Contents

1 00 Intro

Chapter 0

CSci 451 is a theoretical introduction to operating systems. The student will be introduced to the underlying concepts behind all operating systems. The student will also explore the UNIX, Linux, and WindowsNT operating systems in detail. Finally, the student will work with UNIX system calls to gain an understanding of the power of UNIX (and other operating systems as well) and to gain a better understanding of the design philosophy behind UNIX.

TOPICS:

Topics include, but are not limited to:

- CPU Scheduling
- Memory Management
- Storage Systems
- I/O Systems
- Protection and Security
- UNIX
- Linux
- WindowsNT
- UNIX System calls

UNIX INTRODUCTION:

UNIX Commands:

- pwd - Print working (current) directory (like "cd" in dos).
- grep – Searches the specified files for the string specified
  (grep string filenames → grep help *.cpp).
- man - The on-line UNIX manual (try: man man).

Terminal Commands:

- Control U - Cancels the line you're typing.
- Control C - Interrupts the program currently executing.
- Control \ - Interupts the program currently executing
  (stronger than control C and produces a core dump).
- Control Z - Suspends the execution of the current program
  (execution can be resumed later).
- Control S - Stops output sent to terminal (suspends program).
- Control Q - Restarts execution of stopped program.

UNIX INTRODUCTION:

Process Commands:

- ps - Displays your processes
- ps -f - Displays all info on your processes.
- ps -A - Displays info on ALL processes running on the machine.
- ps -fA - Displays all info on ALL processes running on the machine.
- jobs - Displays jobs (including stopped jobs)
- kill - Used to remove a process
  kill -pid - Kills a process.
  kill %jobno - Kills a job.
- kill -s 9 pid - Kills a process by sending the KILL signal (stronger than kill pid).
- fg - Used to restart (move to foreground) a stopped job (fg %jobno or fg pid)
- bg - Used to move jobs to the backround (bg %jobno or bg pid)

UNIX INTRODUCTION:

REDIRECTION:

I/O includes standard input, standard output, and standard error. Redirection lets you reroute the I/O from any of these to or from a file.

<	redirect standard input
> or 1>	redirect standard output (overwrite)
>> or 1>>	redirect standard output (append)
2>	redirect standard error (overwrite)
2>>	redirect standard error (append)

Format: command < input > output

```
cat < file_in > file_out
```

PIPES:

Redirects output from 1 program into another.

Format: command | command

```
ls | more
cat < source_file | sort
cat < source_file | sort > result_file
```

UNIX INTRODUCTION:

ALIASES:

Aliases can be used to define new names for commonly used commands.

alias *newname*="unix_command" - Defines *newname* to be an alias for the UNIX command “unix_command”.

```
alias dir="ls -la"
```

unalias newname - Removes the alias *newname*.

```
unalias dir
```

Since aliases do not remain in effect if you close the session, you can put commonly used aliases in your ".profile" file.

UNIX INTRODUCTION:

LINKS:

Links allow the creation of an alias or a nick name for a file. Changes to the link affect the file and the "rm" command removes the link, but not the file. There are 2 types of links:

1. "hard" links - A UNIX file can have multiple "hard" links. "Hard" links and the file must be on the same file system.
   ```
   link file linking_name
   ln file linking_name
   ```

2. "soft" links - Aka symbolic link. A symbolic link can be created across file systems. A UNIX file can have multiple symbolic links. Symbolic links can also refer to a directory.
   ```
   ln -s file linking_path/file
   ```

Warning symbolic links can create circular references and deletion of the linking_path/file results in an undefined link.

unlink - Removes a link ("hard" or "soft").

```
unlink linked_name
```

UNIX INTRODUCTION:

PERMISSIONS:

File / directory permissions include read (r), write (w), and execute (x). Permissions can be set for the owner (u), members of the owners group (g), and all others (o). The UNIX command "chmod" is used to change permissions.

File permissions:

- r - Required to be able to read a file.
- w - Required to be able to write a file (edit requires r and w).
- x - Required to be able to execute a file. Shell scripts (batch files) require r and x to execute.

Directory permissions.

- r - Lets one see what's in the directory, but nothing else.
- w - Lets one see what's in the directory, but nothing else(?).
- x - Required to do anything with the directory or its contents.

To view the file permissions in your account use the "ls -la" command (list ALL files using a LONG display) or use the "ls -l" command.

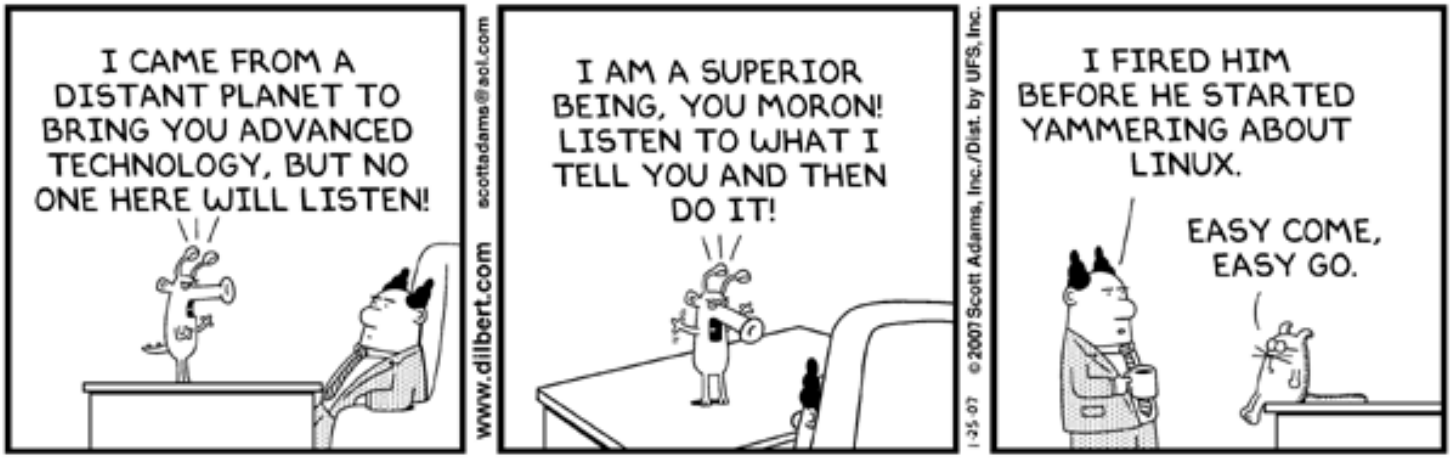
UNIX INTRODUCTION:

PERMISSIONS:

You can also set File / directory permissions using a numeric code. One that is very useful is:

```
chmod 755 file_name
```

As this make the file executable (like a batch script).



2 01 OS intro

Chapter 1

OPERATING SYSTEMS:

Objectives:

1. Convenience for the user.
2. Efficient operation of the computer.
3. Ability to evolve.

1 and 2 conflict. So we look for compromises between them based on machine usage (WindowsNT 3.51 vs WindowsNT 4.0 or Windows 98 vs Windows XP).

Notes:

According to some authors, the OS is the kernel and everything else is an application.

What about loadable device drivers (i.e. modules in Linux) vs monolithic kernels?

OPERATING SYSTEMS:

Tasks:

- Program development (Command-interpreter (shell), GUI, etc).
- Program management/execution.
- I/O device management.
- File management.
- System access (main memory management, networking, etc).
- Error detection.
- *Accounting*.
- Protection <- missed by our author!
- *Shared memory (SMP systems).*
- *Message passing (distributed systems).*

OPERATING SYSTEMS:

Dual-mode Operation (Two levels of commands):

1. User mode – Given to users and their programs.
2. Privileged mode – Given to the OS.

All I/O instructions are privileged instructions. User's programs that require I/O must request the I/O from the OS.

CPU protection is provided by a timer (To prevent a program with an infinite loop from hogging the CPU for evermore. Obviously this timer cannot protect the user from an infinite loop).

Discussion: Since the OS is given unrestricted access to the system (memory, CPU, disk, etc), could a terrorist organization use this to crash America's information network.

OPERATING SYSTEMS:

Structure:

A well structured OS design improves maintainability, can simply the design and development process, and eases verification and validation. Generally, well structured means a layered design.

However, layering decreases efficiency by forcing extra function calls (each function call typically requires storage of the machine's state).

Examples:

- MSDOS – Poorly structured. Levels of functionality and interfaces are net well defined or separated. User written programs can directly access the hardware (and sometimes, they have too!).
- UNIX – Originally, had little structuring (not much better than MSDOS). Newer versions have better / more structuring (The Mach microkernel takes structuring to the extreme).
- OS/2 – Arguably the best OS design to date (for PC's at least).

OPERATING SYSTEMS:

Types:

1. Batch systems.
   Jobs sorted by hand (assuming the job control card was filled out accurately)
   Jobs were primarily CPU intensive (only I/O was tape and paper printout)
   Spoolers are batch systems.
2. Multi-programmed batch systems.
   Use a job pool (on tape, disk, or in memory) to allow better use of system resources. Jobs are swapped in and out to keep CPU busy.
   Requires some form of job scheduling.
3. Time-sharing.
   Time limit or I/O causes a job swap.
   Requires an on-line file system (disk).
   Provides an interactive environment.

Discussion:

- Batch systems matched up well with the hardware of the time. The only common I/O were tape drives and printers. CPUs were slow and memory was very expensive.
- Supercomputers still use the multi-programmed batch system. Why?
- Most modern OS are time-sharing and swap times can be a concern.

OPERATING SYSTEMS:

Parallel systems:

Asymmetric – Different processors have different tasks.

- Front-end machine – like a keyboard buffer.
- Back end machine – like a disk controller chip or graphics accelerator chip.

Symmetric – (SMP) Processors run identical copies of the OS, but the actual jobs may be different.

- The Safeguard site in ND.

Distributed systems:

- Each processing node is a different machine, all connected by a network.
- A Linux print server would fall into this category.

INTERRUPTS:

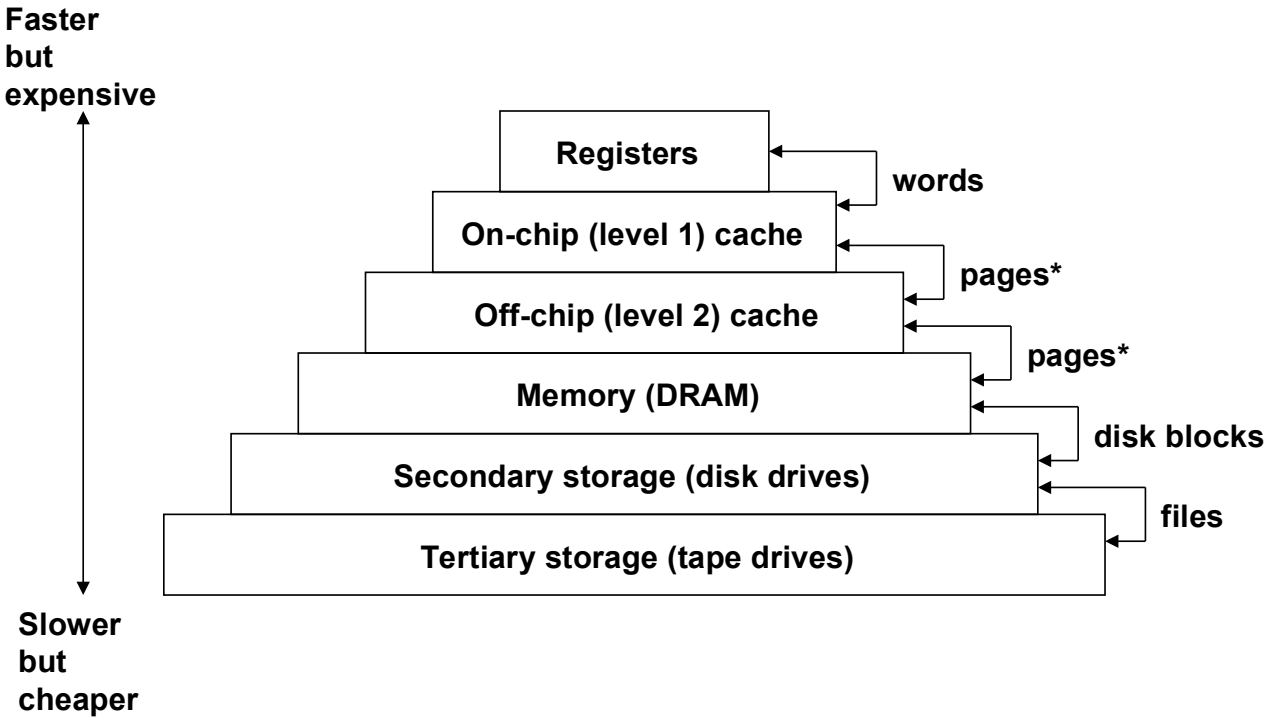
Four Types:

1. Software – Generated by the software (system calls).
2. Timers – Generated by timer (ie CPU).
3. I/O – Generated by an I/O controller.
4. Hardware – Generated by the hardware (usually for some failure).

Notes:

- Interrupts disable (delay) other activities until the current interrupt has finished.
- Interrupts force the machine state to be saved (costs CPU cycles).
- Interrupt service (interrupt-handler) routine locations (addresses) may be stored in a look-up-table (LUT) called an interrupt vector (DOS and UNIX). The idea is to speed up interrupt service execution.

MEMORY HIERARCHY:

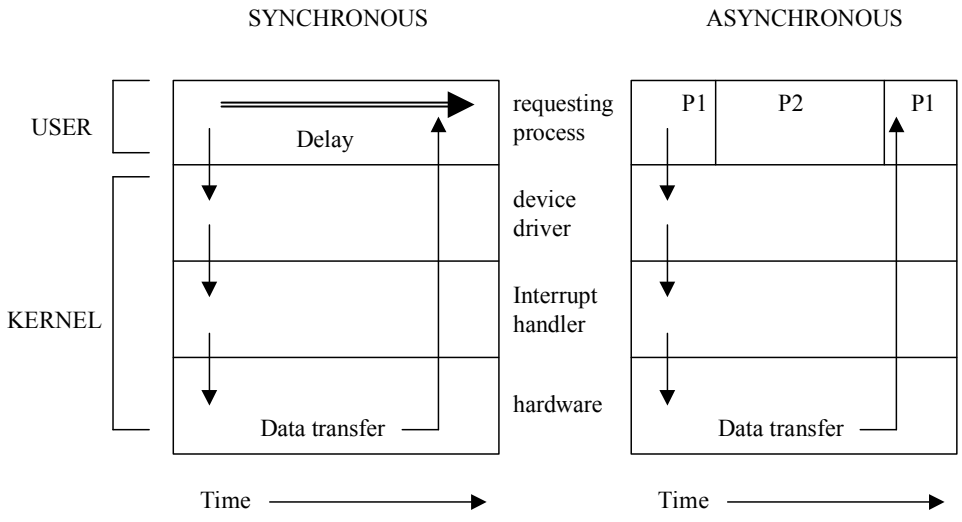


*** In many systems disk blocks = pages.**

I/O:

Three Types:

1. Programmed – synchronous.
2. Interrupt driven – asynchronous.
3. DMA – Direct Memory Access.



I/O:

DMA:
Due to the overhead associated with I/O (transfer a byte, check for errors, set flags and pointers, etc) a faster method of data transfer can be of use. This is especially true for devices that must transfer large blocks of data into main memory.

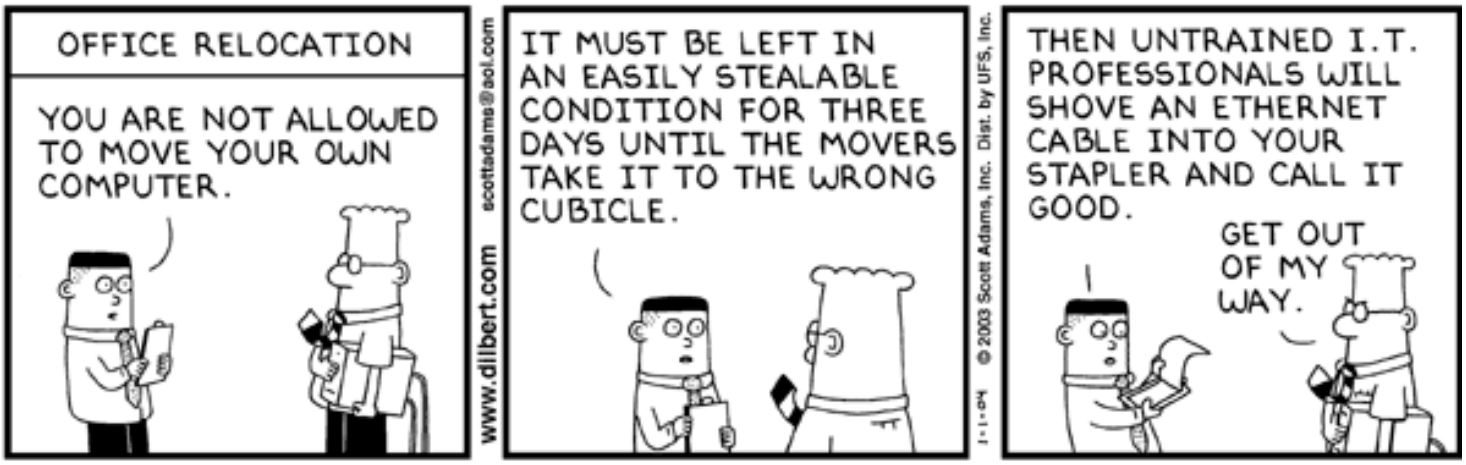
The DMA device transfers data on its own to the appropriate memory locations. The CPU is left free to do other tasks.

