Lecture 16 - P1: Intro to Processes

Mon. Oct. 07, 2019

Process Basics

- A process is an instance of a program being executed
  - Core operating system (OS) concept
- In a multiprocessing OS
  - Multiple programs can be executed at the same time
  - Multiple instances of a program can be executed at the same time
- Executing multiple programs
  - Single-core: time-sharing
  - Multi-core: true parallelism + time-sharing

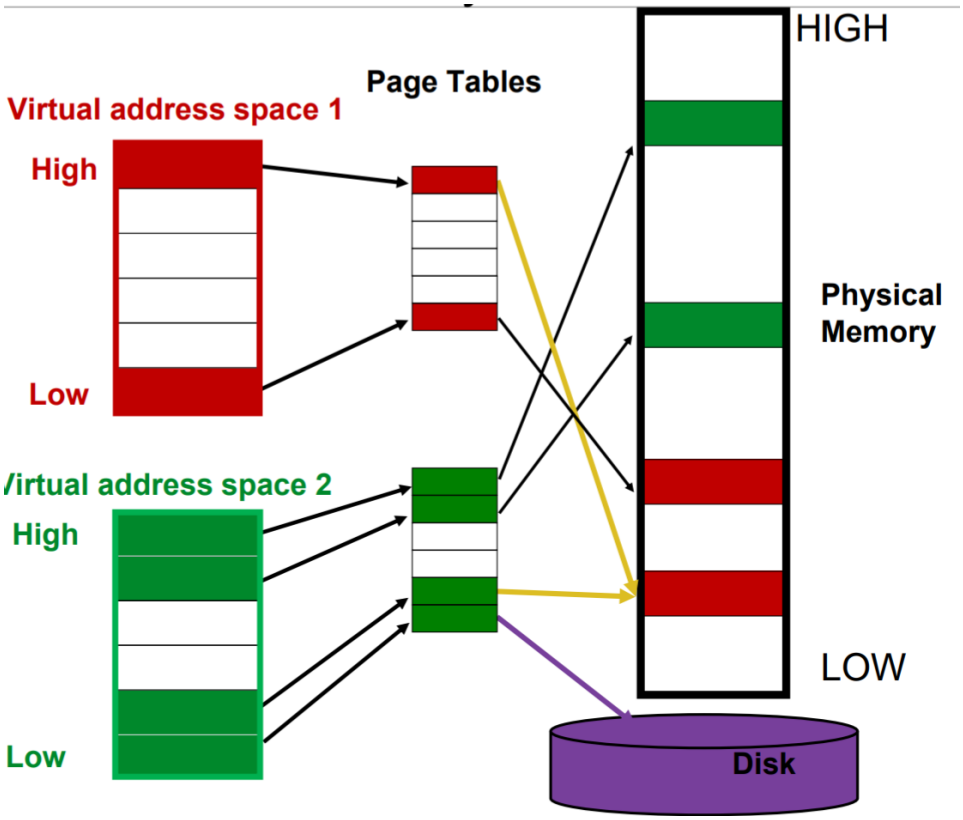
Process Management: OS View

- OS maintains a process table
  - Each process has a table entry, called process control block (PCB)
  - Typical PCB info
- OS scheduler picks processes to be executed at any given time
  - When a process is suspended, its state is saved in PCB
  - What about the process memory?

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Paged Virtual Memory: How Processes Share Memory

- Physical memory is shared by all processes
- Page table maps virtual address to physical address
- Multiple virtual pages can be mapped to the same physical pages



Process Management: User's View

- Events which cause process creation
  - System initialization
  - User request to create a new process (e.g., shell command)
  - Executing a shell script, which may create many processes
- Events which cause process termination
  - Normal program exit
  - Error exit
  - Fatal error, e.g., segmentation fault
  - Killed by user command or signal (Ctrl-C)

