario 1

<table>
<thead>
<tr>
<th>Index</th>
<th>M0</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Process 1 actions**
- Read Index value (0)
- Place msg P1 in slot M0
- Change Index to 1

**Process 2 Actions**
- Read Index value (1)
- Place msg P2 in slot M1
- Change Index to 2

### Scenario 2

<table>
<thead>
<tr>
<th>Index</th>
<th>M0</th>
<th>M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Process 1 Actions**
- Read Index value (0)
- Place msg P1 in slot M0

**Process 2 Actions**
- Read Index value (0)
- Place msg P2 in slot M0
- Change Index to 1
Mutual Exclusion

Mutual exclusion is the capability of a process to exclude all other processes from accessing a shared area of memory for a period of time while it uses it.

A critical section is a portion of a program where shared memory is accessed; such program sections must be protected by some mutual exclusion mechanism.

Desired Properties

- The mechanism must work (!) (Mutual Exclusion Property)
- No assumptions can be made about the speed of the CPU
- No process can block others except via the Mutual Exclusion mechanism (Progress Property)
- Starvation must be impossible (Bounded Waiting Property)

Typically, the only assumption we can make in constructing a solution to this "critical section problem" is that a single machine instruction is executed atomically, i.e. once a machine instruction begins execution, the process that the instruction belongs to cannot be suspended before that instruction completes.
Mutual Exclusion Mechanisms

Many mechanisms use the technique of **busy waiting** - i.e. the process waiting for access to its critical section runs in a loop which does nothing but check the condition it is waiting for. This is not a desirable thing.

- Disabling interrupts - too dangerous for practical use
- Lock variables - doesn’t work; you need lock variables for the lock variables, *ad infinitum*.
- Strict Alternation - access to critical section goes turn for turn between the two processes competing for the shared memory. Processes can block even when the other process is not in its critical section. Requires busy waiting. (Alg. 1)
- Peterson’s solution - essentially requires three lock variables, and handles only two competing processes. Requires busy waiting. (Alg. 3)
- Test and Set Lock - hardware solution, process uses TSL instruction to set flag to 1 before entering critical region, but only enters if previous value was 0. When leaving, flag is set to 0. Requires busy waiting.
- Sleep/Wakeup primitives - essentially, a mechanism for a process to request that it be blocked and for one process to unblock another sleeping process. No busy waiting.
- Semaphores - generalization of Sleep/Wakeup which counts pending wakeup messages.
# Peterson’s Solution

```c
#define N 2 /* But really only works for 2 */

int turn; /* Whose turn is it? */
int interested[N]; /* All values initially false */

void enter_region (int process) /* process == who is entering */
{
    int other; /* Index of the other process */

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other])
        ; /* Busy waiting loop */
}

void leave_region (int process) /* process == who is leaving */
{
    interested[process] = FALSE;
}

Example: Both call enter_region at about same time

Process 0 actions Process 1 actions

Set interested[0] Set interested[1]
Set turn to 0 Set turn to 1

Enter while and exit Enter while and wait
immediately, because turn immediately, because turn
is != 0 == 1
```
The Bakery Algorithm

// Processes share the following common data structures:
boolean choosing[n]; // Initially all false
int number[n]; // Initially all 0

The notation (a, b) < (c, d) is true if a < c or if a == c and b < d

// Process P_i runs the following code:
repeat
    choosing[i] = true;
    number[i] = max(number) + 1; // 1 higher than the current highest value
    choosing[i] = false;
    for (int j = 0; j < n; j++) {
        while (choosing[j]); // Do nothing
        while (number[j] != 0 && (number[j], j) < (number[i], i)); // Do nothing
    }

    // Critical section code goes here

    // Leave critical section
    number[i] = 0;

    // Remainder section
until false;
The Unix shared memory facility

Unix provides the capability for processes to set up shared memory areas through the `shmget(2)`, `shmat(2)`, and `shmctl(2)` system calls. The basic steps are:

- Get an identifier for the shared memory area using a key that all "sharers" have access to. (`shmget( )`)
- "Attach" to the memory area -- i.e. get a pointer to it so you can actually use the memory -- using the identifier (`shmat( )`)
- Use `shmctl( )` to examine or alter permissions, lock the memory, etc.
- Use the memory
- Use `shmdt( )` to "detach" from the memory (if you weren’t the process that created it).
- Use `shmctl( )` to deallocate the shared area when you’re done with it (only done by the process that created it).

The key can be:

- created by a parent that passes it to its children
- agreed upon in advance and kept in a common header file
- generated by a "server" process and stashed in a file for clients to retrieve
- generated from a pathname and a "project id" using the `ftok(2)` system call.
Shared memory system calls

The following header files are needed:

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```c
int shmget(key_t key, int size, int flag);
```

- **key** An integer value, as discussed above, can be IPC_PRIVATE if the returned identifier can be passed to other processes that will use the shared memory.
- **size** Minimum number of bytes to allocate for the shared memory area.
- **flag** Bitflag specifying how the memory is to be accessed. Combine SHM_R (read access) SHM_W (write access) and IPC_CREAT (create a new memory area) with bitwise or ( | ) to get desired value.

The return value is the system ID for the shared memory area, and is used like a file descriptor for other operations on the memory.

```c
void *shmat(int shmid, void *addr, int flag);
```

- **shmid** the shared memory ID returned by shmget()
- **addr** Should be 0, allowing the kernel to determine what address within the shared memory is returned.
- **flag** 0, unless you want to attach for read only access, in which case, you should pass SHM_RDONLY.

Return value is the address of the shared memory area, which can be typecast to the appropriate data type for your program.
Shared memory system calls (cont)

int shmdt(void *addr);

addr The address of the shared memory segment to detach from (which was returned by shmat( ))

The return value is 0 if the call succeeded, -1 if not.

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

shmid The shared memory ID for the segment to access, as returned by shmget( )

cmd Constant specifying the command to apply. IPC_RMID causes the segment to be removed from the system. See shmctl(2) for other options.

buf Used by other commands, can be 0 for IPC_RMID

The data structures and constants associated with these system calls can be found in /usr/include/sys/shm.h
Example: a naive shared buffer

The following example shows how to use shared memory to build a shared array of ints that behaves like a circular queue. It uses no mutual exclusion, and so could experience race conditions.

// File: shared_buffer
// Programmer: Lewis Barnett

#ifndef __shared_buffer__
#define __shared_buffer__

// Maximum size for buffer
#define SIZE_LIMIT 10

// Data used for a naive shared buffer
// It would be difficult to keep this structure consistent.
struct shared_buffer {
    int slot[SIZE_LIMIT];
    int first_used; // The "lowest" index of an occupied slot
    int number_full; // The number of slots currently occupied
};

// Used in calls to Unix shared memory system calls. Please modify
// this to an integer > 1024 of your choosing to avoid colliding
// with your classmates.
#define SHBUF_KEY5643

// Add a value to the shared buffer. Returns true if add succeeds,
// false otherwise.
bool add_value(int value, shared_buffer *sbp);

// Remove a value from the shared buffer. Return value of -1
// indicates attempt to remove from empty buffer.
int remove_value(shared_buffer *sbp);

#endif
#include <stdio.h>
#include <iostream>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>
#include "shared_buffer.h"

#define SHM_MODE (SHM_R | SHM_W | IPC_CREAT) // User read/write

main() {
    int shmid;
    shared_buffer *sbp;

    // Create the area of shared memory -- we must specify the size, // and have the IPC_CREAT bit set in our flag argument.
    if ((shmid = shmget(SHBUF_KEY, sizeof(shared_buffer), SHM_MODE)) < 0) {
        perror("shmget failed");
        exit(1);
    }

    // Attach to the shared memory so we can access it.
    if ((sbp = (shared_buffer *) shmat(shmid, 0, 0)) == (void *) -1) {
        perror("shmat failed");
        exit(1);
    }

    // Initialize shared memory.
    sbp->first_used = 0;
    sbp->number_full = 0;
    for (int i = 0; i < SIZE_LIMIT; i++){
        sbp->slot[i] = 0;
    }

    // Monitor the status of the data in shared memory. This can // be handy for debugging purposes.
    int choice;
    cout << "Choose: 0 - print status; 1 - exit: ";
    cin >> choice;
    cout << endl;
while (choice != 1) {
    cout << "Status: " << endl;
    cout << "first occupied slot: " << sbp->first_used
         << " current size: " << sbp->number_full << endl;
    for (int i = 0; i < SIZE_LIMIT; i++) {
        cout << "i = " << i << " slot[i] = " << sbp->slot[i]
             << endl;
    }

    cout << "Choose: 0 - print status; 1 - exit: ";
    cin >> choice;
    cout << endl;
}

// Delete the shared memory.
if (shmctl(shmid, IPC_RMID, 0) < 0) {
    perror("shmctl error");
}
}
// File: shbuf_client.cpp
// Author: Lewis Barnett

#include <stdio.h>
#include <iostream>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>
#include <string>
#include "shared_buffer.h"

// Bitflag describing the kind of access we want to the shared memory.
#define SHM_MODE (SHM_R | SHM_W) // User read/write

int string_to_num(string s); // A utility function

int main(int argc, char *argv[]) {

    // We expect our process ID (a value between 0 and SIZE_LIMIT - 1
    // to be passed as a command line argument.
    if (argc < 2) {
        cerr << "Usage: shbuf_client proc_id" << endl
             << "\twhere proc_id is the ID for this process."
             << endl << endl;
    exit(1);
}

    string idstr = argv[1];
    int my_id = string_to_num(idstr);
    cout << "Shared buffer client (id = " << my_id << ") running."
         << endl;

    int shmid;
    shared_buffer *sbp;

    // We assume that shbuf_server has already run, so we don't need
    // to create the memory area. This call returns the shared
    // memory ID associated with our key value. We pass 0 as the
    // size of the shared memory area because we assume it
    // already exists.
    if ((shmid = shmget(SHBUF_KEY, 0, SHM_MODE)) < 0) {
        perror("shmget failed");
        exit(1);
    }
// Attach to the shared memory -- following this call, sbp
// points to the shared memory area. shmat returns -1 if
// something goes wrong.
if ((sbp = (shared_buffer *) shmat(shmid, 0, 0))
    == (void *) -1) {
    perror("shmat failed");
    exit(1);
}

int choice;
cout << "Choose: 0 - add number; 1 - remove number; 2 - exit: ";
cin >> choice;
cout << endl;

while (choice != 2) {
    if (choice == 0) {
        if (add_value(my_id, sbp)) {
            cout << "Value added successfully.\n" << endl;
        } else {
            cout << "Sorry, buffer is full...\n" << endl;
        }
    } else {
        int val = remove_value(sbp);
        cout << "Got value: " << val << endl;
    }

cout << endl
    << "Choose: 0 - add number; 1 - remove number; 2 - exit: ";
cin >> choice;
cout << endl;
}

shmdt((char *) sbp);

int string_to_num(string s) {
    // Omitted for the sake of brevity.
}
// File: sbp_lib.cpp
// Author: Lewis Barnett
#include "shared_buffer.h"

// Add a value to the shared buffer. Returns true if add succeeds,
// false otherwise.
bool add_value(int value, shared_buffer *sbp){
    if (sbp->number_full < SIZE_LIMIT) {
        int next_slot =
            (sbp->first_used + sbp->number_full) % SIZE_LIMIT;
        sbp->slot[next_slot] = value;
        sbp->number_full++;
        return true;
    } else {
        return false;
    }
}

// Remove a value from the shared buffer. Return value of -1
// indicates attempt to remove from empty buffer.
int remove_value(shared_buffer *sbp){
    if (sbp->number_full > 0) {
        int retrieved_value =
            sbp->slot[sbp->first_used];
        sbp->slot[sbp->first_used] = 0;
        sbp->first_used = (sbp->first_used + 1) % SIZE_LIMIT;
        sbp->number_full--;
        return retrieved_value;
    } else {
        return -1;
    }
}
Sleep Wakeup Example
The Producer/Consumer Problem

#define N 100 /* Buffer size */

int count = 0; /* Number of items in buffer */

void producer (void)
{
    int item;

    while (TRUE) {
        produce_item (&item);
        if (count == N) /* If buffer is full */
            sleep();
        enter_item (item);
        count = count + 1;
        if (count == 1) /* If buf was empty */
            wakeup (consumer);
    }
}

void consumer (void)
{
    int item;

    while (TRUE) {
        if (count == 0)/* If buffer is empty */
            sleep();
        remove_item (&item);
        count = count - 1;
        if (count == N - 1)
            wakeup (producer);
        consume_item (item);
    }
}
Producer/Consumer race condition

Initially, assume the buffer is empty.

Producer process                      Consumer process

Test count == 0 => true  

Test count == N => false

enter_item(&item)

count = count + 1

Test count == 1 => true, so

    send wakeup (consumer)

    sleep () -- misses wakeup

Producer fills buffer and

sleeps

Patches to this solution (e.g. wakeup waiting bit) must depend on
the number of processes sharing the critical resource.
Semaphores

Proposed by Edsger Dijkstra (1965).

A semaphore consists of a shared variable and a queue of processes waiting on the variable. There are two operations:

**down (s)**
- If \( s = 0 \), sleep on \( s \)
- decrement \( s \).

**up (s)**
- increment \( s \)
- if the process queue waiting on \( s \) is not empty, wake up one of the processes and allow it to complete its down operation.

Notes:
- down() and up() must be executed indivisibly
- if processes are waiting on \( s \), an up() has the effect of waking up a process but leaving the semaphore value at 0
- no process ever blocks doing an up()
#define N 100       /* Buffer size */

semaphore mutex = 1;   /* Controls access to critical region */
semaphore empty = N;   /* Counts empty buffer slots */
semaphore full = 0;    /* Counts full buffer slots */

void producer (void)
{
    int item;

    while (TRUE) {
        produce_item (&item);
        down (&empty);   /* We intend to fill a slot */
        down (&mutex);   /* Request access to critical section */
        enter_item (item);
        up (&mutex);     /* Leave critical section */
        up (&full);      /* Increment full count */
    }
}

void consumer (void)
{
    int item;

    while (TRUE) {
        down (&full);   /* We intend to empty a slot */
        down (&mutex);  /* Request access to critical section */
        remove_item (&item);
        up (&mutex);    /* Leave critical section */
        up (&empty);    /* Increment empty count */
        consume_item (item);
    }
}

• empty must be > 0 before producer accesses CS - it is not altered until the consumer has completed its CS
• full must be > 0 before consumer accesses CS - it is not altered until producer has completed its CS
Other Mutual Exclusion Mechanisms

Monitors

A programming language construct, similar to a module. Only one process at a time can execute a function contained in the module. Needs primitives wait() and signal(), analogous to sleep and wakeup.

In the producers/consumers example, if all access to the shared buffer occurs through functions in a monitor, no race conditions can occur.

Assumes all participating processors have access to shared memory. Not many languages support monitors.

Message Passing

Processes communicate by sending and receiving messages through a channel. Buffered channel is convenient, but not necessary. No shared memory assumed.

Producers/consumers assumes that there are N “messages” which rotate from consumer to producer. Consumer initially sends N empty messages to producer, which fills and returns the messages. When consumer consumes one, it sends back the empty message to the producer.
The Dining Philosophers Problem

Statement

N philosophers are seated at a round table. Each place at the table is set with a dish of Beef Chow Mein from Lee Ho Fook’s in Soho*, and one chopstick. A philosopher may be either eating or thinking. In order to eat, a philosopher must hold two chopsticks. Is there a program which, if used by all philosophers, allows all to eventually eat without anyone starving?

Pitfalls

• Deadlock - all philosophers hold one chopstick, and wait to pick up the other

• Starvation - some philosopher is never able to get two chopsticks at one time.

• Non-optimal use of resources - there is an easy solution which allows only one philosopher to eat at a time. It should be possible for N/2 to eat at one time.

* I have taken some liberties with the usual anecdote that illustrates this problem...
int state[N]; /* One of THINKING, HUNGRY, EATING */
semaphore mutex = 1;
semaphore s[N]; /* One per philosopher, initially 0 */

void philosopher (int i)/* i is philosopher’s ID # */{
    while (TRUE){
        think();    /* A random wait */
        take_chopsticks(i); /* Getting hungry - get chopsticks */
        eat ();    /* Another random wait */
        put_chopsticks(i); /* Release resources */
    }
}

void take_chopsticks (int i)
{
    down (&mutex);
    state[i] = HUNGRY; /* Complain about growling stomach */
    test(i); /* Try to grab the chopsticks */
    up (&mutex);
    down (&s[i]); /* Blocks until put_chopstick() calls by eating neighbors occur. */
}

void put_chopsticks (int i)
{
    down (&mutex);
    state[i] = THINKING;
    test(LEFT); /* Do neighbors want the chopsticks? */
    test(RIGHT);
    up (&mutex);
}

void test(int i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        state[i] = EATING;
        up (&s[i]); /* Releases phil. who had blocked */
    }
}
Creating multiple processes

To create multiple processes from a single process, we use the `fork()` system call.

fork creates an exact duplicate of the calling process which continues execution from the point at which the fork call was made.

In the original process, the fork call returns the process id of the “child” process.

In the “child” process, the fork call returns 0.