However, since the memory controller can only transfer 1 word at a time, if the CPU requires memory access it will be delayed until the DMA transfer is completed.

Memory-mapped I/O and ports:
Memory-mapped I/O and ports are primarily used to ease access to the various devices.

OS HISTORIES:

Unix History:
http://www.youtube.com/watch?v=7FjX7r5icV8&feature=related


OFFICE RELOCATION
YOU ARE NOT ALLOWED TO MOVE YOUR OWN COMPUTER.
www.dilbert.com scottadams@aol.com
IT MUST BE LEFT IN AN EASILY STEALABLE CONDITION FOR THREE DAYS UNTIL THE MOVERS TAKE IT TO THE WRONG CUBICLE.
1-1-04 © 2003 Scott Adams, Inc. Dist. by UFS, Inc.
THEN UNTRAINED I.T. PROFESSIONALS WILL SHOVE AN ETHERNET CABLE INTO YOUR STAPLER AND CALL IT GOOD.
GET OUT OF MY WAY.

3 02 OS system

Chapter 2

OS STRUCTURES:

UNIX Handouts for
•Linux commands

•Linux error codes

•Linux System Calls (including programming examples)

can be found at :

http://undcemcs01.und.edu/~ronald.marsh/CLASS/CS451/CS451.HTML

4 03 Processes

Chapter 3

PROCESSES:

PROCESS:

A program in execution (including the memory, stack space, program counter – i.e. more than just the code). In other words – a verb.

A process (*heavyweight* process) has the states (new, ready, running, waiting, terminating).

- Includes program code, data, system resources, memory, program counter, register set, and stack space.
- Can create child processes.
- Cannot be shared among different tasks.
- Both the OS and users can execute processes.

PROCESSES:

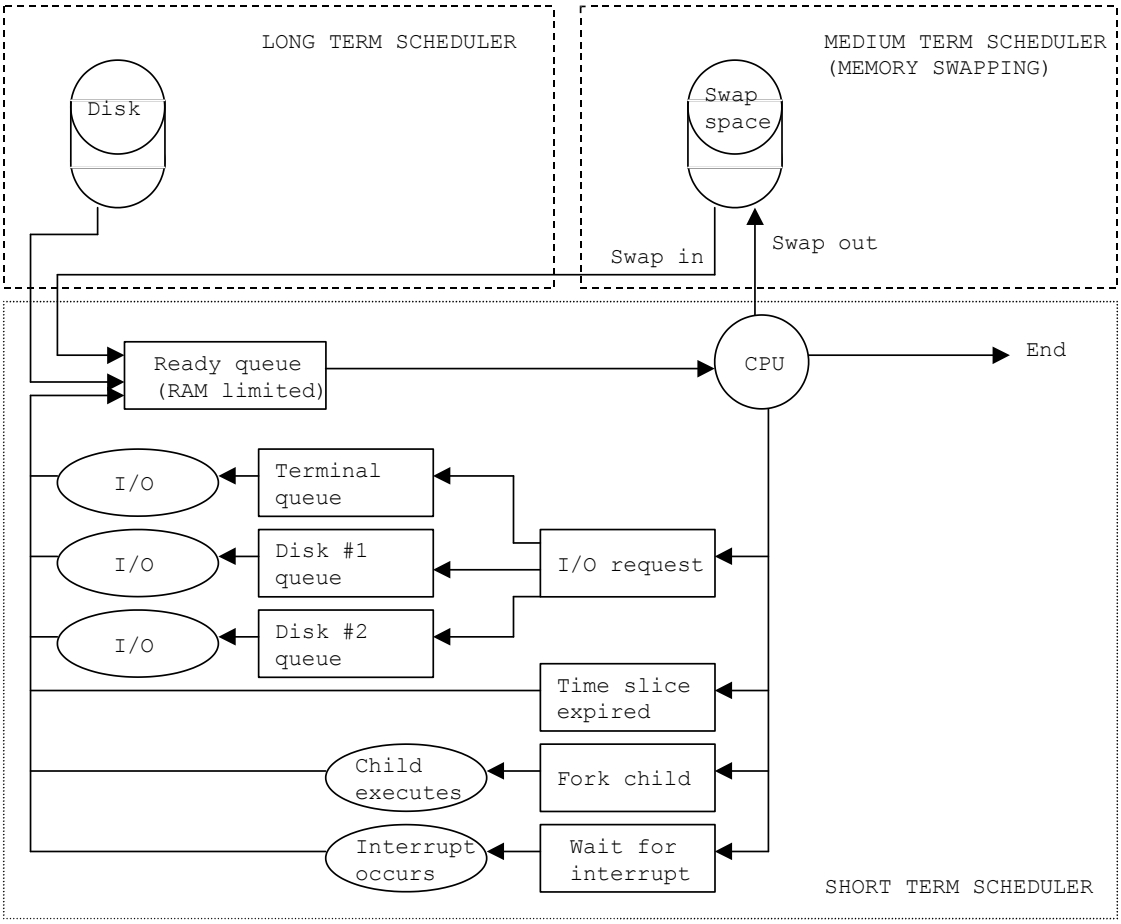
PROCESS CONTROL BLOCK (PCB):

A data structure containing:

- Process identifier.
- The process state (new, ready, running, waiting, terminating).
- Process priority.
- The program counter.
- Memory management information.
- I/O status information.
- Accounting information.
- CPU scheduling information.
- Etc (OS dependent).

On a single CPU time-shared system, only 1 process can be executing at any given instance. Therefore, we must occasionally swap the running process with a waiting process – this swapping is called a **context switch** (in reality we only swap the PCBs).

PROCESSES:


LONG TERM SCHEDULER
MEDIUM TERM SCHEDULER
(MEMORY SWAPPING)
Disk
Swap space
Swap in
Swap out
Ready queue
(RAM limited)
CPU
End
I/O
Terminal queue
I/O
Disk #1 queue
I/O request
I/O
Disk #2 queue
Time slice expired
Child executes
Fork child
Interrupt occurs
Wait for interrupt
SHORT TERM SCHEDULER

PROCESSES:

PROCESS CREATION:

- Parent – creates sub-processes.
- Child – the sub-process.

A child may get resources from its parent (forced to share the parents – prevents system overload) or from the OS.

A child may run concurrently with the parent or the parent may wait for the child to finish.

PROCESS TERMINATION:

Parent can kill child processes. Reasons include:

- Child exceeds resources allocated.
- Task assigned to child no longer needed.
- Parent is terminating (children should not exist without a parent). However, if a parent terminates before a child, the child may be adopted by the "init" process (pid = 1) and is allowed to run to completion.

Child can terminate and optionally return data to parent.

PROCESSES:

DEFUNCT / ZOMBIE:

When a child terminates, its resources are freed up, but the process itself is held in the process table (as a defunct) until the parent calls "wait" or "waitpid" on the child. However, if a parent fails to call wait (because it unexpectedly terminates), the process will be left in the process table (as a zombie).

Defunct → Dead Child
Zombie → Unclaimed dead child

In human terms.

Modern OSs occasionally call wait() to remove any Defuncts / Zombies.

ORPHAN:

A parent (unexpectedly) terminates. The child is adopted by the "init" process (pid = 1) as expected.

Orphaned → Orphaned Child

In human terms.

Note that an orphaned process is still alive!

The book's definitions are very vague – These definitions are what may be on a test!

PROCESSES:

Related UNIX commands:

fork – creates identical copy of the creating (parent) process. On completion, the fork command returns a zero to the child and the child's PID to the parent. A –1 is returned to the parent if the fork failed.

execv – loads a executable program into the memory space allocated by fork. The PID is not changed from that set by fork.

waitpid – forces parent to wait for a child process to terminate.

clone – creates identical copy of the creating (parent) process, like fork. Except, clone allows the child to share parts of the parent process (depends on flags set). So, clone creates a pseudo-thread. On completion, the clone command returns a zero to the child and the child's PID to the parent. A –1 is returned to the parent if the clone failed.

PROCESSES:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
void main(void) {
int   ExitCode = 1;
pid_t PID;
char *Command = "helloworld";
   printf("Spawning a child.\n");
   printf("-----------------\n");
   PID = fork();
   if (PID == 0) {
      printf("CHILD: Hello from the child.\n");
      execv(Command, NULL);
      printf("CHILD: exit code: %d\n", ExitCode);
      exit(ExitCode);
   } else {
      printf("PARENT: Spawning child PID: %ld\n", PID);
      system("ps -u rmarsh");
      waitpid(PID, &ExitCode, 0);
      printf("PARENT: Child exit code: %d\n", (ExitCode >> 8));
   }
}
```

PROCESSES:

COOPERATING:

- Independent processes – Processes cannot affect or be affected by another process.
- Cooperating processes - Processes can affect or be affected by another process.

Reasons for cooperative processes include:

- Information sharing (file handles, data structures, etc).
- Computation efficiency (with multiple CPUs, parallel processing possible – especially when data sharing is also possible).
- Modularity of design.
- Convenience (separate processes force/allow OS to allocate resources between the processes).


I'M PRACTICING MY URGENT-LOOKING WALK.
THIS WALK SAYS I'M WORKING ON STUFF THAT IS SO IMPORTANT I CAN'T RISK WASTING FIVE SECONDS.
WHO IS THAT MAGNIF-ICENT CREATURE?
THAT MAN HAS SOMEPLACE TO BE!
Dilbert.com DilbertCartoonist@gmail.com
8-21-13 ©2013 Scott Adams, Inc./Dist. by Universal Uclick

5 04 Threads

Chapter 4

THREADS:

A thread is considered a basic unit of CPU work (an atomic unit of work?).
A process (heavyweight process) is a task with 1 thread.

A thread (lightweight process):

- Has the same states as a process (new, ready, running, waiting, terminating).
- Has its own program counter, register set, and stack space.
- Can create child threads.
- Cannot be shared among different tasks.
- Can share with peer threads, the tasks PCB information.
- Allows more efficient CPU switching than context switching.
- Is not always supported by the system (OS, library).

THREADS:

WHEN TO USE PROCESSES VS THREADS:

•Processes – When the processes are not related (the OS insures that each process gets an equitable share of the resources).
•Processes – When the programming may be “sloppy” (with processes, each process’s memory area is protected – not the case with threads).

•Threads - When the threads are related, the sharing of resources or I/O devices makes for a more efficient use of system resources.
•Threads - When the threads are related, threads provide for a more efficient way of "context switching" between the different threads. However, this depends on the method of thread implementation (user level / kernel level).
•Threads – Given that threads share program resources, mulit-thread synchronization is easier.

THREADS:

WHEN TO USE KERNEL VS USER LEVEL THREADS:

- User level threads are (generally) supported via a library.
- User level threads provide the most efficient way of switching between “tasks”.
- User level threads make for easier synchronization.
- User level threads are more portable.

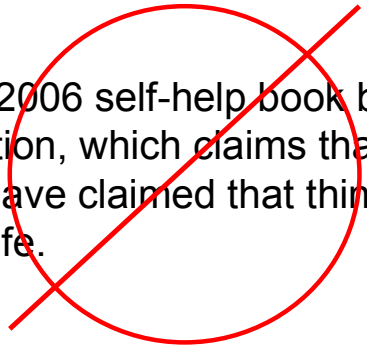
- Kernel level threads are supported (directly) by the OS.
- Kernel level threads provide better load balancing or better sharing of the system resources between all “tasks”.
- Kernel level threads prevent unnecessary blockages caused by OS system calls.

Since some OSs (Solaris) provide both implementations of threads, which do you use? For example: Assume you have 2 processes running on a machine, a web server (for a popular web site) and a seldom used database. What type of thread (for the web server) would best suit your needs and why?

THREADS:

THE SECRET:

The Secret is a best-selling 2006 self-help book by Rhonda Byrne. It is based on the belief of the law of attraction, which claims that thoughts can change a person's life directly. Some have claimed that thinking about certain things will make them appear in one's life.



I've always thought about being tall, dark, and handsome….

Still waiting….

Let's examine a secret worth something…

THREADS:

```
/*************************************************************************/
/* threadSample1                                                         */
/* Load balances by dividing data into N equal size blocks and           */
/* sends each block to 1 of N threads.                                   */
/*                                                                       */
/* runtime = 12 seconds.                                                 */
/*************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>

/*************************************/
/* Global variables.                 */
/*************************************/
#define threads 1
#define dataSize 800

struct thread_data {
   int start;
   int stop;
};
```

Required for threads.

Required for “The Secret”.

THREADS:

```
/*****************************************/
/* Thread code.                          */
/*****************************************/
void *TEST(void *arg) {
int    x, y, color;
int    exitCode = 1;
double xx, yy, xs, ys, xy, cx, cy;
struct thread_data *data;
   data = ((thread_data*)arg);
   for (y = data->start; y < data->stop; y++) {
       cy  = -5.0 + (y * 0.0125);
       for (x = 0; x < dataSize; x++) {
           cx = -5.0 + (x * 0.0125);
           xx = yy = xs = ys = xy = 0.0;
           color = -1;
           do {
              xs = (xx * xx);
              ys = (yy * yy);
              xy = (xx * yy);
              xx = (xs - ys + cx);
              yy = (xy + xy + cy);
              color++;
           } while ((color < 250) && ((xs + ys) <= 4.0));
       }
   }
   pthread_exit(&exitCode);
}
```

Threads are always pointers to functions (pretty cool, but not part of The Secret).

Thread code (generates a Mandelbrot fractal).

How to exit a thread.

THREADS:

```
/*************************************/
/* Main.                             */
/*************************************/
int main(void) {
int t, blockSize, start, stop;
time_t tstart, tstop;
struct thread_data data[threads];
pthread_t thread[threads];
   tstart = time(NULL);
   // Divide into blocks and send each to a thread.
   blockSize = dataSize/threads;
   start     = 0;
   stop      = blockSize;
   for (t = 0; t < threads; t++) {
       data[t].start = start;
       data[t].stop  = stop;
       pthread_create(&thread[t], NULL, TEST, (void *)&data[t]);
       start+= blockSize;
       stop += blockSize;
       if (stop > dataSize) stop = (dataSize-1);
   }
   // Blocking wait for thread completion.
   for (t = 0; t < threads; t++) pthread_join(thread[t], NULL);
   tstop = time(NULL);
   printf("Run time (sec) = %d.\n", (tstop - tstart));
}
```

How we create (launch) a thread.

How we wait for a thread to terminate.

THREADS:

Run times:

•12 seconds w/1 thread (Divides data into 800 equal size rows and sends each row to a thread.)

•10 seconds w/4 threads (Divides data into 800 equal size rows and sends each row to 1 of 4 threads. Detaches each thread and spawns another thread for the next row of data.)

•42 seconds w/4 threads (Divides data into 800 equal size rows and sends each row to 1 of 4 threads. Recalls the thread function for another row of data.)

•10 seconds w/4 threads (Divides data into 800 equal size rows and sends each row to 1 of 4 threads. The thread then waits to be signaled to process another row of data. The threads only terminate when all of the data has been processed.)

•>1 second w/800 threads (Divides data into 800 equal size slabs and sends each slab to 1 of 800 threads.)

THREADS:

THE SECRET:

Did you see The Secret????

THREADS:

```
struct thread_data {
   int start;
   int stop;
};

/****************************************/
/* Thread code.                         */
/****************************************/
void *TEST(void *arg) {
struct thread_data *data;
   data = ((thread_data*)arg);