Useful Commands

- **ps** - List running processes
- **pstree** - Display the tree of processes
- top - Dynamic view of memory & CPU usage + processes that use most resources (to exit top, press q)
- **kill** - Kill a process given its process ID
  - **Try -9** option if simple kill does not work
- Additional functions/options in man page of each command

CSE 3100 Master Notes

Process Management: Programmer's View

- Process birth
  - Processes are created by other processes!
  - A process always starts as a clone of its parent process
  - Then the process may upgrade itself to run a different executable
    - Child process retains access to the files open in the parent
- Process life
  - Child process can create its own children processes
- Process death
  - Eventually calls exit or abort to commit "suicide"
  - Or gets killed

Birth via Cloning

- The function to create a new process in your code

```
#include
pid_t fork(void);
```

- Child is an exact copy of the parent
  - Both return from fork()
- **Only difference is the returned value**
  - In the **parent** process:
    - fork() returns the process identifier of the child (> 0)
    - If a failure occurred, it returns -1 (and sets errno)
  - In the **child** process: fork() returns 0 (zero)

Concurrency

- Parent and child processes return from fork() concurrently
  - They may return at the same time (on a multicore machine) or one after the other
  - Cannot assume that they return at the same time or which one "returns first" (even on a uni-core)
    - Order is chosen by OS scheduler

Cloning Effect

- On memory
  - The parent and child memory 100% identical
  - But are viewed as distinct by OS ("copy-on-write")
  - Any memory change (stack/heap) affects only that copy
  - Thus the parent and child can quickly diverge
- On files
  - *All files open in the parent are accessible in the child!*

    - I/O operations in either one move the file position indicator
- In particular
    - *stdin, stdout*, and *stderr* of the parent are accessible in the child

What can the parent do?

- Depends on application!
    - It could wait until the child is done (dies!)
        - Typical of a shell like bash/ksh/zsh/csh/….
    - It could run concurrently and check back on the child later
    - It could run concurrently and ignore the child
        - If child dies it enters a zombie state

Waiting on a child

```
#include <sys/wait.h>

pid_t wait(int * status);
pid_t (waitpid(pod_t pid, int * status, int options);
```

- Purpose
    - Block the calling process until a child is terminated
        - Or other state changes specified by options
    - Report status in *status (which is ignored if NULL is passed)
        - The cause of death
        - The exit status of the child (what he returned from main)
    - Return value identifies the child process (or -1 on error)
    - Run “man -S2 wait” for full details

Zombies

- A dead process, waiting to be 'reaped' (checked by its parent)
    - You cannot kill it, because it is already dead
    - Most resources released, but still uses an entry in the process table
- Parents should check their kids
    - On some systems, parents can say they do not want to check
- When a parent dies, ‘init’ becomes the new parent
    - Then the zombie child is reaped

System calls

- APIs used to request services from the OS kernel
    - Example: fork()
    - System calls are more expensive than normal function calls
    - Manuals for system calls are in section 2

man -S2 intro ; man -S2 syscalls

Summary

- Clone a process with fork()
    - The child is exactly the same as the parent
    - Check the return value
- Parents wait for child processes
    - Reap the zombies!

Lecture 17 - P2: Exec and Low-Level I/O

Mon. Oct. 09, 2019

Process upgrades

- Usually…
    - A fresh clone wants to run different code
- This is done by
    - Loading another executable3 into the process address space
        - [picked up from the file system of course]
- Note: **open files are NOT AFFECTED** by the upgrade operation

The exec family

- The act of “upgrading’ is done by the child with a system call
    - Many variants ‘many -S3 excel” for all details

```
#include <unistd.h>
int execl(const char *path, const char *arg0, …
                        /*, (char *) NULL */ );
```

- The path to the executable to load inside our own address space
- A list of arguments to be passed to the new executable
- A final NULL pointer to give the “end of argument list”
- If successful, execl() does not return! Started a new process