```c
#include <errno.h>

/* A typical use of fork */
main ()
{
  int pid;

  if ((pid = fork()) == -1) {
    fprintf (stderr, "Ouch!  Fork failed!\n");
    perror();
    exit(1);
  } else if (pid == 0) {
    printf ("I’m the child!  My process id is %d\n",
             getpid());
    exit(0);
  } else {
    printf ("I’m the parent!  The child’s pid is %d\n", pid);
    wait();  /* Wait for child to terminate */
    exit();
  }
}
```
Unix Semaphores

Unix semaphores are allocated using the `semget()` system call, which returns an integer descriptor which identifies an array containing the requested number of semaphores. The descriptor is used in subsequent semaphore operations.

```c
s = semget (IPC_PRIVATE, n, SEM_A|SEM_R);
```

- IPC_PRIVATE - semaphore is shared by children of single parent
- n - number of semaphores in the group
- SEM_A - permission to alter granted
- SEM_R - permission to read granted

`s` indexes a table of structures:

```
<table>
<thead>
<tr>
<th>s</th>
<th>Control structure</th>
<th>Semaphore Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>
```

The `semctl()` system call allows the values of individual semaphores to be read or written.

```c
arg.val = 1;
semctl (s, 0, SETVAL, arg);
```

arg is of type `semun`

```c
union semun {
    int val;
    struct semid_ds *buf;
    ushort array;
};
```
The part of the union used depends on the requested command. It can be either a single value, a pointer to space for returned status information, or an array of values.

- `s` is the semaphore group identifier
- `0` is the index of the semaphore within that group to access
- `SETVAL` directs `semctl` to set the value of the semaphore `s[0]`
- `arg.val` contains the value to give to semaphore `s[0]`.

Other possible commands:

- **GETVAL** - return the value of the specified semaphore
- **IPC_RMID** - remove the specified semaphore from the system semaphore table

To perform various semaphore operations on a semaphore, use the `semop()` system call:

```c
sop.sem_num = 0;
sop.sem_op = -1;
sop.sem_flg = 0;
semop (s, &sop, 1);
```

- `s` is the identifier for the semaphore group.
- `sop` is a pointer to an array of structures of type sembuf, defined in `/usr/include/sys/sem.h`. It includes the index of the semaphore of interest, the code for the operation to perform, and a flag which can be used to specify further information about the operation.
The final argument is a count of the number of sembuf structures in the array pointed to by sop.

The sem_op field of sop is interpreted as follows:

- \( \text{sem} \_\text{op} < 0 \) - if the current value of the specified semaphore is \( \geq |\text{sem} \_\text{op}| \), the absolute value of \( \text{sem} \_\text{op} \) is subtracted from the current value.

- \( \text{sem} \_\text{op} < 0 \) and current semaphore value is \( < |\text{sem} \_\text{op}| \) - return immediately if \( \text{sem} \_\text{flg} \& \text{IPC} \_\text{NOWAIT} \) is true, otherwise, sleep until the value of the semaphore exceeds the requested value.

- \( \text{sem} \_\text{op} > 0 \) - add the value of \( \text{sem} \_\text{op} \) to the value of the semaphore and return.
Command Level Semaphore Management

There are two shell level commands which help manage semaphores:

- **ipcs** - reports status of interprocess communication facilities.

  turing(5)% ipcs

  IPC status from /dev/kmem as of Wed Jan 19 14:53:51 1994
  Message Queues:
  T   ID   KEY      MODE     OWNER     GROUP
  *** No message queues are currently defined ***

  Shared Memory
  T   ID   KEY      MODE     OWNER     GROUP
  *** No shared memory segments are currently defined ***

  Semaphores
  T   ID   KEY      MODE     OWNER     GROUP
  s   0   0 --ra----- barnett  other

- **ipcrm** - remove a message queue, semaphore set, or shared memory id

  turing(10)% ipcrm -s 0
The Readers/Writers Problem

Statement

Several processes want to access a shared database. Accesses can be either read requests or write requests. Any number of readers may simultaneously access the database, but only one writer at a time is allowed. Further, while a writer has access, no readers can be accessing the database.

Solution outline

We must keep track of the number of readers in order to be able to decide when a writer can be allowed access to the database.

Each reader increments a shared variable, rc, when it wishes to read. If the resulting count is 1 (i.e. this is the first reader), do a down on the semaphore db (this causes blocking if a writer is active.)

When done, the reader decrements rc. If it was the last reader, it does an up on db.

The writer does a down on db to request access. If blocked, the last exiting reader will wake it up. When done, the writer does an up on db, granting access to readers.
#ifndef _semaphore_h
#define _semaphore_h

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

/* This is defined in sem.h for SunOS, but not for Solaris or Ultrix */
#if defined(ultrix) || defined(SOLARIS)
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};
#endif /* ultrix || solaris */

#define SEM_KEY IPC_PRIVATE

class semaphore {
    public:

    // Constructor: Create a semaphore group with sem_count semaphores.
    semaphore (int sem_count);

    // Remove a previously defined semaphore from the system tables.
    ~semaphore () ;

    // Set the initial value of a semaphore.
    void init (int sem_idx, int value);

    // Return the current value of the semaphore sem_idx.
    int query (int sem_idx);

    // Decrease the value of a semaphore by 1.  Sleep if semaphore value
    // is initially zero.
    void down (int sem_idx);

    // Increase the value of a semaphore by 1.  If any processes
    // were blocked attempting a down, one of them is allowed to
    // complete its down.
    void up (int sem_idx);

    private:
        const unsigned int mySemCount;
        int mySemID;
    }
#endif /* !_semaphore_h */
#include <stdio.h>
#include <stdlib.h> // exit prototype
#include <iostream.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h> // fork prototype
#include <sys/ipc.h>
#include <sys/shm.h> // shmget, etc.
#include <sys/wait.h> // wait() prototype
#include <assert.h>
#include "semaphore.h"

#define TRUE 1
#define MAX_READERS 4 /* Number of readers */
#define MAX_WRITERS 2 /* Number of writers */
#define READ_SLEEP_MAX 5 /* Maximum sleep time for reader */
#define WRITE_SLEEP_MAX 10 /* Maximum sleep time for writer */

/* Indices for the semaphores in the returned semaphore group */
#define MUTEX 0 /* Mutual exclusion for access to RC */
#define DB 1 /* Mutual exclusion for access to DB */
#define RC 2 /* Counter for number of reader -- not really a semaphore, but needs to be shared. */
#define SEMMAX 3 /* Number of semaphores we’ll use here */

// Global semaphore, shared by all processes.
semaphore s(SEMMAX);

// Function prototypes
void reader (int);
void writer (int);
void clean_up (int);
#ifdef DEBUG
void dump_sem_vals (int);
#endif /* DEBUG */

/* The main function creates and initialized the semaphores, then spawns * the reader and writer processes. */

main ()
{
    int pid;
    int i;

    /*
     * Initialize the semaphores to the proper values */
/* 1 == "permission" to modify RC */
s.init(MUTEX, 1);
/* 1 == "permission" to access the database */
s.init(DB, 1);
/* No readers are initially active */
s.init(RC, 0);

/* Create the readers */
for (i = 0; i < MAX_READERS; i++) {
    if ((pid = fork()) == -1) {
        perror("forking reader");
        exit(1);
    } else if (pid == 0) {
        /* Executing in the reader child process now */
        reader (i);
        exit(0);
    }
}

/* Create the writers */
for (i = 0; i < MAX_WRITERS; i++) {
    if ((pid = fork()) == -1) {
        perror("forking writer");
        exit(1);
    } else if (pid == 0) {
        /* Executing in the writer child process now */
        writer (i);
        exit(0);
    }
}

/* Catch interrupt signals and clean up semaphores before terminating. *
 * Because the readers and writers run forever, we must use Ctrl-C to *
 * stop the program. We want to avoid leaving "orphaned" semaphores *
 * lying around, so we arrange for a function to be called when the *
 * program is interrupted by Ctrl-C. */
signal(SIGINT, clean_up);

/* Wait for all child processes to terminate. It is unlikely that *
 * the program will ever terminate this way, but we make provisions *
 * for it just in case. */
for (i = 0; i < MAX_READERS + MAX_WRITERS; i++)
    wait(0);
}

void reader (int index)
{
    int reader_count;

    /* Generate a different sequence of waiting times for each run. To *
    * generate the same sequence every time for testing purposes, *
    * comment this statement out. */
    srand(getpid());
while (TRUE) {
    /* Enter critical section */
    #ifdef DEBUG
    fprintf (stderr, "Reader %d wakes up.\n", index);
    /* dump_sem_vals (semaphore_id); */
    #endif /* DEBUG */

    /* Increase the number of current readers by 1 */
    s.down (MUTEX);
    s.up (RC);

    /* Find out how many current readers there are */
    reader_count = s.query (RC);

    /* If we are the first reader, we must wait until any current 
       writer finishes */
    if (reader_count == 1)
        s.down (DB); /* Sleep until database is available. 
           Note that no other readers will 
           be allowed to access RC until we 
           finish with it. */

    /* Release RC for other readers */
    s.up (MUTEX);

    printf ("Reader %d started reading. %d current readers.\n", 
            index, reader_count);
    sleep (random() % READ_SLEEP_MAX); /* Pretend to read... */

    /* We're done -- decrease the number of readers by 1. */
    s.down (MUTEX);
    s.down (RC);
    reader_count = s.query (RC);
    printf ("Reader %d finished reading. %d current readers.\n", 
            index, reader_count);

    /* If we are the last reader, make database available to writers */
    if (reader_count == 0) {
        s.up (DB);
    }
    s.up (MUTEX);

    /* Delay next read request */
    sleep (random() % READ_SLEEP_MAX);
}

void writer (int index) {
    int reader_count;

    while (TRUE) {
        sleep (random() % WRITE_SLEEP_MAX); /* Think up update */
        #ifdef DEBUG
        fprintf (stderr, "t\t\tWriter %d woke up.\n", index);
        #endif /* DEBUG */

        /* Enter critical section */
        #ifdef DEBUG
        fprintf (stderr, "Writer %d wakes up.\n", index);
        /* dump_sem_vals (semaphore_id); */
        #endif /* DEBUG */

        /* Increase the number of current writers by 1 */
        w.down (MUTEX);
        w.up (WR);
/* Wait for access to the database */
s.down(DB);

reader_count = s.query(RC);
printf("\t\t\tWriter %d started writing. %d current readers.\n", index, reader_count);
if (reader_count != 0) /* Sanity check */
    fprintf(stderr, "Nonzero reader count in writer critical section!\n");
sleep(random() % WRITE_SLEEP_MAX); /* Write update */

reader_count = s.query(RC);
if (reader_count != 1) /* Sanity check */
    fprintf(stderr, "More than one reader waiting after write!\n");
printf("\t\t\tWriter %d finished writing. %d current readers.\n", index, reader_count);
s.up(DB);
}

/* This function will be called after a Ctrl-C has been typed to terminate *
the program and before the program exits to tidy up before we quit. */
void clean_up (int sig)
{
    /* Clean up the semaphore before exiting */
    exit (1);
}
#endif /* DEBUG */

void dump_sem_vals (int sem_id)
{
    charoutstr[80];
    sprintf(outstr, "MUTEX = %d\nDB = %d\nRC = %d",
        s.query(MUTEX),
        s.query(DB),
        s.query(RC));
    fprintf(stderr, outstr);
}
#endif /* DEBUG */
star[lbarnett:faculty] (201) read_write

Reader 0 wakes up.
Reader 0 started reading. 1 current readers.
Reader 1 wakes up.
Reader 1 started reading. 2 current readers.
Reader 2 wakes up.
Reader 2 started reading. 3 current readers.
Reader 3 wakes up.
Reader 3 started reading. 4 current readers.
Reader 1 finished reading. 3 current readers.
Reader 2 finished reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading. 3 current readers.
Reader 3 finished reading. 1 current readers.
Writer 0 woke up.
Writer 1 woke up.
Reader 0 finished reading. 0 current readers.
Writer 0 started writing. 0 current readers.
Reader 2 wakes up.
Reader 1 wakes up.
Reader 3 wakes up.
Reader 0 wakes up.
Writer 0 finished writing. 1 current readers.
Writer 1 started writing. 1 current readers.
Nonzero reader count in writer critical section!
Writer 1 finished writing. 1 current readers.
Reader 2 started reading. 1 current readers.
Reader 1 started reading. 2 current readers.
Reader 3 started reading. 3 current readers.
Reader 3 finished reading. 2 current readers.
Reader 0 started reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading. 3 current readers.
Reader 1 finished reading. 2 current readers.
Reader 2 finished reading. 1 current readers.
Reader 3 wakes up.
Reader 3 started reading. 2 current readers.
Reader 0 finished reading. 1 current readers.
Reader 2 wakes up.
Reader 2 started reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading. 3 current readers.
Reader 1 finished reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading. 3 current readers.
Reader 1 finished reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading. 3 current readers.
Reader 1 finished reading. 2 current readers.
Reader 0 wakes up.
Reader 0 started reading. 3 current readers.
Reader 0 finished reading. 2 current readers.
Reader 1 wakes up.
Reader 1 started reading.  3 current readers.
Reader 1 finished reading.  2 current readers.
Reader 2 finished reading.  1 current readers.
Reader 3 wakes up.
Reader 3 started reading.  2 current readers.
Reader 0 finished reading.  1 current readers.
Reader 1 wakes up.
Reader 1 started reading.  2 current readers.
Reader 1 finished reading.  1 current readers.
Reader 0 wakes up.
Reader 0 started reading.  2 current readers.
Reader 3 finished reading.  1 current readers.
Reader 2 wakes up.
Reader 2 started reading.  2 current readers.
Reader 0 finished reading.  1 current readers.
Reader 1 wakes up.
Reader 1 started reading.  2 current readers.
Reader 3 wakes up.
Reader 3 started reading.  3 current readers.
Reader 1 finished reading.  2 current readers.
Reader 2 finished reading.  1 current readers.
Reader 2 wakes up.
Reader 2 started reading.  2 current readers.
Reader 2 finished reading.  1 current readers.
Reader 2 wakes up.
Reader 2 started reading.  2 current readers.
Reader 0 finished reading.  1 current readers.
Reader 3 wakes up.
Reader 3 started reading.  3 current readers.
Reader 1 finished reading.  4 current readers.
Reader 3 finished reading.  3 current readers.
Reader 0 finished reading.  2 current readers.
Reader 3 wakes up.
Reader 3 started reading.  3 current readers.
Reader 1 finished reading.  2 current readers.
Reader 2 finished reading.  1 current readers.

^C
star[1barnett:faculty](202)% exit
Accessing process information

SVR4-derived Unix systems allow programs to access status information of other processes via the proc file system.

The directory /proc contains a subdirectory for every process currently active.

The name of the folder is the process ID number.

Access to the information is via the open and ioctl system calls.

The open system call is used to get a file descriptor for the process subdirectory you are interested in.
The ioctl system call is used to request one of the following structures.

The following definitions come from the file /usr/include/sys/procfs.h
/*
 * process status file. /proc/<pid>/status
 */
typedef struct pstatus {
    int pr_flags;/* flags (see below) */
    int pr_nlwp;/* number of lwps in the process */
    pid_t pr_pid; /* process id */
    pid_t pr_ppid; /* parent process id */
    pid_t pr_pgid; /* process group id */
    pid_t pr_sid; /* session id */
    id_t pr_aslwpid; /* lwp id of the aslwp, if any */
    id_t pr_agentid; /* lwp id of the /proc agent lwp, if any */
    sigset_t pr_sigpend; /* set of process pending signals */
    uintptr_t pr_brkbase; /* address of the process heap */
    size_t pr_brksize; /* size of the process heap, in bytes */
    uintptr_t pr_stkbase; /* address of the process stack */
    size_t pr_stksize; /* size of the process stack, in bytes */
    timestruc_t pr_utime; /* process user cpu time */
    timestruc_t pr_stime; /* process system cpu time */
    timestruc_t pr_cutime; /* sum of children’s user times */
    timestruc_t pr_cstime; /* sum of children’s system times */
    sigset_t pr_sigtrace; /* set of traced signals */
    fltset_t pr_flttrace; /* set of traced faults */
    sysset_t pr_sysentry; /* set of system calls traced on entry */
    sysset_t pr_sysexit; /* set of system calls traced on exit */
    char pr_dmodel; /* data model of the process (see below) */
    char pr_pad[3];
    int pr_filler[19]; /* reserved for future use */
    lwpstatus_t pr_lwp; /* status of the representative lwp */
} pstatus_t;
/ * process ps(1) information file.  /proc/<pid>/psinfo */
#define PRARGSZ 80 /* number of chars of arguments */
typedef struct psinfo {
    int   pr_flag;/* process flags */
    int   pr_nlwp;/* number of lwps in process */
    pid_t pr_pid; /* unique process id */
    pid_t pr_ppid;/* process id of parent */
    pid_t pr_pgid;/* pid of process group leader */
    pid_t pr_sid; /* session id */
    uid_t pr_uid; /* real user id */
    uid_t pr_euid;/* effective user id */
    gid_t pr_gid; /* real group id */
    gid_t pr_egid;/* effective group id */
    uintptr_t pr_addr;/* address of process */
    size_t pr_size;/* size of process image in Kbytes */
    size_t pr_rssize;/* resident set size in Kbytes */
    dev_t pr_ttydev;/* controlling tty device (or PRNODEV) */
    ushort_t pr_pctcpu;/* % of recent cpu time used by all lwps */
    ushort_t pr_pctmem;/* % of system memory used by process */
    timestruc_t pr_start; /* process start time, from the epoch */
    timestruc_t pr_time; /* usr+sys cpu time for this process */
    timestruc_t pr_ctime; /* usr+sys cpu time for reaped children */
    char   pr_fname[PRFNSZ]; /* name of execed file */
    char   pr_psargs[PRARGSZ]; /* initial characters of arg list */
    int    pr_wstat;/* if zombie, the wait() status */
    int    pr_argc;/* initial argument count */
    uintptr_t pr_argv;/* address of initial argument vector */
    uintptr_t pr_envp;/* address of initial environment vector */
    char   pr_dmodel;/* data model of the process */
    char   pr_pad2[3];
    int    pr_filler[7];/* reserved for future use */
    lwpsinfo_t pr_lwp;/* information for representative lwp */
} psinfo_t;
/ * Resource usage.  /proc/<pid>/usage /proc/<pid>/lwp/<lwpid>/lwpusage */ typedef struct prusage {
    id_t    pr_lwpid; /* lwp id.  0: process or defunct */
    int     pr_count; /* number of contributing lwps */
    timestruc_t pr_tstamp; /* current time stamp */
    timestruc_t pr_create; /* process/lwp creation time stamp */
    timestruc_t pr_term; /* process/lwp termination time stamp */
    timestruc_t pr_rtime; /* total lwp real (elapsed) time */
    timestruc_t pr_utime; /* user level cpu time */
    timestruc_t pr_stime; /* system call cpu time */
    timestruc_t pr_rtime; /* other system trap cpu time */
    timestruc_t pr_tftime; /* text page fault sleep time */
    timestruc_t pr_dftime; /* data page fault sleep time */
    timestruc_t pr_kftime; /* kernel page fault sleep time */
    timestruc_t pr_ltime; /* user lock wait sleep time */
    timestruc_t pr_slptime; /* all other sleep time */
    timestruc_t pr_wtime; /* wait-cpu (latency) time */
    timestruc_t pr_stoptime; /* stopped time */
    timestruc_t pr_filltime[6]; /* filler for future expansion */
    ulong_t  pr_minf; /* minor page faults */
    ulong_t  pr_majf; /* major page faults */
    ulong_t  pr_nswap; /* swaps */
    ulong_t  pr_inblk; /* input blocks */
    ulong_t  pr_oublk; /* output blocks */
    ulong_t  pr_msnd; /* messages sent */
    ulong_t  pr_mrcv; /* messages received */
    ulong_t  pr_sigs; /* signals received */
    ulong_t  pr_vctx; /* voluntary context switches */
    ulong_t  pr_ictx; /* involuntary context switches */
    ulong_t  pr_sysc; /* system calls */
    ulong_t  pr_ioc; /* chars read and written */
    ulong_t  pr_ioch; /* filler[10]; /* filler for future expansion */
} prusage_t;
// File: track_heap.C
// Author: Lewis Barnett
// Description: Track changes in the size of the heap for a specified process.
// Time-stamp: <98/01/18 09:08:17 barnett>

#include <iostream.h>
#include <stdlib.h> // For exit(2)
#include <unistd.h> // For sleep(3C)
#include <sys/types.h> // for ioctl(2)
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/syscall1.h>
#include <sys/procsfs.h> // For procfs utilities
#include <sys/stat.h> // For open(2)
#include <fcntl.h>
#include <string.h> // For strncpy(3)
#include <stdio.h> // For perror(3C)
#include <errno.h>

#ifndef TRUE
#define TRUE
#endif // !TRUE

// Function prototypes
void track_proc(char *proc_id);

main(int argc, char *argv[])
{
    char proc_id[10];

    // Check for sufficient command line arguments.
    if (argc < 2) {
        cout << "Usage:  " << *argv << " proc_id" << endl << endl;
        exit(0);
    }

    // Process files in /proc have names with five characters,
    // corresponding to the process ID padded with leading zeros.
    // We do not require the argument to have the leading zeros,
    // so we must adjust if they are needed.
    int pid_len = strlen(argv[1]);
    if (pid_len < 5) {
        int i;
        for (i = 0; i < 5 - pid_len; i++)
            proc_id[i] = '0';
        int j = 0;
        for (; j < 5; j++) {
            proc_id[j] = argv[1][j];
            j++;
        }
    }
    proc_id[5] = '\0';
} else {
    strncpy (proc_id, argv[1], 10);
}

track_proc (proc_id);
return 0;

#define HEADING_MOD 24   // Print out a heading after this many lines
#define CHECK_INTERVAL 5  // Seconds between size checks

void track_proc (char *proc_id)
{
    char proc_path[80];
    int interval = 0;
    int proc_fd;
    prstatus p;
    u_long old_heap_size, new_heap_size;

    cout << "Tracking heap size of process " << proc_id << endl << endl;
    strncpy (proc_path, "/proc/", 80);
    strcat (proc_path, proc_id, 80);

    if ((proc_fd = open (proc_path, O_RDONLY)) == -1) {
        perror("track_proc - opening proc file");
        exit(2);
    }

    if (ioctl(proc_fd, PIOCSTATUS, &p) == -1) {
        perror("track_proc - ioctl request");
        exit(3);
    }

    cout << "\tPROC\tHeap\t\tUser time" << endl;
    cout << "\t" << proc_id << "\t" << p.pr_brksize << "\t" << p.pr_utime.tv_sec + p.pr_utime.tv_nsec/1000000000.0 << endl;
    interval++;
    old_heap_size = p.pr_brksize;
    new_heap_size = old_heap_size;

    while (TRUE) {
        while (old_heap_size == new_heap_size) {
            sleep (CHECK_INTERVAL);
            if (ioctl(proc_fd, PIOCSTATUS, &p) == -1) {
                perror("track_proc - ioctl request");
                exit(3);
            }

            new_heap_size = p.pr_brksize;
        }
    }
}
if (interval % HEADING_MOD == 0) {
    cout << "\tPROC\tHeap\t	User time" << endl;
}

cout << "\t" << proc_id << "\t" << p.pr_brksize
    << "\t\t" << p.pr_utime.tv_sec + p.pr_utime.tv_nsec/1000000000.0
    << endl;
old_heap_size = new_heap_size;
interval++;
Deadlock

A set of processes is said to be deadlocked if there exists a ring of dependencies among the processes where each process is waiting for a resource held by the next process in the ring.

Resource types

- Peripherals (printers, tape drives, etc.)
- Files
- Shared memory
- Semaphores
- Messages

Resources can be categorized as either preemptable or non-preemptable. Strategy for resolving deadlock depends on the type of resources involved.
## File Example

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open file foo, read/write</td>
<td>Open file bar, read/write</td>
</tr>
<tr>
<td>Attempt to open file bar, read/write, block</td>
<td>Attempt to open file foo, read/write, block</td>
</tr>
</tbody>
</table>

## Message Example

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send request to p2</td>
<td>Receive request from p1</td>
</tr>
<tr>
<td>Receive ack from p2</td>
<td>Send acknowledgment to p1</td>
</tr>
<tr>
<td>Receive from p2, block</td>
<td>Receive from p1, block</td>