/****************************************/
/* Main.                                */
/****************************************/
struct thread_data data[threads];
pthread_create(&thread[t], NULL, TEST, (void *)&data[t]);
```

Required for “The Secret”.

We create a pointer to the struct.

Type cast each void pointer to a struct.

We create an array of structs.

Type cast each struct to a void pointer.

THREADS:

THE SECRET:

Do you see The Secret yet????

THREADS:

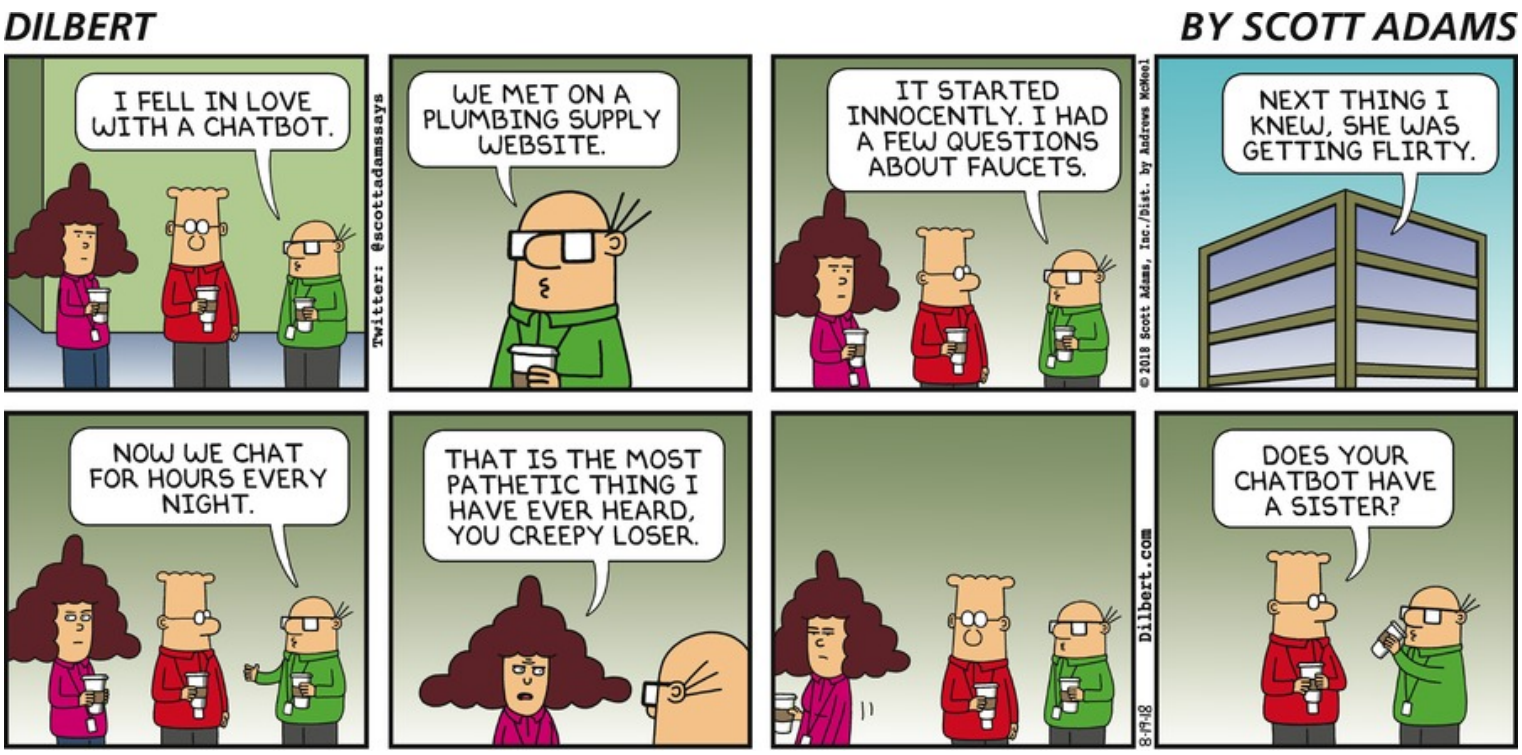
THE SECRET:

Many functions pass void pointers (since the developer wasn't sure what YOU would want to pass).

So, The Secret is to be able to typecast a struct to a void pointer, pass the void pointer, and then typecast the void pointer back to the original struct!

Now that is a Secret worth something.

But, don't tell anyone else – it's a Secret!


DILBERT
BY SCOTT ADAMS
I FELL IN LOVE WITH A CHATBOT.
WE MET ON A PLUMBING SUPPLY WEBSITE.
IT STARTED INNOCENTLY. I HAD A FEW QUESTIONS ABOUT FAUCETS.
NEXT THING I KNEW, SHE WAS GETTING FLIRTY.
NOW WE CHAT FOR HOURS EVERY NIGHT.
THAT IS THE MOST PATHETIC THING I HAVE EVER HEARD, YOU CREEPY LOSER.
DOES YOUR CHATBOT HAVE A SISTER?
Twitter: @scottadamssays
© 2018 Scott Adams, Inc./Dist. by Andrews McMeel
Dilbert.com
8-19-18

6 05 Synchronization

Chapter 5

PROCESS SYNCHRONIZATION:

Processes are concurrent if they exist at the same time (includes processes in the ready queue and the CPU).

Processes are simultaneous (and concurrent) if they are executing at the same time (requires multiple processors).

Concurrent processes can function completely independent of each other, or they can be asynchronous (they require occasional synchronization and cooperation).

A cooperating process is one that can affect or be affected by other processes executing in the system. This may be due to the use of a common data structure or use of a parallel algorithm.

Since we cannot guarantee any particular ordering of the execution of the cooperating processes, the resulting values in the shared resource may be dependent on the order of the execution of the cooperating processes (race condition – a.k.a. lost update in database terms).

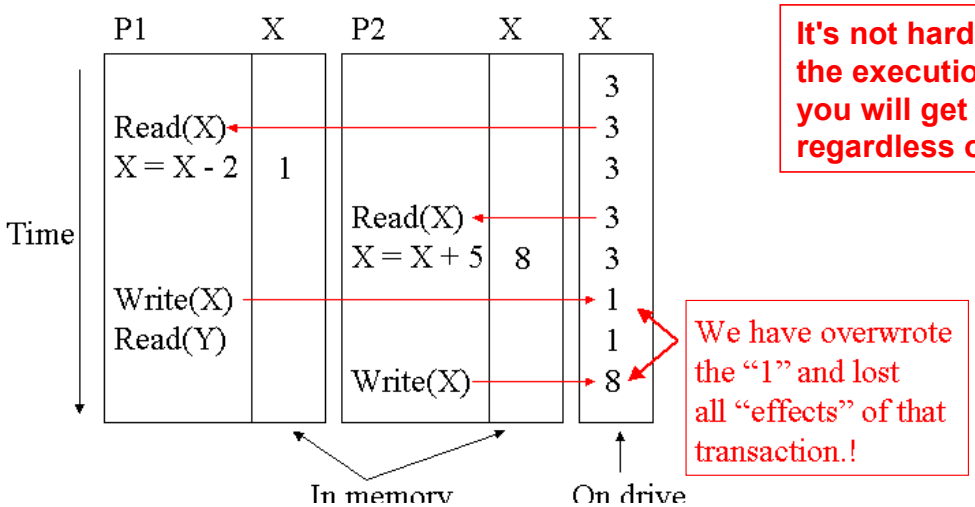
PROCESS SYNCHRONIZATION:

Race Condition:

Race Condition (aka Lost update) occurs when two processes that access the same resource have their operations interleaved in a way that makes the state/value of the shared resource incorrect.

Example: Two processes reading and writing the shared variables X and Y.

P1:	P2:
read(X)	**read(X)**
X = X - N	**X = X + M**
write(X)	**write(X)**
read(Y)	



It's not hard to prove that if you reorder the execution of the two processes that you will get the "correct" result (X = 6, regardless of ordering of P1 and P2).

PROCESS SYNCHRONIZATION:

In some cases, it may be possible to use database serializability methods to insure data consistency. However, a direct application (i.e. 2 phase locking or graph theoretic methods) of this approach would probably add too much overhead to be feasible for this application.

A simpler / more efficient approach is to synchronize the cooperating processes such that they cannot conflict. If they do conflict, we could end up in a state of deadlock. However, if we are not careful our synchronization method may result in starvation.

PROCESS SYNCHRONIZATION:

CRITICAL SECTION:

Any portion of a program where shared data resources are accessed is called a **critical section**. The “other” portions are called the **remainder section(s**).

It is important to note that any given program may have several unrelated critical sections and that these unrelated critical sections may require synchronization with other different processes.

Requirements - Any solution to the critical section problem must satisfy the following:

- **Mutual exclusion** - Only 1 program may be executing in its critical section at any given time. Programs not executing in their critical sections can continue without interference.
- **Progress** – Processes operating outside their critical sections cannot prevent other processes from entering their critical sections.
- **Bounded waiting** – Processes cannot be indefinitely delayed from entering their critical section. If a process terminates while in its critical section, some method must be provided that “releases” the mutual exclusion rights held by the terminated process.
- **Performance** – No assumptions can be made about the relative performance of the involved programs.

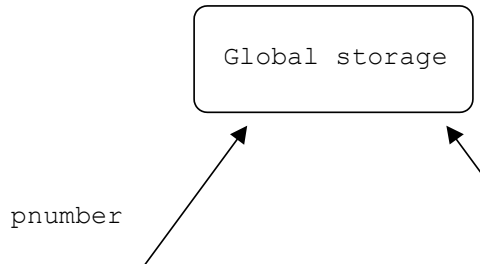
PROCESS SYNCHRONIZATION:

A first effort (lockstep synchronization).

Can only manage 2 processes.

Processes must enter and exit critical sections in strict alternation.

Bounded waiting is not guaranteed. The termination of one causes the remaining to be indefinitely delayed. And, if one process no longer requires execution of its critical section, the other process is indefinitely delayed.




```
Procedure processOne
begin
   while TRUE do begin
      while (pnumber = 2) do;
      // do critical section of 1.
      pnumber := 2;
      // do remainder sections of 1.
   end;
end;
```

```
Procedure processTwo
begin
   while TRUE do begin
      while (pnumber = 1) do;
      // do critical section of 2.
      pnumber := 1;
      // do remainder sections of 2.
   end;
end;
```

Pnumber = 1 initially

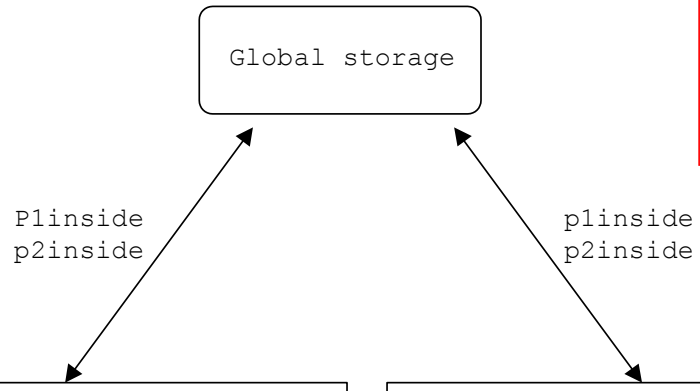
PROCESS SYNCHRONIZATION:

A second effort.

Can only manage 2 processes.

Mutual exclusion is not guaranteed due to concurrent processing .

Bounded waiting is not guaranteed. The termination of one causes the remaining to be indefinitely delayed.




```
Procedure processOne
begin
   while TRUE do begin
      while (p2inside) do;
      p1inside := TRUE;
      // do critical section of 1.
      p1inside := FALSE;
      // do remainder sections of 1.
   end;
end;
```

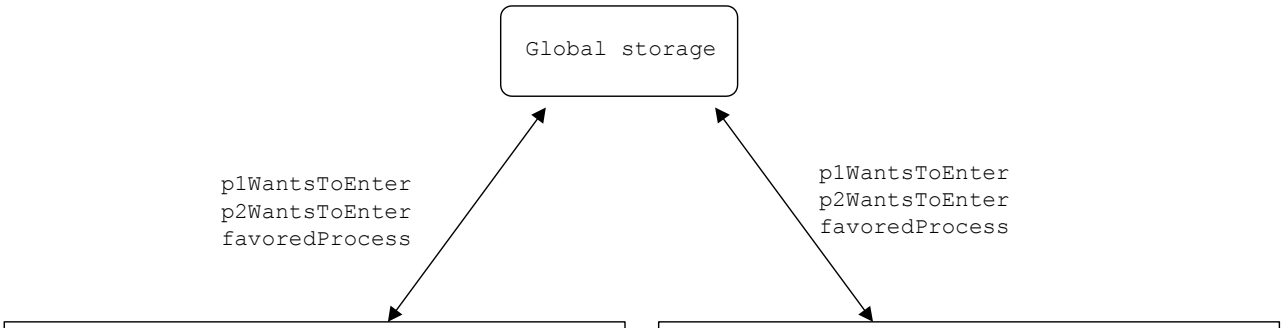
```
Procedure processTwo
begin
   while TRUE do begin
      while (p1inside) do;
      p2inside := TRUE;
      // do critical section of 2.
      p2inside := FALSE;
      // do remainder sections of 2.
   end;
end;
```

P1inside = false initially
p2inside = false initially

PROCESS SYNCHRONIZATION:

Dekker's algorithm.

Can only manage 2 processes.




```
Procedure processOne
begin
   while TRUE do begin
      p1WantsToEnter := TRUE;
      while (p2WantsToEnter) do begin
         if (favoredProcess = 2) then begin
            p1WantsToEnter := FALSE;
            while (favoredProcess = 2) do;
            p1WantsToEnter := TRUE;
         end;
      end;
      // do critical section of 1.
      favoredProcess := 2;
      p1WantsToEnter := FALSE;
      // do remainder sections of 1.
   end;
end;
```

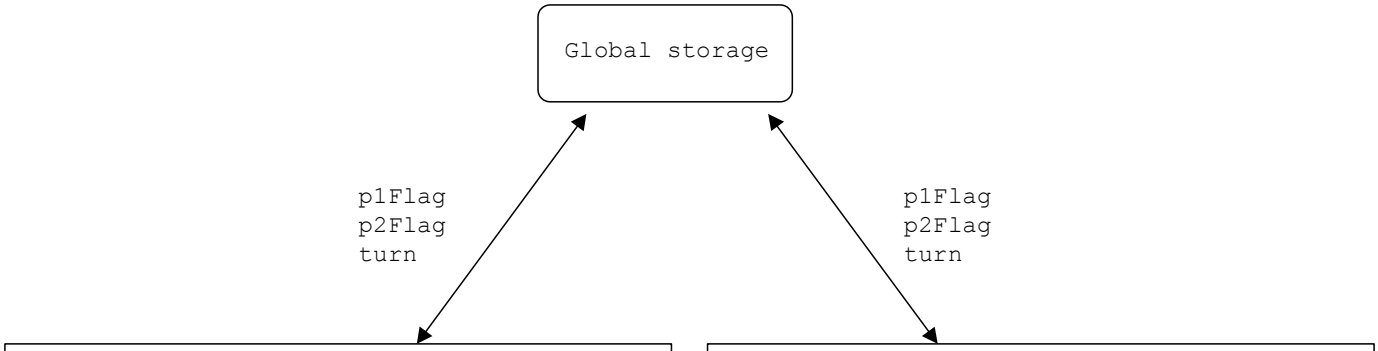
```
Procedure processTwo
begin
   while TRUE do begin
      p2WantsToEnter := TRUE;
      while (p1WantsToEnter) do begin
         if (favoredProcess = 1) then begin
            p2WantsToEnter := FALSE;
            while (favoredProcess = 1) do;
            p2WantsToEnter := TRUE;
         end;
      end;
      // do critical section of 2.
      favoredProcess := 1;
      p2WantsToEnter := FALSE;
      // do remainder sections of 2.
   end;
end;
```

```
p1WantsToEnter := FALSE initially.
p2WantsToEnter := FALSE initially.
favoredProcess := 1 initially.
```

PROCESS SYNCHRONIZATION:

Peterson's algorithm.

Can only manage 2 processes.




```
Procedure processOne
begin
   while TRUE do begin
      p1Flag := TRUE;
      turn := 2;
      while ((p2Flag) and (turn = 2)) do;
      // do critical section of 1.
      p1Flag := FALSE;
      // do remainder sections of 1.
   end;
end;
```

```
Procedure processTwo
begin
   while TRUE do begin
      p2Flag := TRUE;
      turn := 1;
      while ((p1Flag) and (turn = 1)) do;
      // do critical section of 2.
      p2Flag := FALSE;
      // do remainder sections of 2.
   end;
end;
```

```
p1Flag := FALSE initially.
p2Flag := FALSE initially.
turn := 2 initially.
```

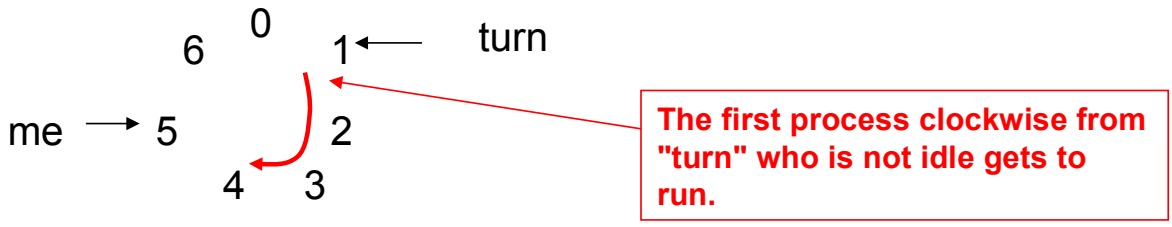
PROCESS SYNCHRONIZATION:

N-PROCESS CRITICAL SECTION:

Dijkstra – The first (1965) to present a software solution for implementation of an n-process mutual exclusion. However, his method allowed the possibility of indefinite postponement.

Knuth – Presented (1966) a software solution that eliminated the possibility of indefinite postponement. However, his method not prevent the possibility of lengthy delays.

Eisenberg and McGuire – Presented (1972) a solution for the Critical Section Problem for N Processes. The algorothm organizes the processes into a circle:



Lamport – Presented the Bakery (Baker's) algorithm (1974). The algorithm uses the "take a ticket" method commonly found in bakeries, service counters, etc.

PROCESS SYNCHRONIZATION:

INTERRUPT DISABLING:

On a **uniprocessor** system, since only one program can be executing at a time, we could disable interrupts guaranteeing mutual exclusion. Why?

SPECIAL MACHINE INSTRUCTIONS:

A variety of multiple machine instruction pairs (which execute as a single atomic "instruction") have been proposed to manage critical sections, including:

1. Set & test
2. Compare & swap
3. Exchange

Using any of the above, we can easily modify the "second effort" algorithm to provide a good, but not perfect, solution (indefinite postponement is slightly possible).

PROCESS SYNCHRONIZATION:

SEMAPHORES:

Critical section solutions are not easily generalized to more complex synchronization problems. Semaphores (Dijkstra 1965) overcome this limitation.

A semaphore is an integer variable that is accessed through 2 **atomic** operations:

```
Wait:        while (S <= 0) do; S--;
Signal:      S++;
```

The above is an example of a spinlock semaphore. A semaphore type that wastes CPU cycles spinning in the while-loop.

Binary semaphore – Can be 1 or 0.

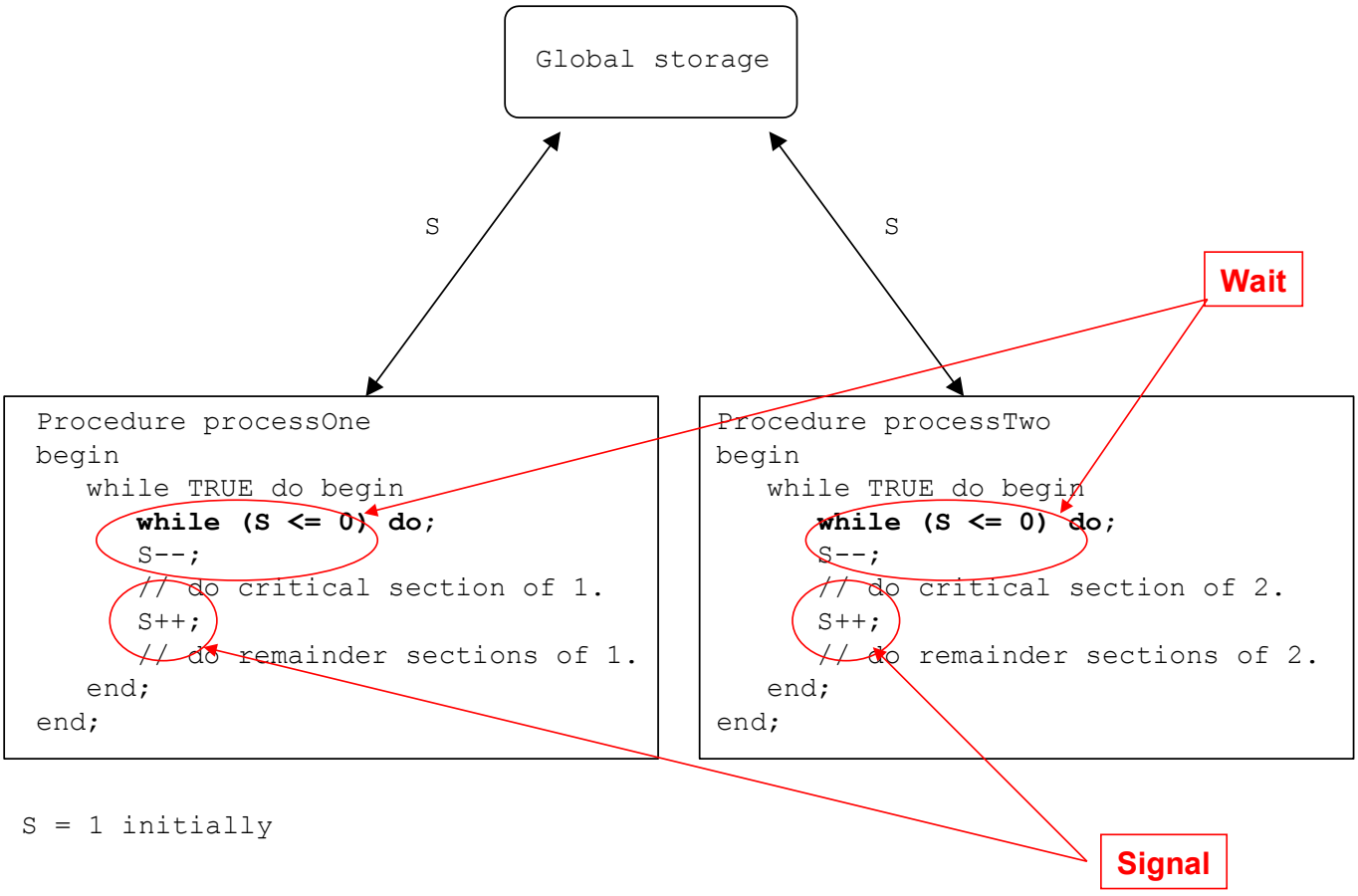
Counting Semaphore – Can be any integer value. Useful for allocating / sharing N identical resources to M concurrent processes (N <= M).

Semaphores can be used to manage critical section problems and can be used to force a specific execution sequence of a set of processes.

Incorrect use of semaphores (or any mutual exclusion method) can result in 2 processes being in their critical sections at the same time.

PROCESS SYNCHRONIZATION:

Lockstep synchronization modified with binary semaphores .



PROCESS SYNCHRONIZATION:

PRODUCER - CONSUMER:

Producer – produces goods, one at a time, and passes the item to the consumer.

Consumer – receives goods, one at a time, from a producer.

If their relative operating speeds are nicely matched – no problem. But, we cannot guarantee this. Therefore, we need either:

1. A message passing method (i.e. handshaking).
2. A fixed capacity storage facility (with message passing).
3. An infinite capacity storage facility (message passing not needed).

Approach 1 can be achieved using any mutual exclusion method. Even our first approach will, sort of, work.

Approach 2 can be achieved using any mutual exclusion method. Even our first approach will, sort of, work.

Approach 3 is not feasible.

PROCESS SYNCHRONIZATION:

MONITORS:

Mutual exclusion methods and semaphores work, but they have a number of weaknesses. Including:

- It is difficult to use to express solutions to more complex concurrency control problems.
- It is difficult to prove the correctness of a program using these methods.
- It is easy to misuse these methods.

Monitors (Dijkstra 1971, Hoare 1974, and Hansen 1975) are a higher-level construct that solves many of the problems associated with earlier mutual exclusion methods and semaphores. Characteristics of monitors include:

- Monitors are a concurrency construct that contains both the data and procedures required to perform the allocation of a shared resource.
- Monitors enforce information hiding.
- Monitors rigidly enforce mutual exclusion (at the monitor boundaries).
- Monitor processes waiting for a resource wait in a queue (free up CPU) until signaled by another process that the resource is available.
- Monitor processes waiting have priority over any new processes.
- Monitors are a programming language construct, so it is up to the compiler to enforce the mutual exclusion. Therefore, it is less likely that something will go wrong (we assume that the compiler is correct).

PROCESS SYNCHRONIZATION:

HOARE VS HANSEN MONITOR:

The difference between Hoare's and Hansen's approaches can be described as follows: Consider a producer-consumer n-buffer process pair. The producer finds the buffer full and does a wait. The consumer consumes one item and signals the producer to "wakeup". We now need some method to prevent the two processes from being in the monitor at the same time.

Hoare proposed that the awakened process (the producer) suspend the consumer process.

Hansen proposed that the process sending the "wakeup" call exit the monitor immediately (the wakeup call must be the last statement in the monitor process).

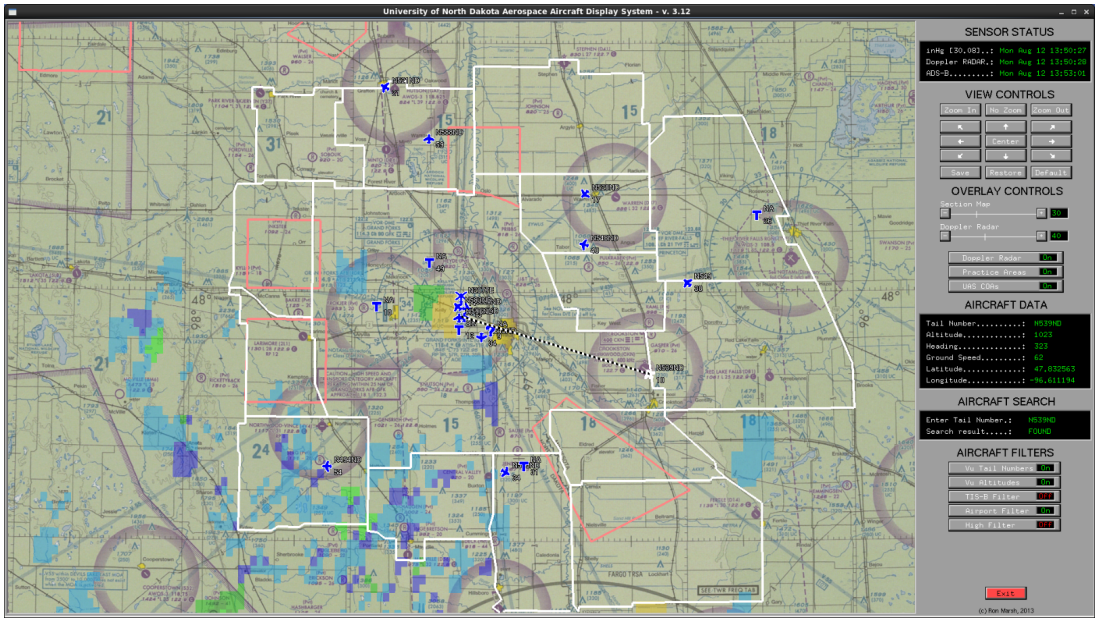
Hoare's method may be more powerful, but Hansen's is simpler to understand and implement.


ARE YOU MARRIED?
I DON'T HAVE A WIFE, BUT I DO HAVE WIFI.
Dilbert.com DilbertCartoonist@gmail.com
I FIND THAT IT MEETS ALL OF MY SOCIAL AND INTELLECTUAL NEEDS.
8-6-13 ©2013 Scott Adams, Inc./Dist. by Universal Uclick
DO YOU MISS THE WARMTH OF HUMAN CONTACT?
NEVER TRIED IT.
SOUNDS PROBLEM-ATIC.

7 05a Mutex case study

MUTEX CASE STUDY:

- The Aerospace Airport Display System (AADS) would acquire ADS-B position data for all ADS-B equi display that information on the screen (as an Information Display System – IDS). The display backgr aviation sectional chart.
- The user could zoom, pan, scroll, and adjust the opacity of the display. The user could also search f
- The user could also toggle the display to show UND practice and UAS areas, the Doppler RADAR, ar data (tail number and altitude)



MUTEX CASE STUDY:

The AADS relied on 3 data sources to keep the display up to date.

1. The AADS would acquire ADS-B data from a Garmin ADS-B receiver (via a socket connection) once

2. The AADS would acquire weather data from a webpage (via wget) and was downloaded every 7 m
   data rate – Nyquist sampling criteria).

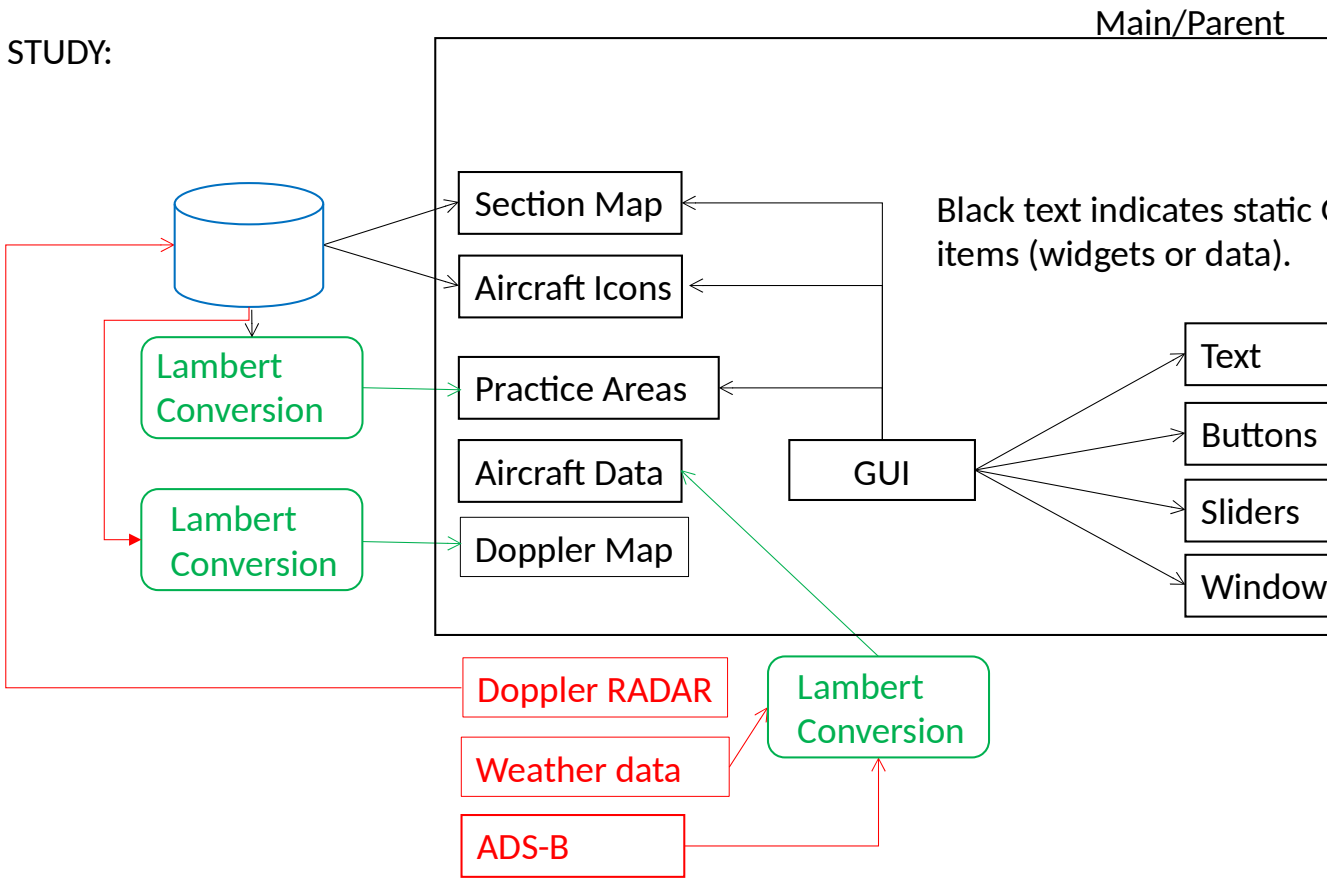
3. The AADS would acquire Doppler RADAR from another webpage (via wget) and was downloaded e
   (twice the data rate – Nyquist sampling criteria).

As the display was an IDS, we expect that the user would make changes to the display (zoom, pan, scr
system was running. As such, one could argue that we have 4 concurrent (possibly simultaneous) proc

We also had to consider that the system was part of the Aerospace Safety Management system and h
(could not fail).

The solution was to use threads as they would provide a more robust system (if a thread failed we wo
without restarting the entire system – in fact, we spawned threads when needed and killed them imm
would provide for good interactivity (e.g. the user could zoom while the system was downloading a ne
map).

MUTEX CASE STUDY:



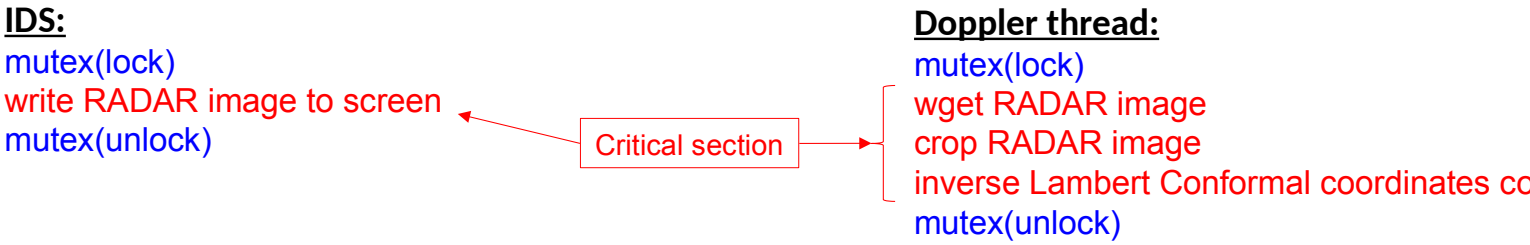
Red text indicates a thread started by GUI.
Each thread has an associated mutex.

MUTEX CASE STUDY:

The most complex component was the Doppler RADAR system:

For each new radar image we have to download it from a webpage, then crop it to fit our display, ther
scale, and then convert the image from Cartesian coordinates to inverse Lambert Conformal coordinat
aliasing). This took several minutes, but we then had a new Doppler RADAR image layer to replace the
RADAR image layer on the display.

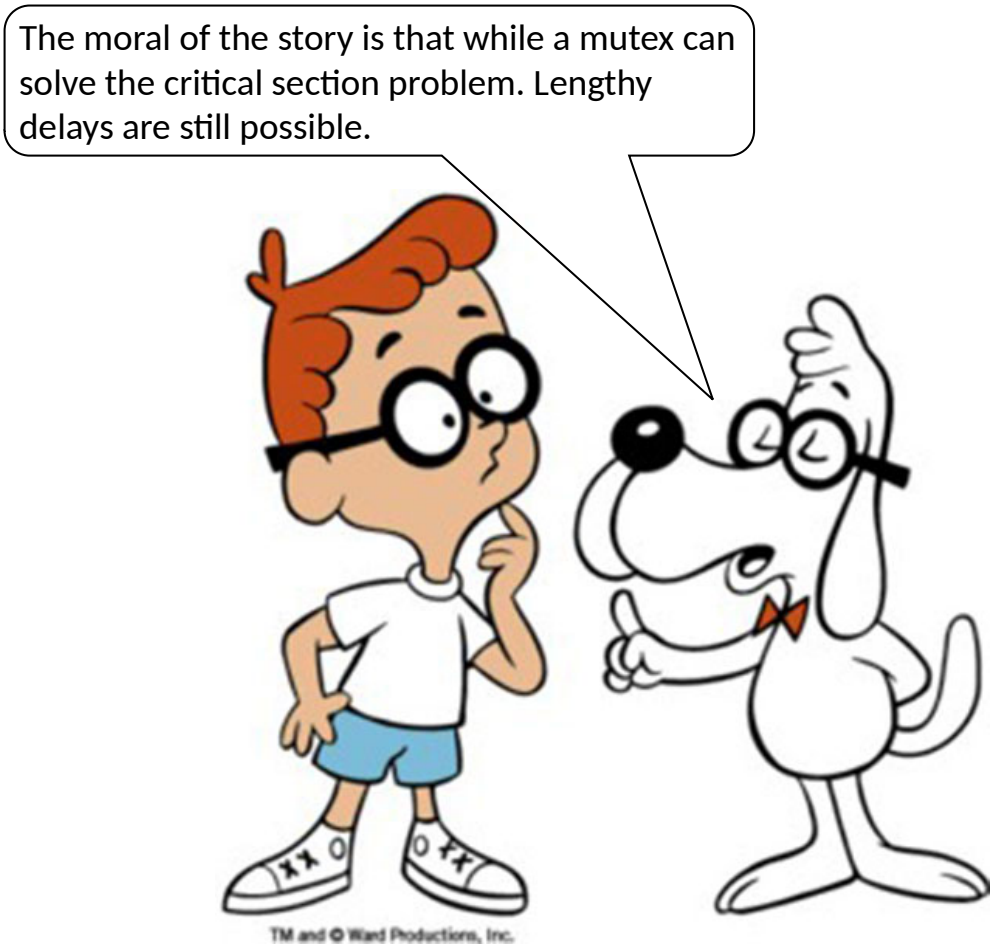
So, the algorithm could be:



The drawback is that if the user was doing a lot of GUI activity the RADAR image would have to consta
the screen, and:

a) The Doppler thread could be repeatedly blocked.
b) The GUI (Main/parent) would be blocked (e.i. entire GUI would freeze).

MUTEX CASE STUDY:



MUTEX CASE STUDY:

So, the revised algorithm was:

We start be defining two pointers to the image data type and allocate memory to them:

```
unsigned char **radar1, **radar2;
radar1 = (unsigned char **)calloc(RADAR_image_height, sizeof(*unsigned char));
radar2 = (unsigned char **)calloc(RADAR_image_height, sizeof(*unsigned char));
for (y = 0; y < RADAR_image_height; y++) {
    radar1[y] = (unsigned char *)calloc(RADAR_image_width, unsigned char);
    radar2[y] = (unsigned char *)calloc(RADAR_image_width, unsigned char);
}
```

We now use the radar1 array to display the RADAR image.

IDS:

```
mutex(lock)
write RADAR image (radar1) to screen
mutex(unlock)
```

MUTEX CASE STUDY:

The revised algorithm (cont):

We now use radar2 array for the initial RADAR processing.

Doppler thread:

wget RADAR image
crop RADAR image
inverse Lambert Conformal coordinates conversion
mutex(lock)
swap what pointers radar1 and radar2 point to
mutex(unlock)

All on radar2

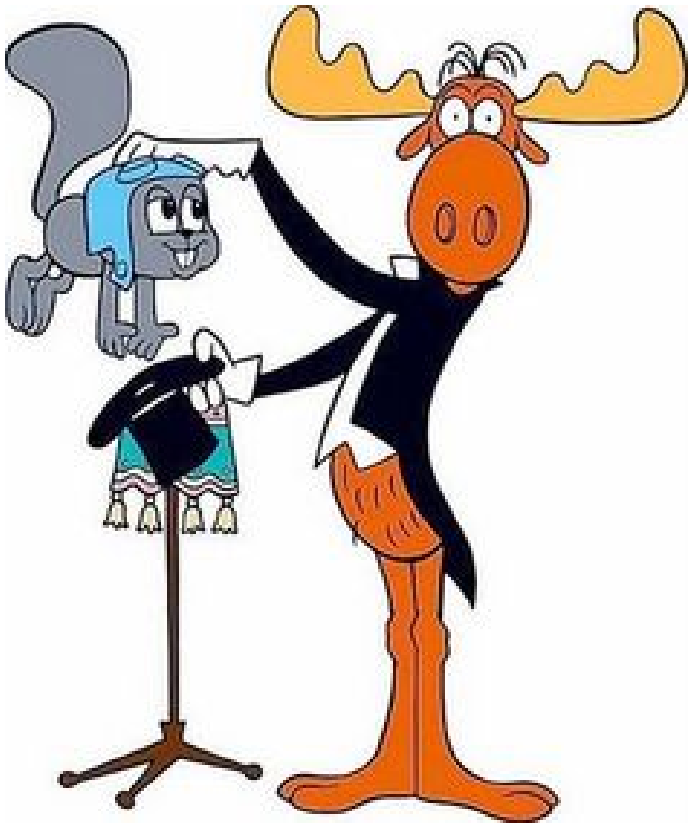
Critical section

Since, we now have 2 separate arrays, this version can move the previous long running critical section remainder section.

The new critical section is MUCH faster and resulted in no perceivable delay in the system.

MUTEX CASE STUDY:

The method we used to improve system performance is called “double buffering” and while not quite
a Rocky out of a hat. Double buffering definitely is a technique used to move long running code out of
section.




ACCORDING TO MY ALGORITHM, WE ARE HEADING TOWARD A PARODY INVERSION POINT.
DILBERT.COM @SCOTTADAMSSAYS
THAT HAPPENS WHEN REALITY BECOMES SO ABSURD THAT IT IS INDISTINGUISHABLE FROM PARODY.
10-2-19 2019 Scott Adams, Inc./Dist. by Andrews McMeel
MAYBE THE GOV— ERNMENT CAN FIX THAT.
GAAA!!! I CAN'T TELL IF YOU'RE SERIOUS

8 06 CPU Scheduling

Chapter 6

UNI-CPU SCHEDULING:

CRITERIA:

Many different scheduling criteria have been proposed. A few include:

- CPU utilization – We want to keep the CPU as busy as possible.
- Throughput – We want to maximize the number of processes being completed per unit time.
- Turnaround time – We want to minimize the time it takes for a job to be completed.
- Wait-time – We want to minimize the time the process spends in the ready queue.
- Response time – We want to minimize the time between input of a command and the response.
- Variance – We want to minimize the variance in the response time.

We would like to achieve all of the above; that is seldom possible. So we settle for some average value. Furthermore, some of these may be obsolete. For example, does it make any sense to use CPU utilization as a scheduling criteria?

UNI-CPU SCHEDULING:

OBJECTIVES:

Scheduling objectives include:

- Should be fair to all processes.
- Should attempt to service the maximum number of processes per unit time.
- Maximize number of interactive users receiving acceptable response times.
- Predictable.
- Minimize overhead.
- Balance resource use.
- Enforce priorities.
- Give preference to any process holding key resources.
- Avoid indefinite postponement - Indefinite postponement can be as bad as deadlock.
- Give priority to well behaved processes.
- Graceful degradation.

Again, we would like to achieve all of the above. However, that is seldom possible.

UNI-CPU SCHEDULING:

TYPES:

1. Non-preemptive – Once a process has been given the CPU the CPU cannot be taken away from that process. However, the process can, but doesn't have to, release the CPU while performing I/O.

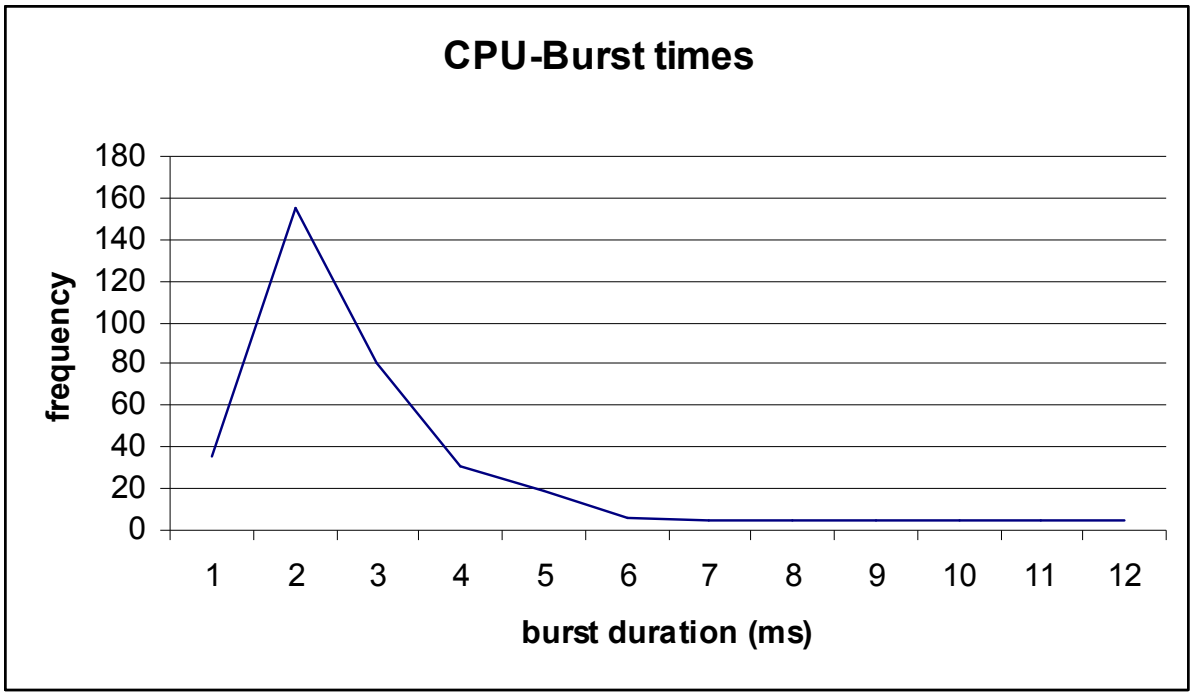
2. Preemptive – The CPU can be taken away by the OS. Characteristics of preemptive scheduling include:
    - Useful in systems where high-priority processes require rapid attention.
    - Required in interactive systems.
    - Preemption incurs overhead (context switching, memory required to hold jobs waiting in ready queue, mechanisms to protect resources held by a waiting process, and CPU cycles lost to the scheduler).

UNI-CPU SCHEDULING:

CPU I/O BURST TIMES:

CPU-burst times can be of use to determine a “good” time slice criteria. Some statistical methods include using the peak height, half peak height, and the peak base height.

I/O-burst times may also be of use.



UNI-CPU SCHEDULING:

METHODS:

First-Come First-Served (FCFS). Characteristics include:

- Simplest (simply a FIFO queue).
- Wide variation in waiting times.
- Non-preemptive – which can cause a convoy effect.
- Not applicable to time-shared or interactive systems.

Shortest-Job-First (SJF) - The job with the shortest next CPU-burst cycle is run first. Characteristics include:

- Optimal in reducing the waiting time.
- Difficult to determine shortest next CPU-burst cycle – the next CPU-burst cycle can be estimated using a moving average (exponential or linear).
- Non-preemptive – the current job is allowed to finish, even if a (new) job with a shorter next CPU-burst cycle arrives at the ready queue.

Shortest-Remaining time – Almost identical to the Shortest-Job-First method. Characteristics include:

- Preemptive – the current job is preempted if a (new) job with a shorter next CPU-burst cycle arrives at the ready queue.

UNI-CPU SCHEDULING:

METHODS:

Priority – Jobs are run based on priority. Characteristics include:

- Job priority may be static or dynamic.
- Job priority may be determined internally (resource requirements) or externally (company policies).
- Can be preemptive or non-preemptive.
- Can result in low-priority jobs being starved.

Deadline – Jobs that are required to complete by a specific time are given priority. Characteristics include:

- Complex – Resource requirements need to be accurately specified by the programmer.
- Multiple deadline jobs can create problems/resource conflicts.

Round-Robin (RR) – Processes are run for a maximum of 1 time-quantum. They are then preempted and place on the tail of the ready queue. Characteristics include:

- Designed for time-shared systems.
- Similar to FCFS, but preemption is added to switch between processes.
- A process may use less than the time-quantum if I/O is required.
- Average wait time is long, but average response time is short.

UNI-CPU SCHEDULING:

METHODS:

Multi-level – A method that provides different levels of priority for different groups of users / job types. Characteristics include:

- Each level may have a separate queue and may use a different scheduling method.
- A “master” scheduler allocates system resources to each queue. It is common to use a priority method between the queues (no lower priority queued job runs until all higher priority queued jobs are completed) or a Round-Robin method between the queues (the higher priority queued jobs get more or longer time quantum than the lower priority queued jobs).

Multi-level feedback – Similar to the Multi-level, except:

- Jobs can migrate up to a higher priority queue as they age. Or can migrate down to a lower queue if they require too much CPU time.

Highest-Response-Ratio-Next (HRN) – Corrects some of the weaknesses of the Shortest-Job-First methods:

- Non-preemptive.
- Priority determined by: $$priority = \frac{(time\_waiting + service\_time)}{service\_time}$$

UNI-CPU SCHEDULING:

METHODS:

Linux - This algorithm is a credit based algorithm. Each process is given a certain number of credits. When a new task is to be run, the process with the most credits runs. As a process runs, every time a timer interrupt occurs, the process losses one credit. When its credit count reaches 0, it is suspended and another process is chosen to run.

If no runnable processes have any credits, Linux re-credits all (even the non-runnable ones) of the processes according to:

credits = (held_credits / 2) + priority

This method favors processes with high priority and processes that have not run in a long time, but are I/O bound (non I/O bound processes are not favored).

UNI-CPU SCHEDULING:

SCHEDULING EVALUATION:

Deterministic modeling – Uses a predetermined workload to determine system characteristics. The method is simple and fast and gives exact numbers for the cases tried, but is only useful for demonstrative purposes (unless you run 100's of cases, but, then you have a simulation).

Queueing models – Use queuing statistical models. The method is useful in comparing scheduling methods. However, the classes of models that can be handled are limited and the math is too complicated to develop a new model. So assumptions are made to simplify the math.

Implementation – Implementation of the scheduling method on a real system. The method is probably the worst evaluation method possible as it requires a working version of the scheduler in code. Is dangerous, as it may have bugs in it. Is frustrating to the users and the testers as the system will probably behave differently.

If the system behaves differently, the users will (a known fact) adjust the ways in which they use the system.

UNI-CPU SCHEDULING:

SCHEDULING EVALUATION:

Simulations – Simulation of the scheduler in software is, perhaps, the best approach. The method is fairly easy to implement (software packages exist for this purpose) and provides statistics. All of the pertinent events (process arrival, size, type - CPU driven, I/O driven, etc) can all be randomly generated (using different distributions) and measurements taken from the actual system may be used to tailor the distribution of events.

The quality of the simulation is affected by the quality of the random number generator.

Simulations can be expensive to run (CPU time) as a good simulation will require many different distributions of data.

Multi-CPU SCHEDULING:

When multiprocessing (or parallel processing) is employed several additional issues become apparent…

TYPES:

Types of multiprocessor systems:

Clusters – Just a bunch of computers connected by a network.

Multiprocessor systems – Machines that have multiple processors and/or mulit-core.

Specialized processors – Like a common computer. Many task-specific processors working towards a common goal.

Multi-CPU SCHEDULING:

DESIGN ISSUES:

Granularity – the frequency by which the individual processes must communicate between themselves:

- Independent processes - do not need to communicate with each other.

- Dependent processes - do need to communicate with each other and the frequency of required communication drives the hardware/software design. Tightly coupled (fine grained) processes raise many problems.

Process / Processor assignment – two major classes of parallel design:

1. Master/slave - environments have a master program run on 1 of the processors with the slave processes run on the other processors.

2. Peer - environments do not have a “master” and any process can run on any processor.

Multi-CPU SCHEDULING:

METHODS:

Process / Processor scheduling has four major scheduling methods:

1. Load sharing – Processes are sent to the current idle processor (aka load balancing). With large numbers of processors accessing the ready-queue (a not shared resource) a bottleneck can result and processes that were pre-empted may not get to run on the processor that they started with, hence, their cached data may have to be “moved.”