Exec example

- We will turn the child process into the following executable

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

int main(int argc,char* argv[]) {
   int i, sum=0;
   for(i=1;i<argc;i++)
      sum += atoi(argv[i]);
   printf("sum is: %d\n",sum);
   return 0;
}
```

This is a simple "adder" program that computes the sum of its integer arguments

Parent Program

```c
int main() {  // complete code is in demo/padder
   char *cmd1 = "./adder", *cmd2 = "addder";
   pid_t child = fork();
   if (child == 0) {
      printf("In child!\n");
      execl(cmd1,cmd1,"1","2","3","10",NULL);
      printf("Oops.... something went really wrong!\n");
      perror(cmd1);
      return -1;
   } else {
      printf("In parent!\n");
      execl(cmd2,cmd2,"100","200","300",NULL);
      printf("Oops.... something went really wrong!\n");
      perror(cmd2);
      return -1;
   }
}
```

How is executable found?

- Specify a path, like /bin/ls
- Specify a file, and the system searches in directories listed in PATH
    - echo $PATH in bash to see directories separated by ':'

in execl(const char *path, const char *arg0, …

```
                        /*, (char *) NULL */ );

// execlp() searches paths for file
int execlp(const char *file, const char *arg0, ….
                        /*, (char *) NULL */ );
```

execv family

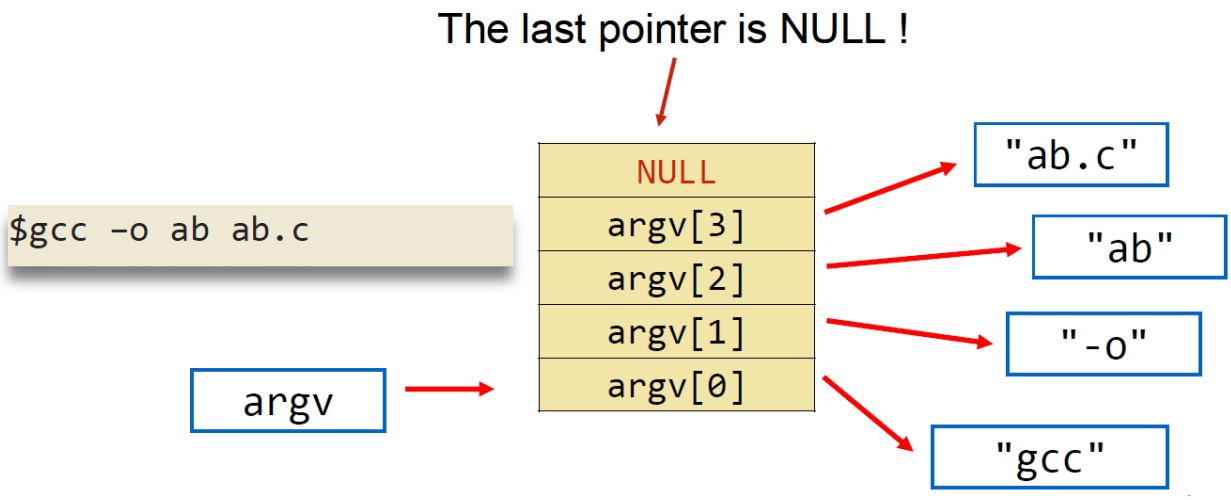
// If the number of arguments is unknown at compile time

```
#include <unistd.h>
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- The arguments in execl in are placed in an array
  - argv is the argv you see in the main function!
- execv needs a path while execvp can search file in PATH
- Start a new process if successful. Similar to execl

argv to execv and execvp

- Note the NULL pointer at the end
- Why?