</tr>
</tbody>
</table>
Conditions for Deadlock to Exist

- Mutual exclusion condition—each resource is either allocated to a single process or is available for allocation. Resources which allow concurrent access can’t cause a deadlock.

- Hold and wait condition—processes can request more than one resource at a time. (This is necessary for some applications.)

- No preemption condition—if a process holds a resource, the resource cannot be taken away; the process must release it. Some resources are inherently non-preemptable.

- Circular wait condition—there must be a circular chain of two or more processes in which each process is waiting for a resource held by the next member of the chain.
Handling Deadlock

• Ignore it (the Unix approach)-deadlock is detected and resolved via external intervention. In some applications, this makes sense because deadlocks are relatively rare and most strategies that attempt to actively deal with them are either expensive or impose undesirable restrictions.

• Detect and recover-a deadlock detection algorithm is periodically executed and deadlocked processes are jump started somehow. For systems with a single resource class, this involves constructing a resource graph then using depth-first search on each node of the graph to detect cycles. Multiple resource classes require a more complex approach.

• Avoid-each resource request is examined and potentially denied if it leads to an “unsafe state.” Typically, this requires the same kind of advanced knowledge that Shortest Job First scheduling requires.

• Structure the environment so that deadlock is not possible.
Deadlock Detection, Multiple Resource Case

Assume that we have processes $P_1 \ldots P_n$ and resources classes numbered $1 \ldots m$. In addition, we maintain two vectors describing the state of system resources:

- $E_1 \ldots E_m$: Existing resources in each class
- $A_1 \ldots A_m$: Available resources in each class

There are also two matrices describing resource allocation requests:

- $C_{ij}$: The number of resources of class $j$ currently allocated to process $i$
- $R_{ij}$: The number of resources of class $j$ requested by process $i$.

At all times, the following equality holds:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j \text{ for all resources classes } 1 \leq j \leq m$$

Let $A \preceq B$ mean that $A_i \leq B_i$ for $1 \leq i \leq m$

Initially, all processes are “unmarked.”

1. Find an unmarked process $P_i$ for which $R_i < A$
2. Add $C_i$ to $A$ and mark this process.
3. Repeat until all processes are marked or there is no process which satisfies $R_i < A$

If any unmarked processes exist, there is a deadlock.
Deadlock Detection Example

Resource classes:

0 = Printers, 1 = Console, 2 = CD-ROM, 3 = Tape

\[ E = \begin{bmatrix} 2 & 1 & 1 & 1 \end{bmatrix} \] \quad \[ A = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \]

\[ C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \] \quad \[ R = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Process 0 can’t run because there is no CD-ROM (class 2)

Process 1 can’t run because the console (Class 1) is in use

Process 2 can run. Assume it does so and releases its resources.

\[ A = \begin{bmatrix} 2 & 0 & 0 & 1 \end{bmatrix} \]

\[ C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \] \quad \[ R = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \]

Process 0 can’t run because there is no CD-ROM (class 2)

Process 1 can’t run because the console (Class 1) is in use

These two processes are deadlocked.
Deadlock Recovery Techniques

Preemption

Take away a resource held by one process and give it to another. Since only non-preemptable resources can cause deadlock, this will likely result in the former process failing.

Checkpointing/Rollback

Periodically save the state of each process. If deadlock occurs, choose a process and roll it back to its most recent checkpoint, presumably releasing some resources to be given to other processes.

Process Termination

Kill a process. Its resources will be released and execution of other processes may continue.

Criteria for choosing the process:
- It is a member of the deadlocked process chain, or
- It holds resources that could affect the deadlock

Detection/Recovery drawbacks

- Requires some advanced knowledge of process resource requirements
- Number of resources classes is potentially large (e.g. record or byte level file locking)
- Expensive in terms of system resources
Deadlock Avoidance

Examine each resource request to determine if granting it would create the potential for deadlock.

A safe region of system execution is a state in which no processes are currently deadlocked and a sequence of process activations exists such that all processes can run to completion.

A system state which doesn’t satisfy this definition is guaranteed to eventually lead to a deadlock.

The Banker’s Algorithm

The Banker’s Algorithm is essentially the same as the deadlock detection algorithm discussed previously, but used in a different context.

The basic idea is to use the algorithm each time a resource is requested to decide whether granting the resource will lead to a deadlock.

If it would, the request is denied, otherwise, it is granted.

Requires advance knowledge of maximum resource requirements of each process
Deadlock Prevention

Structure access to the resources of the system in such a way that deadlock is impossible. To do so, we must eliminate the possibility of one of the four conditions for deadlock occurring.

Mutual Exclusion Condition

Example: printer spooling. Printer daemon is only process that actually has access to the printer. Need for mutually exclusive access is eliminated.

Hold and Wait Condition

- Request all resources up front
- Request of new resource means temporarily releasing all currently held resources and requesting everything you need as a group

Preemption

Some things are inherently non-preemptable.

Circular Wait

- Allow only one resource to be held at a time
- Number resources and only allow requests for resources in classes higher than the currently held resources. If A holds i and B holds j, then either
  • A cannot request j (because i > j), or
  • B cannot request i (because j > i)

so no deadlock can occur.
Memory Management

Given that multiple processes must coexist in memory, what issues must we face in deciding how to manage this coexistence?

Why Multi-program?

- When appropriate, applications can be split into communicating processes
- Multiple interactive users can be accommodated simultaneously
- Processor utilization improves when processes blocked for I/O can relinquish the processor to other active processes.

Fundamental issues

- How can we construct executables when we don’t know where they will be located in memory before run time?
- How can we prevent one process from corrupting the memory used by another process?

Strategies

- Keep symbol table info available in executable so that addresses can be modified when program is loaded. If process is swapped out, addresses will need to be adjusted again. Extra work required for protection.
- Use base and limit registers to adjust addresses and prevent references outside a process’s memory partition. Processes are dynamically relocatable.
Source to Executable stages

Source Code

Compiler

Object module

link editor

Loadable module

Loader

Memory-resident binary image

Compile time

Load Time

Run Time

other object modules

System libraries

Dynamically loaded System libraries

other object modules

System libraries

Dynamically loaded System libraries
Program Address Binding

Addresses (e.g. variable references) in programs must be bound to an address in the physical memory of the computer in order for the program to execute. Binding can occur at different times:

- compile time - results in *absolute code*, which must be loaded at a specific location in memory.
- load time - when the program image is loaded into memory. Program image must be generated as *relocatable code*, which can be adjusted automatically to reflect the location where it is loaded.
- execution time - if process can be moved during execution (e.g. swapped out), binding of references to addresses must be delayed until runtime. Requires special hardware support.

Other factors can affect mapping:

- Dynamic loading - technique of only loading portions of code if and when they are actually used.
  - Results in a reduced memory footprint for the executable.
- Dynamic linking - typically used for libraries; allows system libraries to be linked to the executable at run-time rather than link time.
  - Results in reduced disk and memory footprints.
Logical vs. Physical Address Spaces

*logical addresses* - addresses generated by the CPU to reference memory. (aka *virtual addresses*)

*physical addresses* - addresses generated from logical addresses by the memory management hardware to actually locate data in main memory.

Compile time and load time binding of references to addresses result in code where the logical and physical addresses are the same.

Execution time address binding results in logical and physical addresses that may be different.

*logical address space* - the range of logical addresses generated by the CPU during the execution of a program.

*physical address space* - the range of physical addresses generated by the Memory Management Unit (MMU) during the execution of a program.
Memory Allocation Strategies

Fixed Partitions (OS/360, Batch processing)

- If a queue is maintained for each partition by process size, partition utilization may be sub-optimal
- Small jobs can be discriminated against
- Jobs seldom fit partitions exactly, so memory is wasted

Variable Partitions

Each process is allocated the memory it needs when it is first swapped in.

- How much memory is enough?
- What happens when processes terminate?
- What happens when processes grow?
Memory Management Techniques

All techniques assume memory is allocated in multiples of some fixed Memory Allocation Unit.

**Bitmap**

Each chunk of memory is represented by a bit in a bit string. If the chunk is in use, the corresponding bit = 1, 0 otherwise. Allocation is time-consuming.

**Linked List**

Maintain a list (or lists) of processes and holes. Sorting by address makes coalescing adjacent holes easier. Sorting by size makes allocation easier.

Allocation Strategies

- First fit - allocate the first block large enough to accommodate a new process.
- Best fit - search for hole with size closest to new process.
- Worst fit - choose the largest hole.
Virtual Memory

It is possible for the size of a process to exceed the size of physical memory. In this case, we must use disk space to simulate a larger virtual memory for the process to use.

The process image can be divided into pages, and only the pages that are currently needed are loaded into physical memory.

A hardware device, the Memory Management Unit (MMU), handles translating virtual addresses into physical addresses.

Physical memory is divided into page frames, each of which can hold a page.

If reference is made to a virtual address which is not currently mapped to a page frame, a page fault occurs, requiring a page to be evicted from memory and the referenced page to be read in.

The mapping between pages and page frames for a process is recorded in the page table for the process.

Facts of paging life:

- Page tables can be BIG!
- The mapping must be FAST!
Page Table Contents

- Page frame #
- Present/Absent bit
- Modified bit
- Referenced bit
- Protection bits
- Caching Disabled
- Protection bits

Multilevel paging

- PT1
- PT2
- Offset

To pages...
Paging - Hardware Support Examples

DEC PDP-11

4 Megabyte maximum physical memory
16 bit virtual addresses -> 64K virtual address space
8 slot hardware page table
8K page size (8 slots x 8K pages = 64K)
$4M / 8K = 2^9$ page frames, so physical page frame number is 9 bits
Context switching is very fast
There are no page faults

MIPS R2000 (DECstation chip)

Large associative memory is only hardware support - info is augmented with process id for each page.
No page tables
TLB miss causes OS trap - OS decides which TLB entry to invalidate and replace, handles pages faults
CPU design is simplified, area saved use for on-board MMU and cache controller.
Motorola 68030

Number of levels of page tables is configured by OS, up to 4
Number of entries in page table at each level is configured by OS
Essentially, provides hardware support for any kind of paging scheme the OS writers want to implement.

DEC Alpha AXP

3 level page tables
Allows a prefix (up to 21 bits) of virtual addresses to be ignored
Page size is selectable from 8K to 64K
TLB “granularity hints”

Each TLB entry can map a range of contiguous page frames which have identical protection attributes. When used properly, results in a very low TLB miss rate.

For further info, see Feb. 1993 CACM
Inverted Page Tables

Virtual address space >> physical memory causes problems for traditional page table organization:

64 bit address space can address 18,446,744,070,000,000,000 locations.

4K page size would require 4.5 quadrillion page table entries per process. (We don’t even have disks that big today...)

Physical memory peaks at about 1000M, so seems more feasible to arrange page tables around page frames than virtual pages.

<table>
<thead>
<tr>
<th>Page Frame</th>
<th>Process ID</th>
<th>Flags</th>
<th>Virtual Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Used with TLB, misses require a search of page table for required virtual address - hashing could speed up.

Conventional page table still needed for swapping.
Page Replacement Algorithms

Most systems have Modified (M) and Referenced (R) bits which can be used to decide what pages to swap out when space is needed.

Not Recently Used

Divides processes into classes based on values of R & M bits:

Class 0: R = 0 \hspace{1cm} M = 0
Class 1: R = 0 \hspace{1cm} M = 1
Class 2: R = 1 \hspace{1cm} M = 0
Class 3: R = 1 \hspace{1cm} M = 1

Randomly choose a page from the lowest numbered non-empty class and evict it.

FIFO

Page that has been in memory longest is evicted

Second Chance

FIFO except that old pages with R bit set are recycled.

Clock

FIFO except queue is circular with pointer to oldest page.
Least Recently Used

Keep a timestamp for each page with time it was last referenced. Evict the page with the lowest timestamp.

Obvious implementation requires a linked list of all pages ordered by timestamp. Must be updated on every memory reference.

Hardware Implementation Strategies

- Each page table entry has timestamp, updated each time the page is referenced. On page fault, do linear search of page table entries to find lowest timestamp.
- Keep an \( n \times n \) matrix of bits. On reference to page \( k \)
  - Set all bits in row \( k \) to 1
  - Set all bits in column \( k \) to 0

Effect: gives most recently referenced page the highest value and lowers the value of all other pages. Lowest row of matrix corresponds to least recently used page.

Simulation in software

Each page has a software counter. At each tick, the bits in the counter are shifted right one place and the R bit value for the page is appended to the left of the counter. This provides “aging” of the reference counts.
Implementation Issues

The Working Set Model

Each process is assumed to go through phases

At each phase, the majority of the processes references are to a well defined subset of the total pages for the process. This subset is called the working set.

It is good strategy to allocate at least enough pages for a process to hold its current working set, otherwise, the process will generate frequent page faults (thrashing).

Prepaging is the technique of loading all of the pages of a processes working set into memory before allowing the process to run after being swapped out. This avoids a lot of context switch overhead.

Global vs. Local Allocation

When a process page faults, should all pages in memory be considered candidates for eviction (global allocation) or only pages belonging to the process which caused the fault (local allocation)?

Generally, global algorithms result in better performance, particularly if the working set size of processes vary over time.

Page Fault Frequency Algorithm -- Establish high and low thresholds for page fault frequency. If a process exceeds the high limit, it should be allocated more page frames, preferably from a process whose fault rate falls below the low threshold. Otherwise, a process should be swapped out and its space divided.
Page Size

Small Pages

Pro: Reduces *internal fragmentation*
   Less unneeded code in memory at any given time

Con: Large page tables

Large Pages

Pro: Fewer page faults (potentially)
   Doesn’t take much longer to load a large page (rotational latency dominates disk access time)
   Smaller page tables

Con: More “unused” code in memory

I/O interaction

Page containing the buffer for an I/O request should not be paged out.

- Page can be locked in memory (reduces paging flexibility)
- I/O can be done to kernel buffer, then transferred to user buffer when page is available (requires an extra copy)

Sharing

Main problem is tracking which processes are accessing a shared page to manage deallocation upon process termination.
Backin g Store

- Preallocation - swap space equal to core image size is allocated when process starts. Doesn’t take into account potential size changes.
- Preallocation with separate areas for text, data, and stack, each potentially consisting of several chunks of swap space. Increases complexity of management.
- Demand allocation - swap space is allocated on a page by page basis as pages are swapped out. Requires keeping track of a disk address per page.

Paging daemons

- Paging is easier if there is a pool of free pages waiting to satisfy page requests.
- Daemon runs periodically to see if free pool is too small and applies page replacement algorithm to free up pages if necessary.
- Dirty pages can be written out so that no disk write is necessary when the frame is reused.
- Freed pages can be tracked - if reference occurs before the page is reused, it can simply be reclaimed from the free pool.
Files

*file* - a group of related information with a set of attributes, stored on a secondary storage medium, which can be accessed by a name.

File Structures

- Byte stream- contents are an unstructured stream of bytes
- Record oriented- file is considered to be a set of records with some internal structure
- Tree structured- essentially a search tree, for rapid lookup on key values

File Types

- ASCII (text)- file is composed of readable characters separated by newlines (maybe)
- Binary - bytes can have any values. Ex: executables, data files

Access strategies

- Sequential - information in the file is processed in order, from beginning to end.
- Direct access (or random access or relative access) - mirrors access methods of the disk; any location within the file can be accessed.

*file system* - the part of the OS which manages the organization of, access to, and protection of files.
File System Characteristics

- Naming convention

- Directory structure

- Protection mechanism

- Attribute handling
File Storage Implementation

Contiguous Allocation

Each file is allocated a contiguous area in memory
- Easy to keep track of where file’s blocks are
- File can be read from disk in a single read operation
- Maximum file size must be known in advance
- Deletion creates fragmentation

Linked List Allocation

File is stored as a linked list of blocks
- No fragmentation
- Directory only stores address of first block
- Sequential access is easy
- Random access is more difficult
- Some space in every block is used to store pointer
Indexed Linked List

Rather than storing the links in the disk blocks, they are stored in an index table for the disk.

- File’s directory entry stores only the index for the first block.
- Entire disk block is available for data
- Random access is easier than w/ straight linked list
- Whole table must be in memory

Index Nodes

Keep information about a file’s attributes and block addresses in a file (index node)

- First few block addresses stored in i-node itself
- Subsequent pointers are to “indirect blocks” which hold addresses of disk blocks or further indirect blocks
Directory Structure

- Disk is divided into multiple *partitions* (or *minidisks* or *volumes*) in which directories and files reside
- Partitions may be smaller than the disk (multiple logical disks per physical disk) or larger than the disk
- Typically, there is a subtree of the full directory tree stored on a given partition.
MS-DOS Directory Implementation

A directory entry has the following fields:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>File name</td>
</tr>
<tr>
<td>3</td>
<td>File name extension</td>
</tr>
<tr>
<td>1</td>
<td>Attributes</td>
</tr>
<tr>
<td>10</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>2</td>
<td>Time</td>
</tr>
<tr>
<td>2</td>
<td>Date</td>
</tr>
<tr>
<td>2</td>
<td>First block number</td>
</tr>
<tr>
<td>4</td>
<td>Size</td>
</tr>
</tbody>
</table>

The first block number is an index into a table of block addresses. Allows hierarchical directory structure.

UNIX Directory Implementation

File system is organized around i-nodes, so directory entries are very simple, containing only a filename and an i-node number. Attributes, etc. are stored in the i-node. Hierarchical structure allowed.
Hierarchical Directories

Strict Tree Structure

Graph Structure
File Sharing Mechanisms

Hard Links

• Separate directory entries indicate the same file.
• Easier if directory entries do not contain addresses of disk blocks.
• Creating a link increments a link counter in the file’s i-node
• Deleting the file removes the specified directory entry and decrements the i-node’s link count by one.
• If the resulting link count is 0, the files blocks are placed on the free list and the i-node is deleted.

Soft (Symbolic) Links

• OS creates a special file which contains only the path name of the file that the link refers to.
• Removal of the original file leaves a “dangling link”
• Adds to overhead of file access.
• Extremely flexible.

Example: World Wide Web Universal Resource Locator (URL)

Typical URL format:

```
file://wuarchive.wustl.edu/mirrors/msdos/graphics/gifkit.zip
```

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Hostname</th>
<th>Pathname</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>wuarchive.wustl.edu</td>
<td>mirrors/msdos/graphics/gifkit.zip</td>
</tr>
</tbody>
</table>
Disk Space Management

Block Size

• Large blocks waste space due to internal fragmentation, but access is more efficient
• Small block size uses space more efficiently, but access is slowed down.
• Memory page size and disk block size should be compatible
• Typical block sizes: 512B, 1KB, 2KB. Should be a multiple of the disk’s sector size.

Free list management

• Linked list - each block in the list contains as many free block numbers as will fit. If all blocks referenced by a free list block are used, the free list block can be “freed.”

Example:
500M disk with 1K blocks => 512,000 blocks
Requires 19 bit block numbers, each needing 3 bytes
341 block numbers fit in each free list block
1502 blocks (approx. 1.5M) needed to hold free list.

• Bitmap - each block has a bit in the list (as opposed to enough bytes to hold a disk block index for linked list)

More efficient, except when disk is nearly full. (Example above requires 64KB.)
Disk Quotas

- Table of all users and quotas stored on disk
- Table of all users with currently open files kept in memory
- Typically, hard and soft limits are assigned, hard > soft
- Soft limit may be exceeded during a session, but total usage must be below it before user logs out
- Hard limit may never be exceeded.
- Every change to an open file requires that the quota information in the memory-resident table be updated and appropriate warnings generated if an attempt is made to exceed a limit.
Unix directory processing

Unix provides a set of system calls for reading and interpreting directories.

These system calls give programs the ability to read a stream of dirent (directory entry) structs. (Man page: dirent(4). Header file: dirent.h.)