2. Gang scheduling – tightly coupled processes are run in parallel to make it efficient for them to exchange data or to synchronize.

3. Dedicated assignment – dedicates of a set of processes to a set of processors.

4. Dynamic scheduling – allows the OS and software to dynamically change the scheduling method as required.

The above methods can be used with threads on single processor machines to improve performance as well.

REAL-TIME OS:

In many of today's autonomous systems, the system is required to behave in a known and timely manner. Traffic lights are expected to change on set intervals and be correct. Airline flight control systems are expected to respond to the pilot's inputs, etc, etc, etc.

TYPES:

Four types of real-time systems:

1. Hard – where the tasks must be executed in a timely manner.

2. Soft - where the tasks should be executed in a timely manner.

3. Aperiodic tasks – have a deadline by which they must start or stop.

4. Periodic tasks – must perform something on a set periodic schedule.

REAL-TIME OS:

CHARACTERISTICS:

Real-time OS characteristics:

- Determinism – the temporal characteristics of the system must be known in advance and must be such that we can count on those characteristics. Of special interest is “How long will it take the system to determine that an interrupt has occurred?”.

- Responsiveness - of special interest is “How long will it take the system to service the interrupt once it has occurred?”.

- User control – a real-time OS must allow the user to easily select / adjust OS parameters such as task priority.

- Reliability – many real-time systems are autonomous and / or are designed to insure human lives and must not fail.

- Fail-soft operation - many real-time systems are autonomous and must be able to repair themselves or adjust their parameters (or workload) in an attempt to not fail. Some systems are designed to “catch” a failure and simply restart the system (street lights).

REAL-TIME OS:

ALGORITHMS:

Real-time scheduling algorithms:

- Static table-driven – Applicable to periodic tasks where the run-time and execution times are known and static.

- Static priority-driven – When run-times are known but priorities are flexible we use a typical multitasking priority scheduler.

- Dynamic planning-based – If a new processes is added, the old schedule is examined to see if the new process can be fit (without messing up the existing run-times and execution times).

- Dynamic best-effort – Applicable to aperiodic tasks where the priority of new tasks is determined as they are added.

REAL-TIME OS:

CATEGORIES:

Real-time scheduling categories:

- Deadline – Attempts to meet/guarantee some deadline (task start, stop, etc) using priority and timing data in order to insure the timely execution of tasks.
- Rate monotonic – Prioritizes tasks according to run-time (fast first, etc).
- Priority Inversion – Occurs when a lower-priority task gets to run before a higher priority task due to some unexpected circumstance.

EMBEDDED OSs:

Embedded systems comprise a mix of hardware and software dedicated to a specific purpose (task).

EXAMPLE USES:

- mobile phones,
- videogame consoles,
- digital cameras,
- DVD players,
- GPS receivers,
- printers,
- microwave ovens,
- washing machines,
- dishwashers,
- digital watches,
- MP3 players,
- traffic lights,
- factory controllers,
- nuclear power plants.

EMBEDDED OSs:

CHARACTERISTICS:

- Real-time – whether soft or hard depends on application's requirements.
- Reactive – it is common for these systems to react to (or be required to react) to inputs from sensors.
- Configurability – the embedded OS should be configurable (many types of embedded systems exist).
- I/O flexibility – same concern as with configurability.
- Streamlined protection – due to the limited scope of operation, sophisticated protection is rarely required.
- Direct use of interrupts – due to the limited scope of operation, we can allow the software to make direct use of the interrupts.

EMBEDDED OSs:

EXAMPLES:

A/ROSE
BeRTOS
DSPnano RTOS
Embedded Linux
FreeBSD
FreeRTOS
Inferno (distributed OS, Bell Labs)
LynxOS
Psos
MontaVista Software
Nucleus RTOS
CMX
uC/OS
eCos RTEMS
MINIX 3
.NET Micro Framework
OS/RT
Operating System
Embedded (OSE)
Open AT OS
polyBSD (embedded NetBSD)
QNX
RTXC Quadros
ROM-DOS
T2 SDE
TinyOS
Unison RTOS
µTasker
VxWorks
RTLinux
Windows XP Embedded
Windows CE
Android
IOS
Windows 8

EMBEDDED OSs:

ECOS:

ECOS - eCos is an open source, royalty-free, real-time operating system intended for embedded applications. The eCos distribution is available in both Linux and Windows versions.

Architectures supported:

ARM, CalmRISC, FR-V, FR30, H8, IA32, 68K/ColdFire, Matsushita AM3x, MIPS, NEC V8xx, PowerPC, SPARC, SuperH

Devices supported:

Flash devices, Ethernet devices, Serial devices, USB devices, Timekeeping devices

http://ecos.sourceware.org/

EMBEDDED OSs:

TINYOS:

TinyOS is an event based operating environment designed for use with embedded networked sensors. It is designed to support the concurrency intensive operations required by networked sensors with minimal hardware requirements.

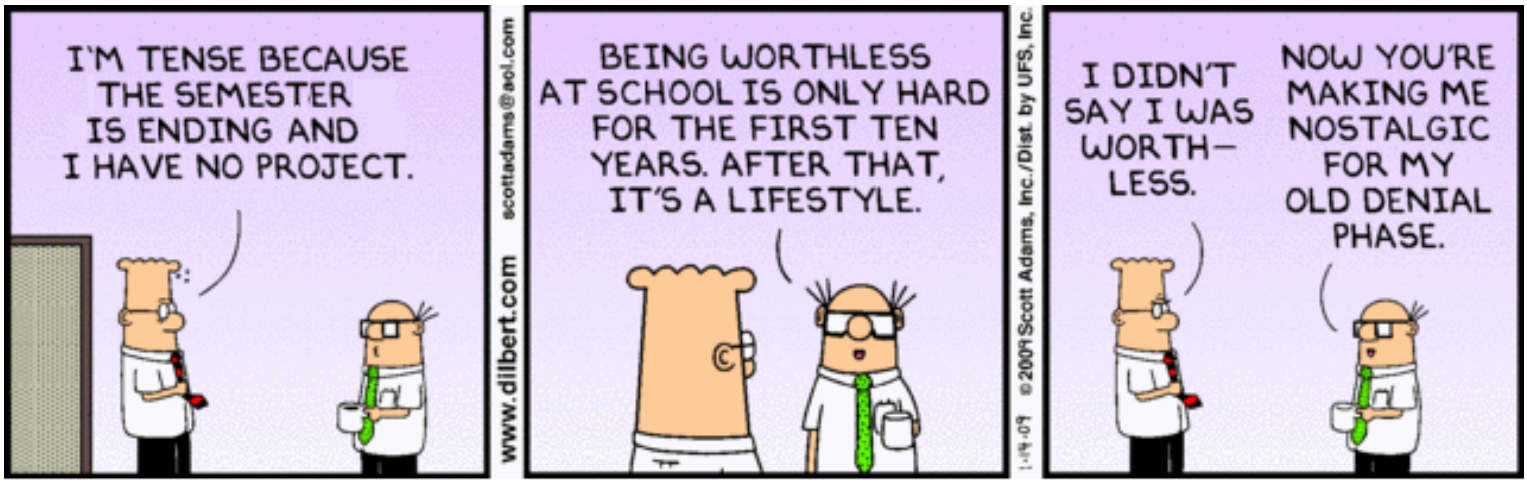
There are hundreds of TinyOS projects throughout the world (http://webs.cs.berkeley.edu/tos/related.html)

The programming language of TinyOS is stylized C that uses a custom compiler 'NesC'.

TinyOS was initially developed by the U.C. Berkeley EECS Department..

TinyOS has been ported to over a dozen platforms and numerous sensor boards.

http://webs.cs.berkeley.edu/tos/


I'M TENSE BECAUSE THE SEMESTER IS ENDING AND I HAVE NO PROJECT.
www.dilbert.com scottadams@aol.com
BEING WORTHLESS AT SCHOOL IS ONLY HARD FOR THE FIRST TEN YEARS. AFTER THAT, IT'S A LIFESTYLE.
1-19-09 ©2009 Scott Adams, Inc./Dist. by UFS, Inc.
I DIDN'T SAY I WAS WORTH-LESS.
NOW YOU'RE MAKING ME NOSTALGIC FOR MY OLD DENIAL PHASE.

9 07 Deadlock

Chapter 7

DEADLOCK:

Resource sharing is one of the primary goals in multi-programmed computer systems. Whenever resources are shared among a population of users and whenever those users have exclusive control over resources allocated to them, deadlock can occur.

A process is in a state of deadlock if it is waiting for an event that will not occur. (A set of processes are in a state of deadlock when every process in the set is waiting for an event that can be caused only by another process in the set.).

We assume that all processes abide by the same rules with regards to resources. These rules, in order of use, are:

1. Request – Processes request the use of a resource. If that resource is not available, the process waits until it is.
2. Use – Once a resource has been made available to a process, the process can make use of it.
3. Release – Processes release resources no longer required.

DEADLOCK:

Four simultaneous conditions are necessary for deadlock:

1. Mutual exclusion – At least 1 resource must be held in a non-sharable mode.

2. Hold and wait – There must be a process holding a resource and that is waiting to acquire another resource held by some other process.

3. No preemption – Resources cannot be preempted (resources are used to completion).

4. Circular wait – There must exist a set of waiting processes (general case of hold and wait).

DEADLOCK:

EXAMPLE – A print spooler:

- Disk space allocated for the spooling is fixed.
- No spool file can be printed until it is completely written into the spool disk.
- No spool file can be removed until the printer has completed printing the file.
- If the allocated disk space is exceeded, spool files can no longer be appended to or new ones created.
- In a multi-user system, it is possible that multiple spool files will be created concurrently and that these (partial) spool files will exceed the allocated space. Since we cannot remove any spool files until they have been processed and since we cannot process a spool file until it has been written completely, we become deadlocked.

DEADLOCK

PREVENTION:

Havender (1968) concluded that if any of the four necessary conditions are denied, deadlock is impossible. Methods to deny the conditions include:

1. Allow sharable resources (i.e. read-only files).
2. Deny the hold and wait condition by requiring that a process request and be granted all of its required resources before it can begin execution. Another approach is to require processes to release any currently held resources before any more are granted.
3. Deny the no-preemption condition by requiring that whenever a process is denied a resource request that it also release all currently held resources. The process may then request them again, if necessary.
4. Deny the circular wait condition by assigning unique numbers to all resources and by requiring that all processes request the resources in ascending order. This strategy has been implemented, however:
   - Resource numbers are assigned during the installation. If new resources are added, existing programs may have to be rewritten.
   - Resource numbers are assigned assuming some typical sequence of use.
   - The use of resource numbers does not agree with today's "free spirited" software design goals.

DEADLOCK

AVOIDANCE:

Algorithms for avoiding deadlock include:

- The Ostrich algorithm - Stick your head in the sand and pretend there is no problem (UNIX).

- Resource-Allocation Graph algorithm - If we have only 1 instance of each resource type, we can apply graph theory for deadlock avoidance. Any cycle in the graph indicates an unsafe state.

- Dijkstra's Banker's algorithm (1965) - Assume that we have 2 customers who have been given a line of credit. Initially none of the lines have been charged to. The banker knows that all of his customers will not need the full line of credit at any given time. So he only allocates 10 "chunks" of money to the credit pool.

Name	Used	Credit	
Andy	0	6	
Suzy	0	7	Available: 10 (safe)

The bank is in a safe state if there exists a sequence of states that leads to all customers getting loans up to their credit limit (we also assume that they pay the loans back).

DEADLOCK

DETECTION:

A popular notion is to use resource allocation graphs to detect deadlock. However, since these algorithms incur overhead, the question is “How often should we invoke the algorithm? Possible answers include:

- Every time a resource request is made and refused.
- At fixed time intervals.
- When CPU usage is low (deadlocks eventually cripple the system).

Reduction of resource allocation graphs – Whenever a process’s resource requests are granted, the corresponding edges in the resource allocation graph are removed.

DEADLOCK

RECOVERY:

Deadlock must be removed from the system.

Deadlock recovery is difficult because of:

1. It may not be clear that the system is in a state of deadlock.
2. Most systems have poor facilities for suspending a process, removing it from the system, and resuming it at a later time.
3. Deadlock recovery facilities incur overhead and typically require a skilled operator to manage them.
4. Recovering from deadlock of modest proportions requires work. Recovering from deadlock on a grand scale requires an enormous amount of work.

DEADLOCK

RECOVERY:

Deadlock recovery generally has four options:

1. Abort all deadlocked processes.
2. Abort 1 process at a time until the deadlock is broken.
3. Rollback deadlocked processes to some “safe” state.
4. Preempt 1 resource at a time until the deadlock is broken.

However, 3 and 4 require some effective suspend/resume mechanism (like databases). However, to date, few operating systems include this type of mechanism (suspend/resume and checkpoint/restart).

However, aborting processes has problems also. For example:

- The deadlocked processes may all have the same priority.
- The priorities may be muddled by special considerations.

If we could choose preemption, we still have issues:

- How do we select a victim?
- What do we do with the preempted process?
- How do we guarantee that the same process will not always be preempted?

STARVATION

Starvation (aka Indefinite postponement) occurs when a process is waiting for an event that may never occur because of biases (or poor design?) of the scheduling method.

In deadlock, the problem can be resolved if one of the processes terminates. In starvation, the problem is (or can be) caused by a processes terminating.

Example: You are camping with your spouse, she has the car keys, the food is locked in the car (prevents bears from getting at it) and she is abducted by aliens. You starve.


EVER SINCE YOU MOVED OUR EMAIL SERVERS TO TRAN-SYLBONIA, MY INBOX HAS NOTHING BUT VOWELS.
Dilbert.com DilbertCartoonist@gmail.com
WE I.T. PEOPLE ONLY RESPOND TO WHOEVER COMPLAINS LOUDEST. YOU SHOULD COMPLAIN TO YOUR BOSS.
I WILL!
2-7-12 ©2012 Scott Adams, Inc./Dist. by Universal Uclick
A UI AOE UIE OU EAI!
?

10 08 Memory

Chapter 8

MEMORY MANAGEMENT

ADDRESS SPACE:

We have 2 types of address spaces:

1.Physical address – Memory address seen by the memory management unit (MMU). I.e. the actual hardware address.

2.Logical address – Address location generated by CPU.

MEMORY MANAGEMENT

ADDRESS BINDING:

When a program is loaded into memory the addresses of data and instructions must be mapped into memory locations. This is called address binding. The 3 methods of address binding are:

1. Compile-time – The compiler generates absolute memory addresses (i.e. DOS .COM files). These programs can only reside at a specific location in memory and if that location is already occupied, the program cannot load.

2. Load-time – The compiler generates *relocatable* code. With relocatable code each memory address is relative to a base address. So, when the program is loaded, it can load into any memory location.

3. Execution-time – The compiler generates *relocatable* code. However, since it is possible that the program will be moved to different memory locations during its execution phase, the base address may change. Requires special hardware (an MMU).

MEMORY MANAGEMENT

ADDRESS SPACE and BINDING:

Compile-time binding - logical addresses (generated by the CPU) and the physical addresses are the same.

Load-time binding - logical addresses and the physical addresses being the same (once the program is loaded).

Execution-time binding - logical addresses being different than the physical addresses. While the program runs the MMU re-maps the logical addresses to the correct physical address.

MEMORY MANAGEMENT

DYNAMIC MEMORY MANAGEMENT:

Up until the 1960's (batch OS's) it was common for one program to reside in memory. As programs outgrew the available memory methods were developed to conserve memory usage. These include:

- Dynamic loading – Only the main program loads at startup. Routines are not loaded until needed.

- Dynamic linking – Similar to dynamic loading. The compiler includes a stub for each routine used from the dynamic linked library (.DLL) in the user's program. The stub has information as how to load the routine. When a routine from the DLL is needed, it is loaded into memory and the stub is replaced with the memory address of the routine.

- Overlays – Only a portion of the program is kept in memory. When another part of the program is required, it is loaded and overwrites the existing portion of the program that was in memory.

- Virtual memory – Similar to overlays, except the OS manages the process.

MEMORY MANAGEMENT

MEMORY VS CPU USAGE:

With multiprogramming we can make better use of the CPU. However, the assumption is not always valid. On average, programs are 80% I/O dependent. Therefore, it is very likely that at any given time, all of the processes will be waiting for I/O and the CPU will have nothing to do. The probability that the CPU will be idle at any given time is:

$$P^n$$

and CPU utilization is:

$$1 - P^n$$

P = I/O dependence
n = # of processes

We have assumed that the processes are independent. This is not completely accurate. But, the model does give us a good idea as to what is going on.

MEMORY MANAGEMENT

MEMORY VS CPU USAGE:

Even though the model is simple it can still be used to make approximate predictions about CPU usage. For example:

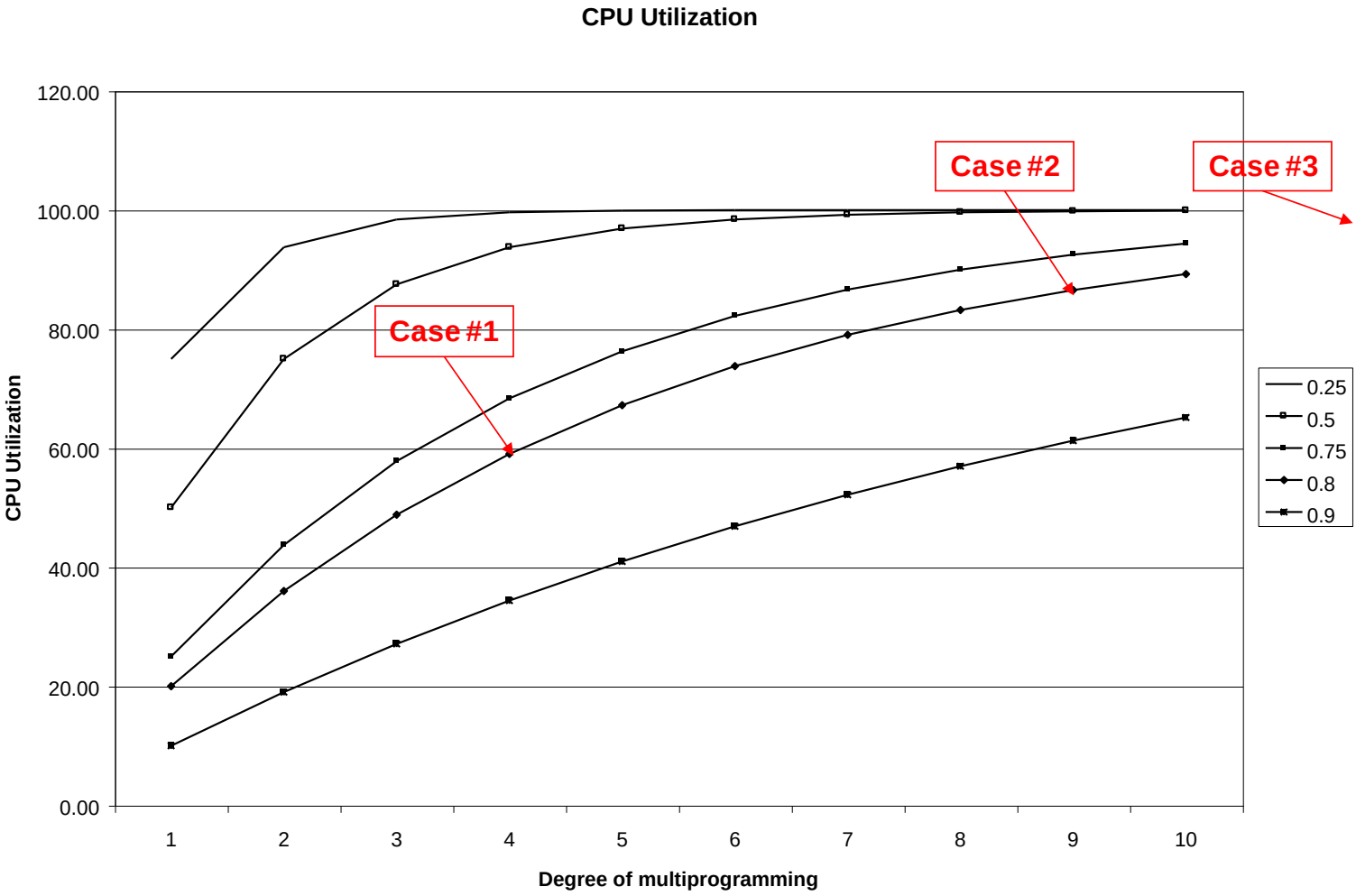
1. Assume we have 1Meg RAM and the OS requires 200K and each of the 4 users requires about 200K. With an average I/O dependency of 80%, we get a CPU utilization of about 68%.

2. If we add another 1Meg RAM, we can increase our degree of multiprogramming to 9 and we can raise our CPU utilization to about 87%. We've increased CPU utilization by about 28%.

3. If we add another 1Meg RAM, we can increase our degree of multiprogramming to 14 and we can raise our CPU utilization to about 96%. We've further increased CPU utilization by about 10%.

MEMORY MANAGEMENT

MEMORY VS CPU USAGE:

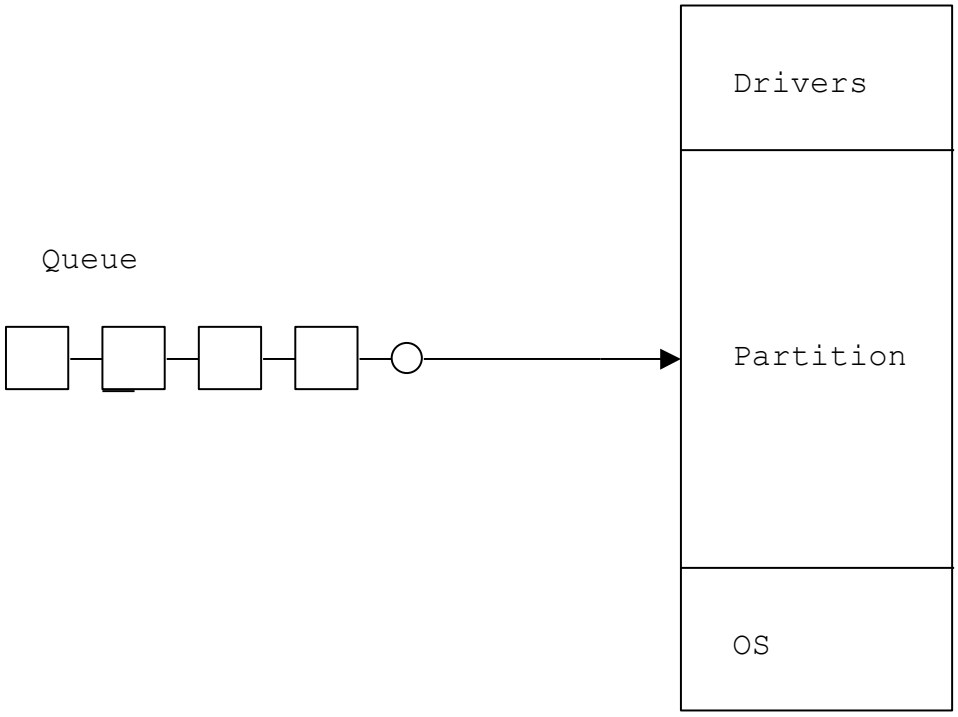


MEMORY MANAGEMENT

CONTIGUOUS MEMORY ALLOCATION:

With contiguous memory allocation, a program is allocated a chunk of memory large enough to hold the entire program. We can have a single memory partition or multiple memory partitions.

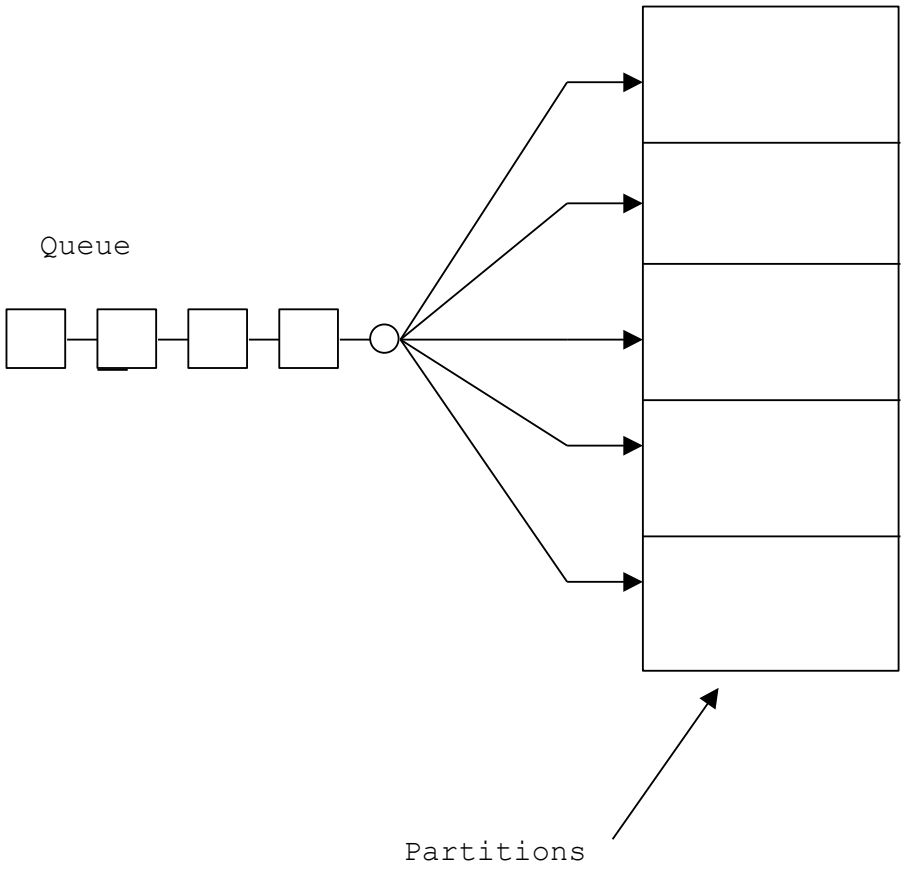
- Single memory partition – The OS and programs share the same memory partition. Used in DOS.



MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

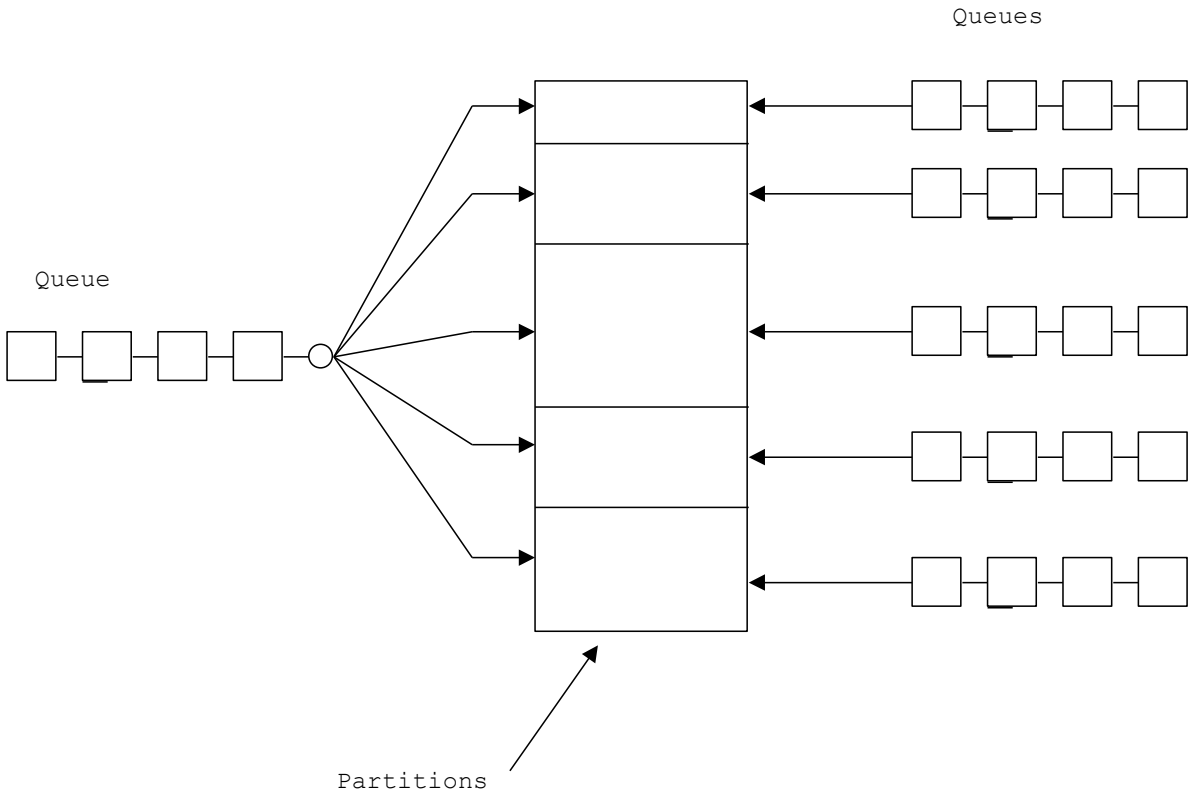
- Fixed partition #1 – To allow more than one user program in memory, we need to allocate multiple partitions. The simplest method is to divide the memory into N fixed sized partitions. Where each partition can only contain 1 process.



MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

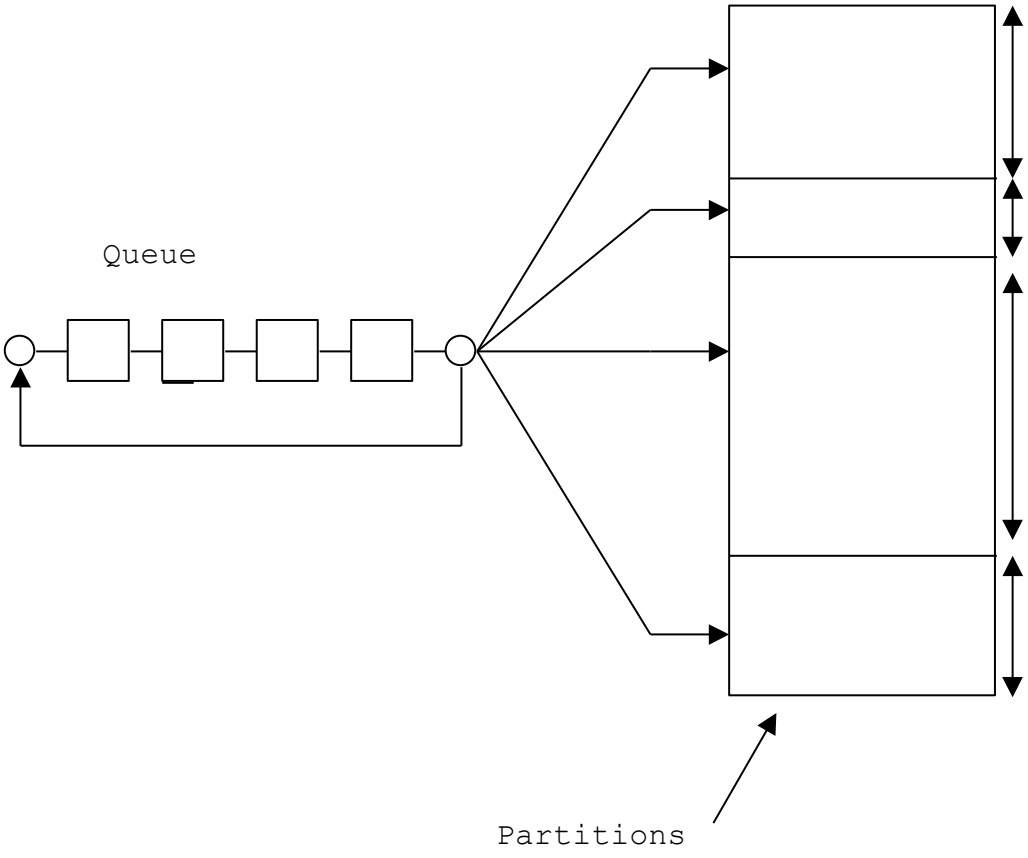
- Fixed partition #2 – The method used in the IBM 360 was to allow the sys-admin to define multiple partitions, each of different size. Therefore, the system could be custom configured on a day-to-day basis (memory protection was by a 4-bit code assigned to each partition).



MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

- Dynamic partition – This scheme assumes that the partition sizes are dynamic (from process to process). This method requires some type of data structure (bit map, linked list, or table) to maintain a list of occupied memory locations and holes.



MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

- Segmented – Supports the user's view of the memory. Where the memory is not broken down into fixed sized meaningless chunks, but into chunks (segments) of related material. Segments might include global variables, stack space, local variables for each routine, the code for each routine, etc.

  Segmentation makes protection and sharing easier.

  Segmentation memory management is very similar to contiguous multiple memory partition scheme #3 (variable sized partitions) and suffers from the same problem of external fragmentation and occasionally, internal fragmentation.

  Segmentation could be used in combination with paging.

MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

When using the **dynamic partition** memory partitioning scheme, when a process terminates its memory partition is returned to the system (creating a **hole** in the memory – we also assume that adjacent holes are combined to form 1 larger hole). When another process becomes ready, the OS searches for a hole big enough to hold the process. If none are found, the process must wait until a large enough hole is available. If one is found, the process is allowed to start and to acquire the memory it needs (the left over memory is now marked/recorded as a hole).

Since we have random sized holes and processes requiring random sized holes, we need a method to manage this*. Assumptions:

1. That there are holes scattered throughout the memory.

2. That adjacent holes are combined into a single larger hole.

3. That memory compaction (i.e. disk defragging) is not done because it is too expensive (except on the CDC Cyber mainframes which had special hardware to do this).

* This same problem occurs with many disk file systems.

MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

External fragmentation – As processes are loaded and terminated, the memory holes get broken down into smaller and smaller holes.

- 50% rule - With the BEST-FIT allocation method, averaged over time, there will be half as many holes as processes. Therefore, 1/3 of the total memory will be wasted. This rule also applies to swap systems.

- Unused memory rule – Used to determine percentage of wasted memory (%W). Let:

  K = average hole size / average process size

  %W = K / (K + 2)

Internal fragmentation – A process may require all but a few bytes of an available hole. Since it is probably not worth the effort to allocate only the required memory and have to record a small hole, we would allocate the entire hole to the process. The leftover unused section is called the internal fragmentation.

MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

We know that fragmentation is a concern and that compaction is not feasible, so several allocation methods have been proposed to reduce the fragmentation problem:

- First fit – Allocate the first hole that is big enough (a greedy method). Fast, but the problem here is that we may allocate a hole that is much larger than required.

- Best fit – Allocate the smallest hole that is big enough. Since we must search the entire list of holes, this method is slow, but does the best job (simulations suggest it is about equal to FIRST-FIT) of allocating memory.

- Worst fit - Allocate the largest hole. Since we must search the entire list of holes, this method is slow, and does the worst job of allocating memory.