File APIs

- Remember the (C standard library) IO APIs
  - The "**f**" family (fopen, fclose, fread, fgetc, fscanf, fprintf,...)
  - All these use a FILE* abstraction to represent a file
    - Additional features: user-space buffering, line-ending translation formatted I/O, etc.
- UNIX has lower-level APIs for file handling
  - Directly mapped to system calls

        - Open, close, read, …
    - Use file descriptors [which are just integers]
    - Deal with bytes only

Some low level file APIs

- Read the man pages (man -s2 …) for more functions

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
int close(int fd);

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);

off_t lseek(int fd, off_t offset, int whence);
```

Open a file

```
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag);
```

- Parameters
    - path: the path to the file to be open/created
    - oflag: read, write, or read and write, and more (on the next slide)
- The function returns a file descriptor, a small, nonnegative integer
    - Return -1 on error

Flags in open()

- Must include one of the following:

O_RDONLY (read only), O_WRONLY (write only), or O_RDWR (read and write)

- And or=ed (|) with many optional flags, for example,
    - O_TRUNC: Truncate the file (remove existing contents) if opening a file for write
    - O_CREAT: Create a file if it does not exist

Example:

```
// remember open() returns -1 on error
```

```
fd1 = open("a.txt", O_RDONLY);   // open for read
fd1 = open("a.txt", O_RDWR):     // open for read and write
fd1 = open("a.txt", O_RDWR | O_TRUNC);read, write, truncate the file
```

Create a file with open()

```
// a mode must be provided if O_CREAT or O_TMPFILE is set
int open(const char *path, int oflag, int mode);
```

- mode: specify permissions when a new, or temporary, file is created

```
open("b.txt", O_WRONLY|O_TRUNC|O_CREAT, 0600);

// open b.txt for write. If the file exists, clear (truncate) the
contents.

// if the file does not exist, create one, and set the permission so
that the owner of the file can read and write, but other people
cannot.
```

File descriptor vs stream

```
#include <stdio.h>
int fileno(FILE *stream);
// returns a file descriptor for a stream
```

FD	FILE *
0	stdin
1	stout
2	stderr

File descriptors after fork and exec

- Opened files are NOT AFFECTED by the upgrade operation

```
pid_t pid = fork();
assert(*pid >= 0);
if (pid == 0){
      // Child process can access FDs 0, 1, and 2
      // if execl() is successful, gcc can access FDs 0, 1, and 2
      execlp("gcc", "gcc", "a.c", NULL);
      // If control gets here, execlp() failed.
      // Remember to terminate the child process!
```

```
    return 1;
}
```

Lecture 18 - P3: Redirection

Mon. Oct. 14, 2019

Review

- Function open() returns a file descriptor, a non-negative integer
- the file descriptor is used later in funcs. like read() and close()
- Every opened file has a file descriptor
  - stdin: 0
  - stdout: 1
  - stderr: 2
- Files opened in a process remain open after fork() and exec(0

Shell redirections

- Available when executing commands in your shell (e.g. bash)
  - Implemented with the close/open/dup technique

```
$ command <infile > outfile
```

  - < infile: Take input from file infile
  - > outfile: Send output to file outfile
- Other variants
  - >> outfile: Append output to file outfile
  - 2> outfile: Send errors to file outfile
  - &> outfile: Send both output and errors to file outfile
- Read the manual for more variants like 2>>, 2>&1, etc.

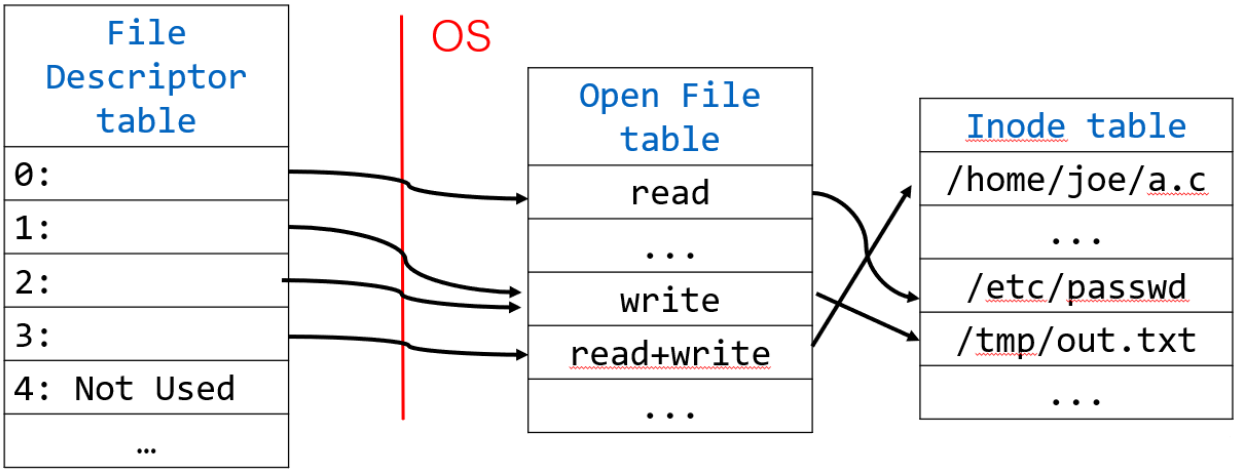
Shell redirection examples