```c
struct dirent {
    ino_t d_ino;
    off_t d_off;
    unsigned short d_reclen;
    char d_name[1];
};
```

- `d_ino`: Unique i-node number for the file
- `d_off`: File-system dependent offset to next directory entry.
- `d_reclen`: The length in bytes of this record.
- `d_name`: The name of the file. (Because this may vary in size, we need `d_reclen` to tell us how big this dirent is.)

The maximum allowed length of a filename is given by the `dirent.h` constant `MAXNAMLEN`. 
Directory system calls

To open a directory for reading

```c
DIR * opendir(const char *dirname);
```

opendir takes the name of the directory to open as a parameter and returns a pointer to a DIR structure (defined in dirent.h) which acts as a “directory descriptor” (analogous to a file descriptor) for subsequent operations on the open directory. (Man page: opendir(3C). To use, include header files sys/types.h and dirent.h.)

To read the next directory entry

```c
struct dirent * readdir(DIR * dirp);
```

readdir takes the DIR pointer returned by opendir and returns the next dirent struct in the “directory stream.”

readdir returns null when all directory entries have been read. (Man page: readdir(3C).)

To close a directory when processing is complete

```c
int closedir(DIR *dirp);
```

Man page: closedir(3C)
Retrieving file information

The dirent structs returned by readdir give us only the name of the next file in the directory. To get further information, we must use the `lstat` system call.

`lstat` gives us access to the information stored in the file’s i-node in the form of a `stat` struct. (Header file: `stat.h`)

```c
int lstat(const char * path, struct stat * buf);
```

`path` is the complete pathname to the file whose information you want. `lstat` fills in the data members in the `stat` struct to which `buf` points.

0 is returned on successful completion, -1 otherwise.

Descriptions of structure members of interest are as follows:

- **st_mode**: The mode of the file as described in mknod(2). In addition to the modes described in mknod(), the mode of a file may also be `S_IFLNK` if the file is a symbolic link.

- **st_ino**: This field uniquely identifies the file in a given file system. The pair `st_ino` and `st_dev` uniquely identifies regular files.

- **st_dev**: This field uniquely identifies the file system that contains the file.

- **st_uid**: The user ID of the file's owner.

- **st_gid**: The group ID of the file's group.

- **st_size**: For regular files, this is the number of bytes in
the file.

st_atime      Time when file data was last accessed. (Times measured in seconds since 00:00:00 UTC, Jan. 1, 1970.)

st_mtime      Time when data was last modified.

st_ctime      Time when file status was last changed.

st_blkszsize  A hint as to the "best" unit size for I/O operations.

st_blocks     The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.
Using bitflags

System calls often use a sequence of bits called a bitflag to record information about the type or status of a device or resource in a compact way. The stat.h header file has the following defines that are used in the st_mode data member of the stat struct:

```c
#define S_IFIFO 0x1000 /*0b0001000000000000*/
#define S_IFCHR 0x2000 /*0b0010000000000000*/
#define S_IFDIR 0x4000 /*0b0100000000000000*/
#define S_IFBLK 0x6000 /*0b0110000000000000*/
#define S_IFREG 0x8000 /*0b1000000000000000*/
#define S_IFLNK 0xA000 /*0b1010000000000000*/
```

These “bitmasks” represent various kinds of files recognized by the Unix file system. The “0x” prefix indicates a number in hexadecimal format. I have used “0b” here to indicate a number in binary format.

We use the bitwise OR operator “|” and the bitwise AND operator “&” to manipulate such quantities.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int mode = S_IFIFO</td>
<td>S_IFCHR;</td>
</tr>
<tr>
<td>int mode2 = (mode &amp; S_IFCHR);</td>
<td>mode2 == 0x2000</td>
</tr>
</tbody>
</table>

- Bitwise OR combines bitflag values
- Bitwise AND “masks out” bits -- you get only bits that are on in both operands in the result.
Comparisons using bit masks

If all of the bit masks for a bit flag contain only a single bit that is “on,” then we can do comparisons in a straightforward way. The following if tests a stat instance called s to see if the file it refers to is a directory:

```cpp
if (s.st_mode & S_IFDIR) {
    cout << “This is a directory.” << endl;
}
```

This works because C/C++ equate false with the integer value 0 and true with any other value; if the & operation results in any bits being set in the resulting value, the condition is satisfied.

Note that this doesn’t really work here, though, because some of the other masks have bits that overlap with S_IFDIR.

If the set of masks we need to use contains masks with multiple bits, the simple comparison shown above is not sufficient. We must further compare the result to the bitmask to be sure that all of the bits it uses are on:

```cpp
if ((s.st_mode & S_IFLNK) == S_IFLNK) {
    cout << “This is a symbolic link.” << endl;
}
```
# File handling example

```c
#include <stdio.h> // for perror
#include <dirent.h> // for DIR, opendir, readdir, dirent, MAXNAMLEN
#include <sys/stat.h> // for struct stat, S_IFLNK, S_IFDIR, S_IFREG
#include <iostream> // for cout, cerr
#include <string>

struct stat *get_stat_entry(string, string);

main (int argc, char *argv[])
{
    DIR *dp; // Descriptor for a directory stream
    struct dirent *entry; // Pointer to a directory entry
    struct stat *buf; // Pointer to stat struct returned by stat()
    string dirname;
    string errstr;

    if (!(--argc)){
        dirname = "."; /* If no name supplied on command line,
        use the current directory */
    } else {
        argv++;
        dirname = *argv;
    }

    // Get a directory entry stream for the requested directory.
    if ((dp = opendir (dirname.c_str())) == NULL) {
        errstr = "scandir(): couldn’t open directory "+ dirname;
        // perror prints an "official" error message.
        perror (errstr.c_str());
        exit(1);
    }

    // Good directory name -- scan entries and process each one.
    for (entry = readdir (dp); entry != NULL; entry = readdir (dp)) {
        // Get the stat struct for this entry using the
        // name of the file.
        buf = get_stat_entry(dirname, entry->d_name);
    }

```
// If the current entry is a subdirectory... We want
// to skip the two special directories "." and "..".
// The strcmp function returns 0 for equality, which
// is interpreted as "false" by C++...
if ((buf->st_mode & S_IFDIR) == S_IFDIR &&
    (strcmp (entry->d_name, ".")) &&
    (strcmp (entry->d_name, ".."))) {
    string subdirname(entry->d_name);
    cout << subdirname << endl;
}

// The memory for buf is allocated in the
// get_stat_entry function.
// We return it to the system here.
free(buf);

closedir(dp);

// Get the status information on a file given the name of the
// directory it belongs to and its file name.
struct stat *get_stat_entry (string dir, string name)
{
    struct stat *buf; // We’ll dynamically allocate a stat
                      // structure.
    string        inname;
    string        basename;
    string        errstr;

    /* This space should be freed by whatever routine calls
       get_stat_entry */
    if ((buf = (struct stat *) malloc (sizeof (struct stat)))
        == NULL) {
        errstr = "get_stat_entry(): " + dir + "/" + name;
        perror (errstr.c_str());
        return NULL;
    }

    /* Construct a full pathname from the directory name and
       file name. */
    if (dir == "."){
        inname = name;
    } else {
        inname = dir + "/" + name;
    }

    return buf;
}
// lstat tells us about symlinks rather than the file they
// refer to.
if (lstat (inname.c_str(), buf) < 0) {
    errstr = "Couldn’t stat “ + inname;
    perror (errstr.c_str());
    return NULL;
}

return buf;
## Unix File Classification

Many files stored on a Unix file system begin with a “magic number.” A table of rules for detecting these numbers is stored in the file `/etc/magic`. Each entry has the following form:

<table>
<thead>
<tr>
<th>offset</th>
<th>type</th>
<th>value</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>Position, in bytes of the magic number to test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>Type of data to be tested. One of byte, short, long, or string. Byte, short, and long may be followed by ‘&amp;’ and a bit mask to be anded with the number before a comparison is done.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>Value to be compared to the value in the file. If numeric, specified in C form (\234\021 is a combination of two bytes, each interpreted as an octal number, for example). The usual C escapes are allowed in strings (e.g. \t for tab).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Numeric values may be preceded by a comparison operator to use. =, >, <, &, and ^ are allowed. The “operator” x may be used to indicate that any value should match. Default is =.

| message | A message to be printed out for this type of file. If the message contains a printf format specifier (e.g. %d) the string is used as a format string to print out the value tested-useful in conjunction with the x operator to print out information from the file. |

A ‘>’ in the first column means an additional test/message.
Example /etc/magic File

0 short 070707 cpio archive
0 short 0143561 byte-swapped cpio archive
0 string 070707 ASCII cpio archive
0 string \037\036 packed data
0 string \377\037 compacted data
0 string \037\235 compressed data
>2 byte&0x80>0 block compressed
>2 byte&0x1fx %d bits
0 string \032\001 Compiled Terminfo Entry
0 short 0433 Curses screen image
0 short 0434 Curses screen image
0 string <ar> System V Release 1 archive
0 string !<arch>\n__.SYMDEF archive random library
0 string !<arch> archive
0 long 0x1010101MMDF mailbox
0 string %! PostScript document
# version ID follows, in the form PS-Adobe-nn
0 string StartFontMetricsASCII font metrics
0 string StartFont ASCII font bits
0 long 0x137A2944NeWS bitmap font
0 long 0x137A2947NeWS font family
8 long 0x137A2B45X11/NeWS bitmap font
8 long 0x137A2B48X11/NeWS font family
0 long 0x137A2950scalable OpenFont binary
0 long 0x137A2951 encrypted scalable OpenFont binary
0 string <MakerFileFrame Maker document
0 string <MIFFileFrame Maker MIF file
0 string <MML Frame Maker MML file
0 short 017613 gzip compressed data

For further details, see man pages for file and magic.
Reliability

Bad Block management

All hard disks have some number of bad blocks.

- Hardware solution: a sector of the disk is dedicated to a list of bad blocks. This sector is not accessible except to the file system portion of the OS. Any disk reference is checked against the bad block list. If there is a “hit,” the reference is forwarded to a file system selected spare block.

- Software solution: at boot time, OS constructs a bad block table, then removes those blocks from the free list so that they are never referenced.

Backups

In case of disk failure or accidental deletion of files, backups on an independent medium must be maintained.

- Typical devices: 9 track (reel) tape, tape cartridge
- For small devices, periodic full backup is practical
- For large devices, infrequent full dump followed by incremental dumps is usual strategy.
  - Dump only files modified since last full dump
  - Dump only files modified since last incremental dump

Requires a table of dump times for all files. At least the last two full dumps should be retained.
Consistency Checking

Some file system problems can be detected before they cause serious problems. Typically, a file system consistency check is run every time a computer is booted.

The Unix `fsck` program does the following checks:

- **Block consistency checking** - makes sure that
  - every block is either free or part of a file (missing file detection)
  - every free block occurs in the free list only once
  - every allocated block is part of only one file
  - no block appears both in a file and in the free list

- **File consistency checking** - checks i-nodes against directories. Two problems can occur:
  - Directory reference count is lower than i-node count. Leads eventually to an i-node which no directory entry points to, which wastes disk space.
  - Directory Reference count is higher than i-node count. Can lead to references through freed i-nodes.

In both cases, the solution is to force the i-node count to agree with the count generated by the directory search.
Performance

Disk access is up to five orders of magnitude slower than memory access, so care is usually taken to minimize disk accesses.

Caching

- Keep a buffer cache in memory of recently referenced blocks.
- Hits are handled completely in memory with no disk access.
- Misses are handled like page faults - references are infrequent enough to actually use a modified LRU replacement policy.
- I-nodes or other file system related blocks should be written out as soon as modified to avoid file system corruption on crashes.
- Even if used often, blocks should be written out periodically to avoid lost work if crash occurs.
  - Update daemon which syncs disks periodically (Unix)
  - Write-through cache - all writes to blocks in cache cause block to be written out. (MS-DOS)
Hardware dependent tweaks

- Put blocks that are likely to be referenced in sequence near each other on disk to minimize arm motion. Easy for bitmap allocation management, harder for linked list.

- Same technique, but take advantage of rotation in positioning blocks that may be referenced sequentially. Rotational latency must be taken into account.

- Keep I-nodes closer to files they refer to.
Security

Typical Security Concerns

• Data Loss
  - Hardware failure
  - Human error

• Voyeurism-peeking at other people’s files

• Financial hacking
  - Actual theft
  - Logic bomb/blackmail

• Espionage
  - Industrial
  - Military

Typical OS Security Flaws

• “Deleted” information is only deallocated, not erased
• Inadequate parameter checking on system calls
• Interruptible login programs
• Modifying system generated user-space data structures
• Poorly designed utilities that grant temporary root access

“Human Engineering”

• Mimic programs
• Shmoozable/bribable systems admins
Security Holes

The Setuid Mechanism

Most Unix security problems involve poorly written programs that grant temporary root privileges.

uid user ID number

effective uid current user ID of a process

setuid bit bit in the protections that specifies that the program should run with the effective uid of the owner of the executable rather than the uid of the user running the program.

• The `mkdir` hole

  mkdir’s actions:
  - Create the inode for the new directory
  - chown inode to uid of user who ran `mkdir`

  The hole: if done quickly, the inode could be replaced with a link to a (normally restricted access file), which `mkdir` would then chown to be owned by the user running the `mkdir` command.

• The `emacs movemail` hole

  emacs mail allows you to forward a message to another user. To do so, it must setuid root to copy the file to the other user’s directory. It originally did not check to see whether the requested directory was `/`, allowing files with owner root to be created. (Wily Hacker)
The Internet Worm - exploited three holes

- The .rhosts "trusted host" mechanism - required breaking into an account first
- A bug in the `finger` daemon

The daemon expected requests to be under a certain length, but did not check the length of actual lengths of the requests. `rtm` sent a request longer than the limit which wrote bytes that were a custom function onto the daemon’s stack space. When the daemon tried to return from the current function, it executed `rtm`’s function instead. The function executed `/bin/sh`, creating an executing shell on the remote machine.

- The mail bug

The sendmail program allowed a program to be mailed to a unix system and executed. (This was only if the program had been installed with a debugging flag on.)

Once established, the worm tried to break passwords, search .rhosts files for other "trusted systems" and spread itself to those systems.
Virii

virus a program fragment that is attached to a legitimate program with the intention of infecting other programs and potentially performing other actions determined by the programmer.

Viruses are typically transmitted by infecting a useful program and then making it publicly available. When the infected program is run on a new system, it looks for other files to infect, which might then be passed on to other computers.

PCs are generally most vulnerable because they lack memory and file protection mechanisms of larger computers.

Effects

- *Passive infection*-the virus simply spreads itself, without taking any other action. Deleterious effects include growth of executable size by repeated infection.

- *Advertizing*-annoying messages or sounds are produced whenever the infected executable runs.

- *Destruction of data*-virus deletes or mangles files at random, reformats disks, etc.

- *System disruption*-some viruses infect the boot program or other parts of the operating system, making the system unbootable or unreliable.
The Confinement Problem*

The problem: program A (the customer) runs program B (the service). Can program B be prevented from transmitting information about program A or its owner?

Potential leaks:

- If the service has static memory, information about the customer can be collected and then delivered when the owner of the service next calls B.
- The service may write information about the customer to a file in the service owner’s area.
- The service may create a temporary file and grant access to its owner.
- The service may send messages via the OS Interprocess communication mechanism.
- The service may encode data in its pattern of requests for locks on a file owned by its owner, or its pattern of I/O requests.

Confinement

- The service must be memoryless
- If the confined program calls other programs, they too must be confined. (Requires a trusted OS)
- Any storage used by the system must be monitored by the OS to be certain it is not being used as a covert channel.

---

* Butler W. Lampson, “A Note on the Confinement Problem,” CACM, October 1973
Passwords

Problems

• People don’t like passwords
• People forget passwords
• People choose easy to guess passwords
• People don’t change passwords often enough

Good Passwords

• Avoid anything that appears in system dictionary
• Avoid proper names
• Avoid city, state, country, etc. names
• Avoid license plate numbers, phone numbers, SSN, etc.
• Avoid any of the preceding in reversed order
• Should be at least six characters long
• Should be a mixture of upper and lower case, and if numbers and punctuation are included, so much the better
• Pronounceable passwords improve memory retention

Other precautions

• Salting
• Shadow password files
File Protection Mechanisms

A computer system contains many *objects* which need to be protected from unauthorized access (files, devices, processes, semaphores, etc).

Typically, a system has a set of *access rights* to the objects (read, write, execute, etc).

A *protection domain* is a set of ordered pairs of objects and access rights. Each pair describes the type of access that is allowed to the object.

When a process is a “member” of a domain, the set of objects it may access and the operations it is allowed to perform on the objects are determined by the domain.

This results in a matrix rows = domains and columns = objects

<table>
<thead>
<tr>
<th>Domain</th>
<th>File 1</th>
<th>File 2</th>
<th>Printer 1</th>
<th>Printer 2</th>
<th>Dom 1</th>
<th>Dom 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rwx</td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>rwx</td>
<td>rwx</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typically, this matrix is very sparse.
Protection Domain Implementation

Access Control Lists

Each object has associated with it an ordered list of domains and the rights each domain has to the object.

- Only domains which have rights to the object appear in the list.
- Can be used to exclude specific domains (i.e. users) from accessing an object, even in case where the “group” the user belongs to are granted access.
- Owner of the object can change the access control list at any time.
- ACL could be in a separate disk block pointed to by the i-node or directory entry for the object.
- Implemented in Multics. Unix scheme provides a subset of this functionality.

Example:

<table>
<thead>
<tr>
<th>Object</th>
<th>Access Control List</th>
<th>(format: [user, group, rights])</th>
</tr>
</thead>
<tbody>
<tr>
<td>passwd</td>
<td>[* , system, rw-], [* , *, r--]</td>
<td></td>
</tr>
<tr>
<td>annfile</td>
<td>[ann, <em>, rw], [</em> , student, r--]</td>
<td></td>
</tr>
<tr>
<td>emacs</td>
<td>[bob, system, rwx], [* , *, r-x]</td>
<td></td>
</tr>
<tr>
<td>billprog</td>
<td>[bill, *, rwx], [hacker, <em>, ---], [</em> , student, r-x]</td>
<td></td>
</tr>
</tbody>
</table>
Capability Lists

Corresponds to recording (object, rights) pairs for each domain.

The list for a domain is called the *capability list*, individual members are *capabilities*.

- C-lists are themselves objects, with associated rights
  - Copy capability-new copy can be given another user
  - Copy object-new object with new copy of capability
  - Remove capability
  - Remove object-removes both the object and its capability

- Revoking access is difficult because capabilities can be duplicated and passed around between domains.
Case Study: Unix Fast File System

Physical Organization

Disks are divided into partitions, which are composed of a set of contiguous cylinders.

The partition begins with a boot block, which describes the device to the OS at boot time.

Each disk contains a number of file systems, each file system residing on a separate partition.

A file system’s attributes are described by a super block, located at the beginning of the file system’s disk partition after the boot block.

- Size of blocks for file system
- Number of blocks in file system
- Maximum allowed number of files

Each file system is divided into a set of cylinder groups, a smaller set of contiguous cylinders.

Each cylinder group contains:

- Duplicate of super block
- I-nodes for the cylinder group (cylinder group size / 2048 = # of inodes allocated)
- Bitmap describing available blocks
- Summary of data block usage for the file system
Block size

• Blocks are 4096 bytes to increase efficiency of disk access.
• Blocks can be divided into between 2 and 8 smaller chunks called fragments. 1024 bytes is typical size.
• Fragments can be allocated individually for small files. This reduces internal fragmentation.
• The bitmap of allocated memory records usage at the fragment level.

Consequences

• Disk address is now (block, fragment) pair
• Fragments of a single block can be allocated to different files
• If data must be appended to a file whose last block is fragmented, if there is not enough free space in the last block of the file, the data are moved to a block with enough free space. This avoids multiple fragmented blocks at the end of a file.
• If files are extended a fragment at a time, the last block of the file may be copied many times.
Layout Policy

**Policy**: All inodes of the files in a directory should be in the same cylinder group.

**Rationale**: The inodes of a directory are often referenced together, as in the `ls` command, or wildcard processing.

**Policy**: Place new directories in cylinder groups with more than average number of free inodes.

**Rationale**: Too much locality can cause crowding in a cylinder group and lead to poor access performance.

**Policy**: Place all blocks belonging to a file in the same cylinder group, preferably at rotationally optimal positions.

**Rationale**: The blocks of a file tend to be referenced together, e.g. the `cat` command.
“Local” Layout Policies

The global layout policies make requests for specific blocks. Sometimes these requests cannot be answered. To choose the block to allocate, the following heuristics are used.

- Choose the rotationally closest block on the same cylinder
- If the cylinder is full, choose a block from the same cylinder group.
- If the cylinder group is full, hash the current cylinder group number to choose a new cylinder group.
- If the hashing fails, search exhaustively for a new group.
Input/Output

Device Types

**Block**  Devices which are accessed using a fixed block size. Most allow random access of blocks. Primarily disk drives of various sorts, but tape drives can usually be treated as a special case which does not allow random access.

**Character**  Devices which provide or accept a stream of characters with no other structure imposed. Terminals, printers, network interfaces, etc. Some block devices can also be treated as character devices. (Also called *raw* devices.)

Devices communicate with the rest of the system via a *device controller* (which may also be called a `<blank>` interface or `<blank>` card, where `<blank>` is the type of device). This piece of hardware handles the low level interaction between the device and the system.

- Controller can issue commands to the device and check the status or result of the command.
- Typically, the controller has some buffer space and may be able to use DMA to move data to and from system memory.
- The system communicates with the controller via a set of registers on the controller which can be accessed from system memory, i.e via *Memory Mapped I/O*. 
Structure of I/O Software

Good I/O software design dictates a layered structure:

- **Interrupt handlers** - typically, a process blocks when it does an I/O operation. This layer of software handles gracefully waking the process up when the completion interrupt comes in. (May be integrated with the device driver.)

- **Device Drivers** - all device-dependent code should be isolated in the device driver. The device driver translates I/O requests into the actual commands that must be sent to the device, providing an abstract interface to the next layer up.

- **Device-independent OS software** - presents as uniform an interface to all I/O devices as possible. Handles naming, protection, buffering, sharing, etc.

- **User level software** - programmatic interface to I/O functionality, usually embodied in a library of functions for handling input/output tasks. In Unix, this is known as the `stdio` package.
Example: InterLan NI1010 Ethernet Interface

**Hardware Interface:** The NI1010 has three memory mapped registers which the device driver manipulates:

- **Command and Status Register (CSR)**-writing this register causes a command to be executed. Certain bits can be queried to determine the outcome of an operation.

- **Buffer Address Register (BAR)**-write-only register which device driver loads with the memory address of the buffer the controller should work with.

- **Byte Count Register (BCR)**-write only register for the length of the buffer whose address is in the BAR.
Command and Status Register format

<table>
<thead>
<tr>
<th>Address extension bits (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Code (W)</td>
</tr>
<tr>
<td>Command Done Bit (R)</td>
</tr>
<tr>
<td>Command Done Interrupt Enable (RW)</td>
</tr>
<tr>
<td>Receive Done (R)</td>
</tr>
<tr>
<td>Receive Done Interrupt Enable (RW)</td>
</tr>
<tr>
<td>Command Status Code (R)</td>
</tr>
</tbody>
</table>

Commands:

- **ILC_PRMSC**: Turn "promiscuous" mode on (listen to all traffic, regardless of address)
- **ILC_CLPRMSC**: Turn promiscuous mode off.
- **ILC_STAT**: Return statistics information and reset on-board counters.
- **ILC_XMIT**: Transmit a ready packet.
- **ILC_RCV**: Move a received packet from on-board buffers to main memory.
- **ILC_DIAG**: Run on-board diagnostic tests.
- **ILC_OFFLINE**: Set interface offline.
- **ILC_ONLINE**: Set interface online.
The ioctl Mechanism

Unix provides the ioctl call mechanism for interaction between user level processes and the device drivers for various devices. The set of available functions depends on the object being accessed.