- Quick fit – Maintains a list of commonly requested partition sizes. Fast, but only for holes of exactly the commonly requested size.

- Buddy system – Maintains a dynamic binary tree of holes (leaf nodes are called buddies). Fast, but is inefficient as all memory partitions must be a power of 2.

MEMORY ALLOCATION

CONTIGUOUS MEMORY ALLOCATION:

If a program, using dynamic memory techniques, tries to acquire additional memory after it has begun execution, we have 3 choices:

1. If there is an adjacent hole, allocate the required additional memory from it.

2. Rearrange the processes in memory such that an adjacent hole is formed.

3. Kill the process.


HELLO, WALLY. HUMAN RESOURCES ASSIGNED ME TO BE YOUR ERGO BUDDY.
MY WHAT?
DilbertCartoonist@gmail.com
I'LL OBSERVE WHILE YOU WORK, THEN GIVE YOU SUGGESTIONS ON IMPROVING YOUR ERGONOMICS.
YOUR POSTURE IS ALL WRONG, AND YOUR KEYBOARD NEEDS TO BE LOWER.
YOUR BAD ERGONOMIC PRACTICES HAVE TURNED YOU INTO A GOLEM.
GOLEM?
© 2010 Scott Adams, Inc./Dist. by UFS, Inc.
IT MEANS A SHAPELESS MASS IMBUED WITH LIFE. A LUMP, CLOD, OR FOOL.
IT'S RIGHT HERE ON MY CHECKLIST, AFTER "COMPUTER SCIENTIST."
DOES THAT SAY YOU'RE SUPPOSED TO POUR SALT ON ME UNTIL I DRY UP AND DIE?
LET'S PRETEND YOU DIDN'T SEE THAT.
www.dilbert.com

11 09 Paging

Chapter 9

PAGED MEMORY ALLOCATION

Paging is a method used to manage the overlay process in virtual memory systems (UNIX and Windows10 are virtual memory systems).

Paging requires the memory to be divided into equal sized chunks. Physical memory chunks are called **page frames** and logical/virtual memory chunks are called **pages**. The frames and pages are always of equal size. A data structure is used to log the location and use of each page.

Paging allows noncontiguous memory allocation.

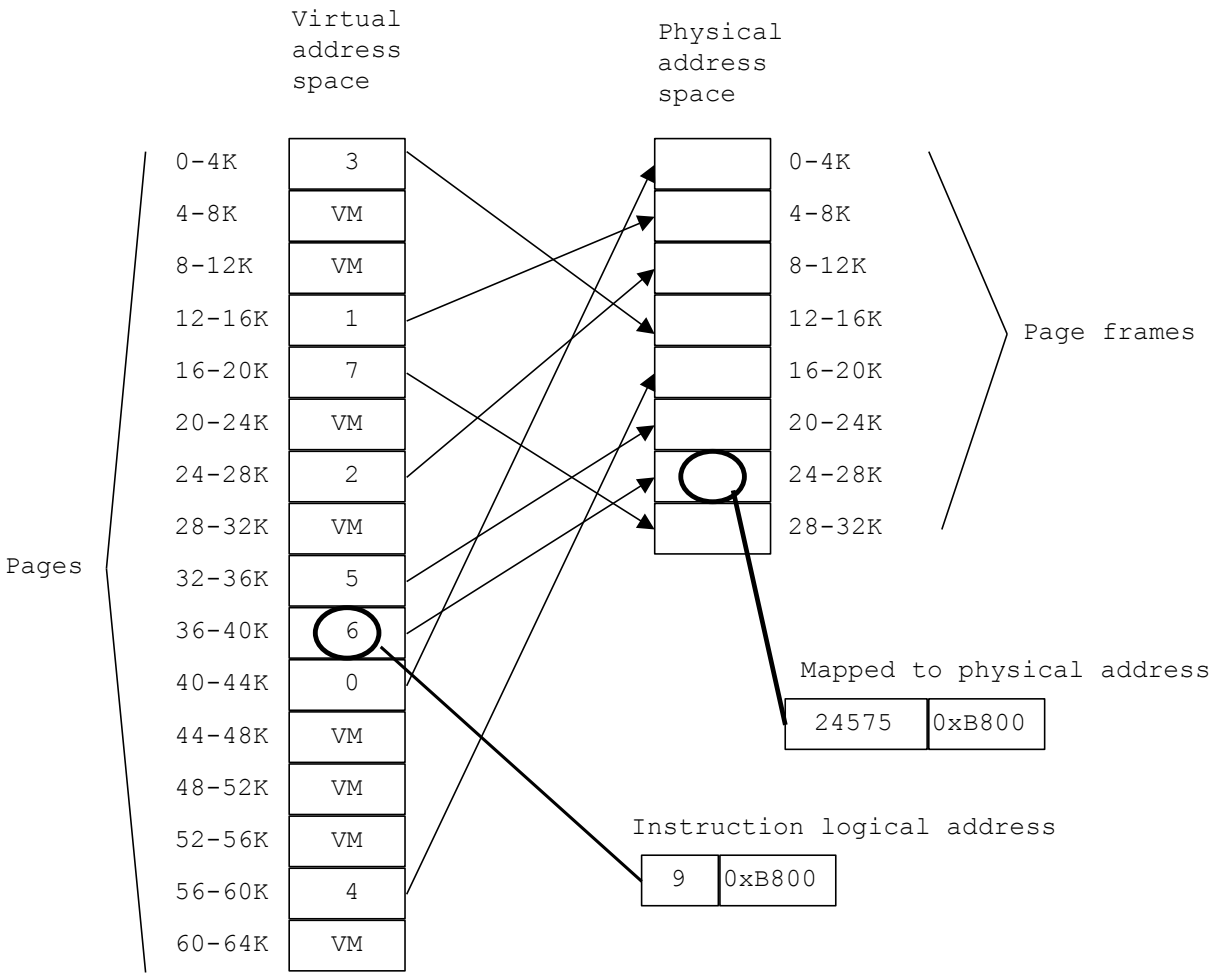
Paging is a form of dynamic relocation.

Paging eliminates external fragmentation (any free page frame can be allocated to any waiting page).

Paging does not eliminate internal fragmentation.

PAGED MEMORY ALLOCATION

EXAMPLE – A system w/32K of RAM, 64K of virtual memory, and 4K pages:



PAGED MEMORY ALLOCATION:

PAGE TABLES:

We must expect a lot of page (memory) accesses and we want these accesses to be fast, thus any data structure (page table) that is used to manage the pages must be efficient. Methods include:

- Hardware – Using a set of dedicated registers. Fast, but limits the number of pages possible.

- Hardware – Using a protected area of main memory and a single register (page-table base register). Allows for many pages, but is very slow.

- Hardware – Using a memory cache (**translation look-aside buffer**). Fast, but expensive. So we only keep the part of the page table that references active pages in the buffer. Other page requests require accessing the main memory version of the page table. However, each context switch requires that the buffer be flushed (to prevent using the wrong pages).

To provide memory protection, the page table may also include a set of bits for protection.

PAGED MEMORY ALLOCATION:

PAGE TABLES:

Because modern computer systems can include large amounts of memory, the flat-file approach to page tables is not feasible. Options include:

- Two-level page tables – An index to an index idea.

- Multi-level page tables – Similar to the above, except multiple layers of tables are possible. Advantages and disadvantages are the same.

- Inverted page tables – Usually, each process has a page table associated with it that records the pages required by the process. However, the individual page tables may be very large.

  Inverted page tables only have an entry for each frame page (physical memory). Each entry in the page table includes the PID of the process (owning the associated frame page) and a page number (what virtual memory page is located there). Advantages are that less memory is required to store the page table. Disadvantages are it is slower (since the table is not sorted by PID or page number, a linear search is required).

PAGED MEMORY ALLOCATION:

PAGE TABLES:

Page number/size – 2 possibilities:

1. Split the memory into N frame pages (may be limited by hardware constraints).

2. Smaller page sizes tend to reduce internal fragmentation, yet result in a larger page table. If we consider only wasted memory and the page table size, the optimal page size (*P*) can be computed from:

$$P = \sqrt{2se}$$

s is the average process size
e is the number of bytes per page table entry.

PAGED MEMORY ALLOCATION:

PAGE FAULTS:

Whenever a process tries to access a page not in physical memory, a page fault is generated. This causes the CPU to pick an unused or little used frame page, write its contents to disk (if it was being used), fetch and load the referenced page (which caused the page fault), re-map the page table (to reflect the page swap), and continue processing the process.

Sounds simple enough, but how do we decide exactly what page to remove?

We must also realize that some pages should never be replaced. We use **frame locking** to “lock” those pages in memory.

PAGED MEMORY ALLOCATION:

PAGE FAULTS:

Page replacement options include (we assume that all of the pages are in use):

- Optimal page replacement – We remove the page that will not be required for the longest time. Impossible to implement!

- Least Recently Used (LRU) – Assuming that we have a method to keep track of the usage of pages, we can remove a page that has not been used in the longest time. Theoretically possible, but expensive.

- Least-Frequently-Used (LFU) – By keeping a counter for each page, we can determine which page to remove by determining the least referenced page. Expensive and faulty!

- Most-Frequently-Used (MFU) – By keeping a counter for each page, we can determine which page to remove by determining the least referenced page

PAGED MEMORY ALLOCATION:

PAGE FAULTS:

- Clock – Requires 1 additional bits (use bit) associated with each page. Whenever we load or access a page we set the use bit to 1. Whenever we want to replace a page, we scan the buffer (implemented as a circular list) looking for a page with a use bit set to 0. If we find a page with a use bit of 1, we reset that bit to 0 and move on.

- Additional-Reference-Bits – By keeping an n-bit word for each page in the page table, we can determine which page(s) have not been used recently. At regular intervals the OS shifts the word right by 1 bit. The reference (R) bit is put on the left. The numeric value of the word indicates how long ago it was last referenced.

PAGED MEMORY ALLOCATION:

PAGE FAULTS:

- Not-Recently-Used (NRU)– Requires 2 additional bits associated with each page (a referenced bit “R” and modified bit “M”). Whenever we read from or write to a page we set the R bit to 1. Whenever we write to a page we set the M bit to 1. When a process is started, both bits for all of its pages are set to 0. Periodically, the OS resets the R bit to 0. When a page fault occurs the OS inspects all pages and assigns them to 1 of 4 categories:

  Class 0: R = 0, M = 0
  Class 1: R = 0, M = 1
  Class 2: R = 1, M = 0
  Class 3: R = 1, M = 1

  Also known as the Enhanced-Second-Chance algorithm.

  Also considered a variation of the Clock approach.

  The OS removes (at random) a page from the lowest non-empty class set.

  Easy to implement and provides good (but not optimal) performance.

PAGED MEMORY ALLOCATION:

PAGE FAULTS:

- First-In, First-Out (FIFO) #1 – We simply remove the oldest page.

- First-In, First-Out (FIFO) #2 – We modify the NRU algorithm by removing the oldest page from the lowest non-empty class set.

- Second-Chance – A modification of the FIFO method. We inspect the reference bit of the oldest page. If the bit is 0 it is removed. Otherwise, we reset its reference bit to 0 reset its time-stamp to the current time.

It would seem that the more memory, the fewer page faults would be generated. However, it has been shown (Belady et al 1969) that with FIFO this is not always the case. This is known as Belady's anomaly.

PAGED MEMORY ALLOCATION:

PAGING PERFORMANCE:

- Page-buffering – The OS keeps a set of free pages, so it can load a page and get the process started before spending the time to write the replaced page out to disk.

- Pre-paging – The working set pages (or neighboring pages) are loaded before execution starts.

- Paging on demand – Pages are not loaded into memory until they are required.

- Proportional allocation – Allocate the number of frames based on the size of the program.

PAGED MEMORY ALLOCATION:

PAGING PERFORMANCE:

Local vs global allocation policies – When the OS chooses a page to remove does the OS only consider pages belonging to the process causing the page fault (local policy) or does the OS consider all pages (global policy)?

Generally, global policies work better.

Cleaning – When does the OS synchronize a page with that on the disk? Two common methods:

1. Demand cleaning – A “dirty” page is written out only when selected for replacement.

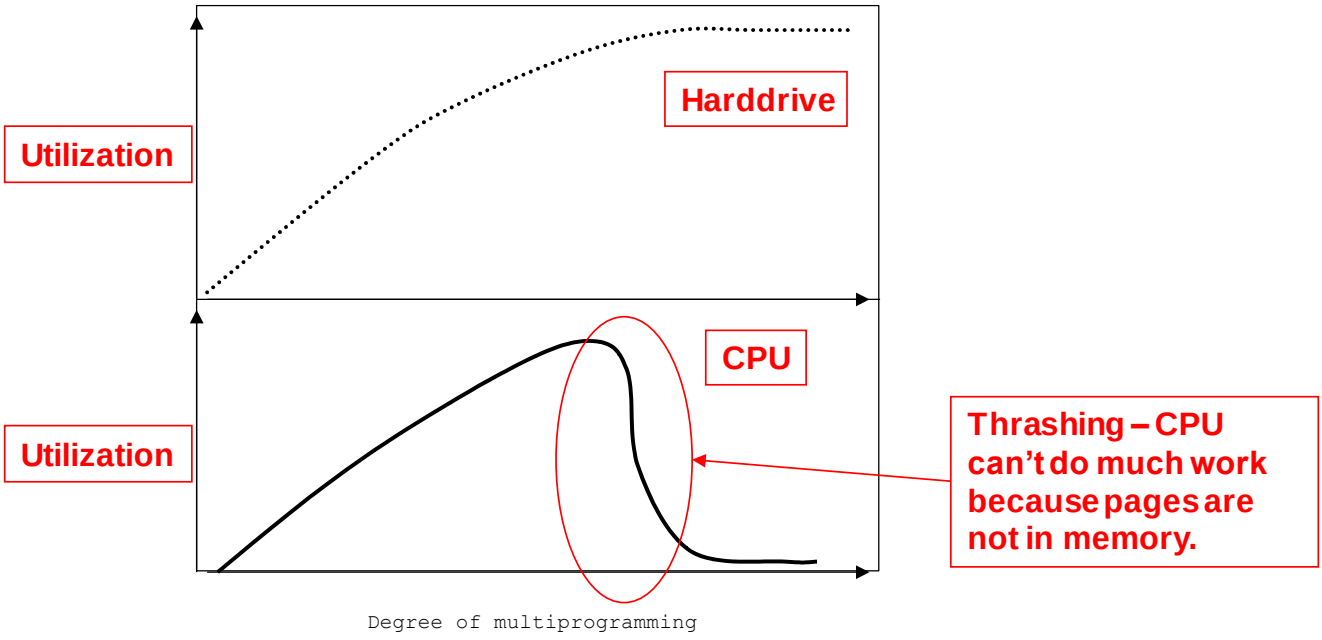
2. Pre-cleaning – Writes “dirty” pages out before selected for replacement. Allows for “batch” writes to disk.

PAGED MEMORY ALLOCATION:

THRASHING:

Thrashing – When page faults occur every few instructions. Typically, the case is that several processes will be in contention for memory space and (assuming a global allocation policy) will generate page faults that will remove each other pages. It's a vicious cycle.

We can run into a similar problem when we have a shared file system (on a server). As more and more processes request a file from the server, the server gets backlogged. However, this is not the same as thrashing:



PAGED MEMORY ALLOCATION:

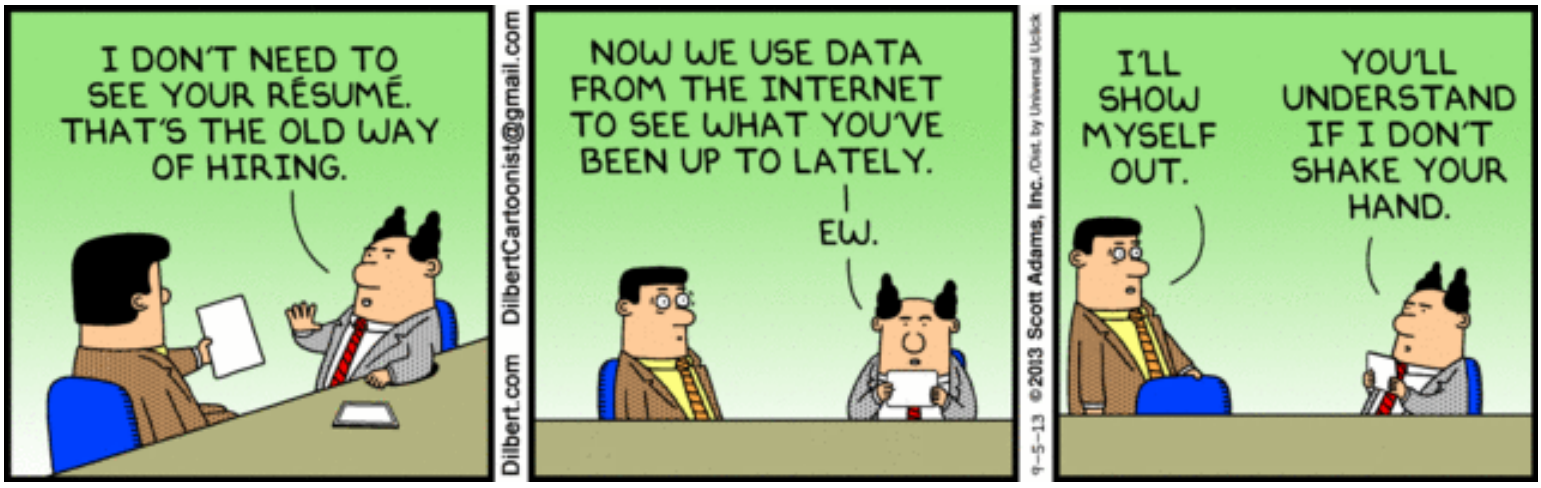
THRASHING:

Methods to reduce thrashing include:

1. Local allocation policy.

2. Page fault frequency (PFF) allocation – With Least-Recently-Used (LRU), increasing the number of pages available reduces the number of page faults. Therefore, if a process is generating a lot of page faults, we would remove some other process's page(s) giving the problem process more pages to work with.

3. Working-Set model allocation – We keep track of N currently active pages for each process. The OS only allows a degree of multiprogramming such that each process can be allocated its working-set of pages. Prevents thrashing and provides a near optimal use of memory, however it is expensive to implement and the determination of N is/can be difficult.


I DON'T NEED TO SEE YOUR RÉSUMÉ. THAT'S THE OLD WAY OF HIRING.
Dilbert.com DilbertCartoonist@gmail.com
NOW WE USE DATA FROM THE INTERNET TO SEE WHAT YOU'VE BEEN UP TO LATELY.
EW.
9-5-13 ©2013 Scott Adams, Inc./Dist. by Universal Uclick
I'LL SHOW MYSELF OUT.
YOU'LL UNDERSTAND IF I DON'T SHAKE YOUR HAND.