```
$ sort < file.txt > sorted.txt
```

- sort will read lines from file.txt, instead of terminal
- The output or sort will be saved in sorted.txt
  - You cannot see it on screen
- The statements in sort are not changed
- They read from stdin (0), and print to stdout (1)

File Descriptor Table

- Each process has a **file descriptor table**
  - Holds indices of entries into the **Open File Table** managed by OS
- The system-wide **Open File Table**
  - Records the mode of the opened files (e.g., reading, writing, appending)
  - Holds index into the **Inode Table** that has the actual file name and location on disk



Duplicating File Descriptors

- Do not change file descriptor table directly
- Used open() and close() and two new functions

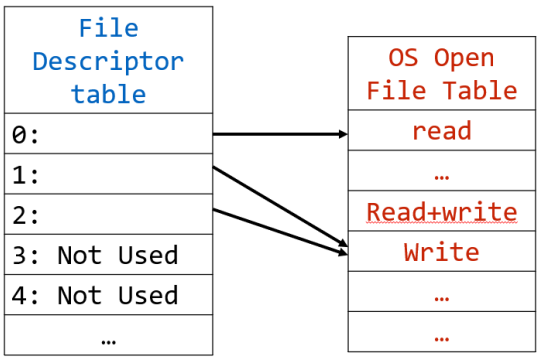
```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- dup() copies oldfd to the **first available entry** (in FD table)
- dup2() copies oldfd to newfd
  - Closes newfd if it is in us

***There is dup3(), but it is not in POSIX. We should not use it in this course.

Example: stdout redirect

- A program can change its standard output
- How?



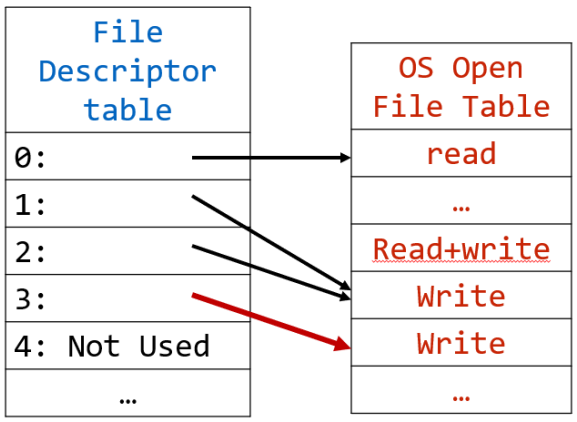
Steps for redirecting stdout

1. open(). open file (and save the file descriptor in fd)
2. dup2(). copy fd to 1 (so that the file descriptor 1 points to the file just opened)
3. close(fd)