```
ioctl (int fd, int request, char *arg)
```

- **fd**: open descriptor for the device (returned by the `open` system call)
- **request**: integer value indicating function to perform
- **arg**: pointer to data to be used by the function or to be filled in by the function.

File I/O Examples

- **FIONREAD**: Return the number of immediately readable bytes in an int pointed to by arg.
- **FIONBIO**: Set or clear non-blocking I/O, depending on the value of an int pointed to by arg.
- **FIOASYNC**: Set or clear asynchronous I/O, depending on the value of an int pointed to by arg.

`fd` can refer to a regular file, a character special device (e.g. terminal), named pipes, sockets.
Disk Arm Scheduling

**First Come-First Served**-service requests in the order that they arrive. May require long (and slow) seeks at times.

**Shortest Seek First**-analogous to shortest job first; next request serviced is the one that requires the shortest seek. Improves average request latency over FCFS. Favors requests in the middle of the disk.

**SCAN (Elevator) Algorithm**-service request from lowest to highest track number, then reverse and service from highest to lowest. Usually slightly worse than SSF. Eliminates any bias based on track number.

**Circular SCAN (Modified Elevator) Algorithm**-instead of reversing directions after highest track request, return to lowest track and service again in the same direction. Has smaller variance in response time than original Elevator Algorithm.

Trends in disk technology are reducing the size of the platter, resulting in seek times decreasing while rotational latency stays constant. When seek time becomes less than rotational latency, algorithms which optimize based on rotation rather than seek time should be used.

**Redundant Array of Inexpensive Disks (RAID)**-Each bit of a word is stored on a separate disk, with additional disks used to store check bits for an error correcting code. Even if some disks fail, the data can still be correctly deduced.
Memory Mapped Terminal I/O

- Video Controller card contains memory mapped buffers which system uses to store information to be displayed on screen.
- Controller circuitry reads bytes from video memory and drives display hardware appropriately.
- Character-mapped terminals expect ASCII characters to be stored in the VRAM, and convert each character to commands that draw the character on the screen.
- Bit-mapped terminals expect data which describes which pixels on the screen to turn on. Monochrome uses one bit per pixel, greyscale or color use more.
- Keyboard is a separate device.
Serial Terminal I/O

- Terminal is a separate device connected to computer via an RS-232 serial line (or coax cable if you’re IBM)
- The System writes characters to the serial interface, which contains a Universal Asynchronous Receiver Transmitter that converts them to a stream of bits for transmission over the serial line.
- A UART in the terminal’s serial interface circuitry reverses the conversion and sends the characters to the terminal’s video control circuitry.
- Most terminals contain a microprocessor which manages interaction with the video circuitry much the way the CPU does for a memory mapped terminal.
Terminal I/O Software

Terminal drivers usually provide a choice of two modes:

- **Raw mode** - driver passes the raw stream of characters generated by the keyboard to the process reading the terminal.

- **Cooked mode** - driver does some intermediate processing on the stream of characters before passing it on to the process.
  - Backspace removes character from buffer rather than passing along the character and the backspace
  - Line kill character deletes the entire line before it is passed to the process reading the terminal.

Other tasks handled by the terminal device driver

- Buffering (typeahead)
- Echo
- Tab handling
- Carriage return/line feed
- Control Character processing
Unix Case Study

- Unix Pedigree

- Standards Efforts

- Review of Process and Memory Models

- Paging details

- I/O Handling
Unix Pedigree

AT&T Unix

AT&T Version 6

AT&T Version 7

BSD1-BSD3

Minix (PC)

System V

System V Release 3

BSD4.2

BSD4.3

Mach

System V R4

SunOS 4

SunOS 5 (Solaris)

BSD4.4

Ultrix

GNU

Hurd

NeXT

Other Variants
HP/UX - Hewlett Packard
AIX - IBM
A/UX - Apple
IRIX - Silicon Graphics

PC Versions
Xenix (Microsoft/SCO)
Linux (Freeware)
Minix (Educational)
NetBSD
Unix Milestones

- 1969 - first version of Unix written at Bell Labs on PDP-7 to support Ken Thompson’s *Space Travel*.
- 1970 - Port to PDP-11
- 1973 - Unix Version 4, the first Unix written in C.
- 1973 - UC-Berkeley receives the first Unix License.
- 1976 - Port to Interdata (Bell Labs/Wollongong).
- 1979 - Version 7, first truly portable Unix.
- 1979 - Port of Version 7 to DEC VAX (Unix/32V)
- 1979 - 3BSD, incorporating paged virtual memory, released for the VAX (Based on Unix/32V).
- 1980 - 4.0BSD, including TCP/IP networking.
- 1982 - AT&T allowed to enter computer business, begins sale of System III.
- 1983 - AT&T releases System V.
- 1984 - X/Open formed to promote Common Applications Environment (based on existing de facto standards). Publishes X/Open Portability Guide (XPG), based on Posix.1
- 1987 - AT&T and Sun collaborate on SVR4
- 1988 - AT&T, Sun, other Sys V vendors form Unix

- 1991 - Linus Torvalds completes the first Linux kernel
- 1993 - Solaris 2 supports fully preemptible kernel, symmetric multiprocessing
<table>
<thead>
<tr>
<th>Effort</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVID</td>
<td>System V Interface Description-specified system calls, file formats, etc. Ignored by everyone but AT&amp;T.</td>
</tr>
<tr>
<td>Posix</td>
<td>Portable Operating System-Unix. Formulated by IEEE working group 1003.1. Includes specifications of library interface to system calls, Posix shell and utilities, language bindings for library, etc. Intersection of features of System V and BSD, with BSD signal handling and new terminal handling system.</td>
</tr>
<tr>
<td>OSF</td>
<td>Open Software Foundation-vendors who decided to gang up on AT&amp;T. OSF/1 is Posix compliant, but includes X11 plus Motif, distributed computing and management specifications, etc.</td>
</tr>
<tr>
<td>UI</td>
<td>Unix International-AT&amp;T’s response to OSF</td>
</tr>
<tr>
<td>FSF</td>
<td>Free Software Foundation, Richard Stallman’s soapbox for his vision of a free software utopia. Working on GNU (Gnu’s Not Unix), a completely rewritten and freely redistributable Unix, called Hurd, based on the Mach kernel.</td>
</tr>
</tbody>
</table>
Unix Processes (Review)

- Unix allows multiple processes, each with a single thread of control.

- Processes are created by the fork system call, which creates an exact copy of the “parent” process, including open file descriptors.

- Processes can communicate with each other:
  - Pipes - messages can be passed via read/write system calls
  - Signals - a form of software interrupt which can be “caught” and “handled” by a process

- Processes have a “user part” and a “kernel part” with separate program counters and stacks. Kernel part handles system calls.

- Unix uses two structures for managing processes:
  - Process table - always memory-resident, contains information needed for scheduling and swapping, signal handling, alarms, etc.
  - User structure - swapped out with the process, includes register images, system call status, open file descriptors, accounting information, and the kernel stack.

- Scheduling uses a multi-level feedback algorithm
Unix Memory Management (Review)

• Each process is composed of four segments
  - text-the executable code for the program running in this process. Can be shared between processes.
  - data-initialized variables and strings (stored on disk with program binary).
  - BSS-uninitialized data
  - stack-used to handle function calls

• Swapping
  - Eviction
    • Choose blocked process if possible
    • Choose process with largest recent CPU consumption
    • Never swap out a process that has been in memory less than 2 sec (to prevent thrashing)
  - Reactivation-Choose ready process with longest “swapped out” time.

• Paging
  - Demand paging is used. Only process table entry and user structure need to be in memory for process to be “runnable.”
  - Page replacement algorithm is “double-handed clock.”
    • Global algorithm
    • Checks only a window of pages
    • May not check whole list at each pass.
  - Uses a core map to describe contents of each page.
Core Map Layout

- Index of next entry
- Index of previous entry
- Disk block number
- Disk device number
- Block hash code
- Index into process table
- Text/data/stack
- Offset within segment

Free
Wanted
In transit
Locked in memory

Core map

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5

Core map layout with various fields and pages.
I/O Handling

Block Special Devices

- Goal: minimize number of actual transfers
- Accomplished using a buffer cache
  - Holds the most recently referenced blocks from the device.
  - Hits result in no new device activity.
  - Blocks are arranged in a linked list ordered by time of reference (practical because disk cache is relatively small).
  - “Dirty” blocks are flushed to disk periodically

Character Special Devices

- Mechanism to allow “cooked” streams needed
- Input is buffered in a C-list -- a linked list of small (up to 64 characters) buffers
- Each c-list entry contains a group of characters, a count, and a pointer to the next member of the c-list.
- Before being passed to a process doing a read or an output device, the c-list is filtered through some kernel code called a line discipline which performs some intermediate processing:
  - For input, processing consists of intra-line editing
  - For output, may be things like tab expansion, handling delays for slow display equipment, linefeed/CR processing, etc.
Linux Case Study

Linux is a freely redistributable Unix "work-alike" O.S.

- Original versions ran only on Intel based systems. Support has expanded to include Sparc, DEC Alpha, Motorola 68000 and PowerPC architectures.

- Linux kernel was written from scratch and supports basic OS facilities.

- The Linux "system" is a complete operating environment incorporating many other freely available components (BSD software, GNU software, X-Window System, etc.)

- Linux is "open source software" maintained by a large group of volunteer developers and distributed under the GNU General Public License.

Linux Design Principles

- Monolithic microkernel architecture
- Supports multiple users, multitasking
- Uses standard Unix file system layout
- Unix networking model is implemented
- POSIX compliant
- SVR4 programming interface, but provides BSD compatibility libraries
- Supports dynamically loadable kernel modules
MS/DOS Case Study

- Historical background

- Process handling

- Memory model

- File system

- System Calls
MS-DOS Background

- Hobby computing (ca. 1975-1980)-the Altair and other 8080 based computers (some kit-built) are the only PCs. Gates writes small BASIC interpreter that runs on these machines. (8080 has 8 bit data paths and can address 256 bytes of memory.) CP/M is dominant operating system.

- IBM decided to do a quick ‘n‘ dirty development of a personal computer.
  - Used 8088 rather than 16-bit 8086 because it was less expensive.
  - Decided to divide address space into lower 640K for RAM, upper 384K for ROMs, video memory, etc.
  - No protection hardware.

- Tried to buy CP/M to run on it, but version for the 8088 was behind schedule. Gates bought a test harness (86-Dos) from a company that built memory boards, and hired its author to pretty it up for IBM. MS-DOS 1.0 was born.

- Features
  - Flat directory structure
  - Ran many CP/M 8080 programs unmodified
  - Used BIOS (Basic Input Output System-in ROM) for manipulating standard devices (console, printer, serial ports)
  - Supported batch files.

- Version 2.0 was completely rewritten for release of PC/XT. Contained many features lifted from Microsoft’s Unix product.
- Tree directory structure
- Unix-style I/O system calls (with protection mechanism deleted)
- Shell allowed redirection of I/O, supported process pipelines (implemented with temp files)
- Support for user-installed device drivers, print spooling, flexible system configuration, memory management, and customized shells.

- Version 3.0 was released with PC/AT, first 80286-based PC. Though the 286 could have supported a form of multi-tasking, Microsoft ignored this -- MS-DOS 3.0 used the PC/XT as a faster 8088.
  - Supported some new devices, made shell a separate program, allowed RAM disks
  - 3.1 provided minimal networking capability

- Version 4.0 was released to submarine the elegant but unpopular IBM/Microsoft collaboration for the PS family, OS/2.
  - support for larger disks
  - DOS shell-menu driven interaction with DOS
  - Programs still limited to maximum 640K, but the 16M memory could be used as a huge RAM disk to improve disk performance.

- Version 5.0
  - Improved use of extended memory.
  - Multiple shell which allowed switching between several memory resident programs using a hot key.
  - Updated utilities were included.
• Windows
  - Graphical user interface pasted on top of and poorly integrated with MS-DOS
Basic MS-DOS Operation

- Power-on causes a jump to hardwired BIOS address
- This is a jump instruction to the BIOS bootstrap program
- Hardware diagnostics are performed
- Boot sector read from A: drive
- Primary boot sector read from hard disk (usually C:). This contains the partition table, which indicates the active partition.
- Read secondary boot sector from active partition. (Allows automatic booting from MS-DOS or other OS, selectable at boot time.)
- Read root directory for `io.sys` and `msdos.sys` and transfers control to `io.sys`.
- `io.sys` initializes hardware, calls `sysinit`, reads `config.sys`
- Transfers control to `command.com`. 
MS-DOS Process Handling

- Supports multiple processes in a limited fashion. A new process can be started, but the parent process must be suspended until it completes. A chain of such processes can exist, but only the process at the end of the chain is actually executing.

- Executables come in two flavors, which are treated differently:
  - .com files—loaded into memory as-is and executed. This type of executable is a CP/M hangover.
    - Consists of a single segment which combines text, data, and stack.
    - Maximum size for the executable is 64K (memory model limitation)
  - .exe files
    - relocatable, so processed by OS before being loaded into memory to take into account their position. (No Base/Limit registers to support relocation.)
    - Have separate text, data and stack segments, plus as many “extra” segments as needed.

- OS prepends a Program Segment Prefix (PSP) to the executable before copying to memory
  - Contains program’s size, pointer to environment block, Ctrl-C handler address, arguments, pointer to parent PSP, file descriptor table, etc.
  - Counts as part of .com executable’s address space, .exe executable are relocated above the PSP.

- Child inherits parent’s open file descriptors.
• No paging support.

• Overlays are supported, but program must typically be structured in a special way to take advantage of it.

• Control can be returned to the parent without returning the child’s memory to the system. Referred to as Terminate and Stay Resident programs (TSR).
  - In combination with ability to install alternate interrupt handles, programs that behave this way can respond to hot keys
  - TSR’s cannot make use of system calls (for I/O, for example) without making use of undocumented features of MS-DOS.
Memory Model

Much of the idiosyncrasy of MS-DOS’s memory handling can be traced back to the Intel 8080 (ca. 1974). Intel decided to make improvements in the 8088, but to keep it backward-compatible with the 8080.

8088 Architecture

- Supported up to 1M of memory, but preserved 8080’s 16 bit addressing (16 bit addresses = 64K address space). This led to the dreaded segmented memory architecture.
- The 8088 register set includes four 16 bit segment registers, which are used to hold the upper 16 bits of a 20 bit address.
  - Memory addressed using these registers is thus divided into 16 byte chunks called paragraphs.
  - The separate registers are used for the different segments of a process, the code segment, data segment, stack segment, and extra segment.
  - Segments must begin on paragraph boundaries.
  - Before executing a fetch, the 16 bit address is “relocated” by adding the address in the appropriate segment register to it to form a 20 bit address.
- Because “real” addresses are still 16 bits long, the maximum size of a segment is still 64K, but the 64K can be located anywhere in the 20 bit address space provided by the segment mechanism.
- Real 32 bit addresses are supported with system calls, but they are slower than the 16 bit calls.
8088 Memory Architecture (continued)

- High Memory Area - segmented addressing scheme leads to possibility of addressing parts of memory above 1M (e.g., segment register value 0xFFFF refers to a paragraph that begins 16 bytes before the 1M boundary, but allows addressing 64K of memory starting at that point). 8088 wrapped these addresses back to the beginning of the address space, but later versions of the chip could actually use this memory. Most didn’t for the sake of backwards compatibility.

- Extended Memory - 286 and later processors have larger address spaces, but support backwards compatibility with a CPU mode called real mode. MS-DOS uses real mode, so working with memory above 1M is quite complicated.

- Upper Memory Area - the 384K between 640K and 1M. Reserved for BIOS and video RAM, etc. 386 and later chips could map the unused holes in this area and use them.

- Expanded Memory - hardware scheme that allows expanded data segments. (Overlay mechanism doesn’t work for data.) Because MS-DOS supports Extended Memory badly, it is often used to simulate Expanded memory.
MS-DOS Memory Management

- Free memory is managed as a group of *arenas*, which begin on a paragraph boundary and contain an integral number of paragraphs.

- Each arena contains a 16 byte header:
  - 2 byte magic number
  - Pointer to PSP of process using this arena, or 0 if free
  - Arena’s size in paragraphs
  - Name of executable binary file which “owns” the arena

- Arena list can contain arenas in Upper Memory Area if the appropriate system call has been made.

- Uses first-fit by default to allocate space, but a system call is provided to change this to best-fit or last-fit.

- Freed blocks are not immediately coalesced-this is postponed until the next full scan of the arena list because the “list” is not doubly linked.

- Doesn’t work in Extended Memory because pointers/sizes in arena header are only 16 bits. Use of Extended Memory requires installation of an extended memory device driver.
MS-DOS File System

- Each partition has a File Allocation Table (FAT) containing “linked lists” of the blocks in each file.

- Directories are simply files in a special format. Each directory entry contains:
  - File name
  - File extension
  - Attributes (only thing not provided by Unix protection mechanism is Archived bit)
  - Creation/modification dates
  - Block index of first block (other blocks are found by referencing the FAT)
  - Size of file

- No links are possible (consistency problem)