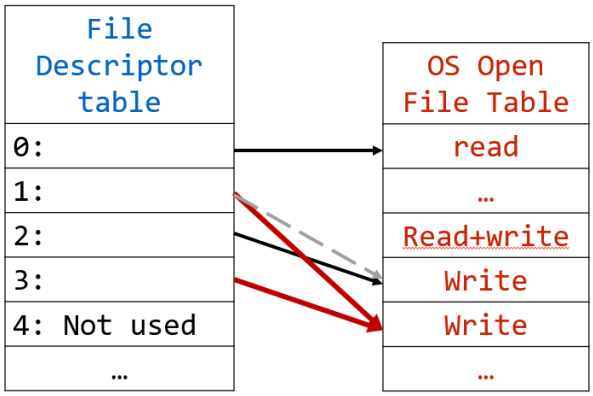
Example: stdout redirect

<table>
<tr>
<td>

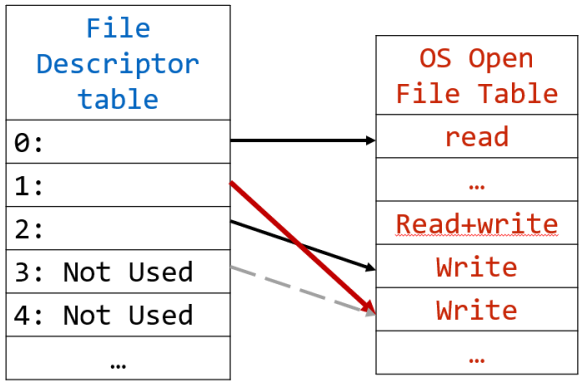
- Open the new file for writing; 3 is the returned fd
  - We will use 3 instead of a variable in this example

</td>
<td> 
</td>
</tr>
<tr>
<td>

```
// Method 1: two functions. not atomic
close(1);  dup(3);
// Method: better. dup2() closes newfd first
dup2(3, 1);
```

</td>
<td> 
</td>
</tr>
<tr>
<td>

```
close(3);
```

</td>
<td> 
</td>
</tr>
</table>

Implementing redirections

- How does bash do redirection for other processes?
- When bash starts a new process, it does fork() and exec()
- Recall that the file descriptor table **is preserved during fork & exec**

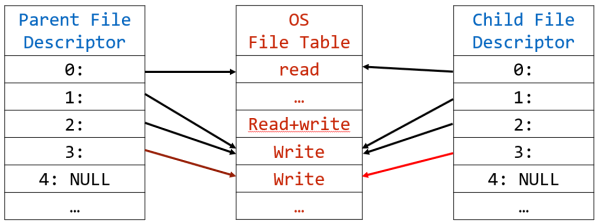
Idea:

- In child process, set up proper file descriptors before upgrading
    - Simply change the files corresponding to stdin, stdout, or stderr

Example: redirecting stderr for another program

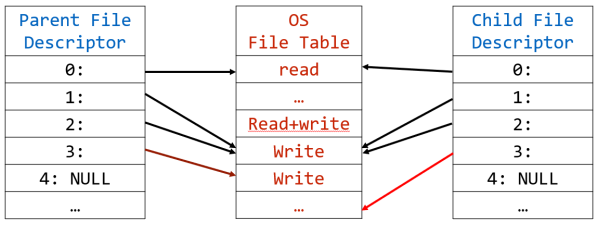
<table>
<tr>
<td>

- Assume the parent uses FD 3
- After fork(), the child has the same file descriptors as the parent
    - Close FDs that are not needed!

</td>
<td> 
</td>
</tr>
<tr>
<td>

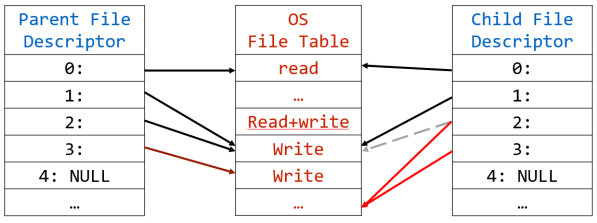
How do you close FD 3 in child process?

- Child: open the file (to save error output)

</td>
<td> 
</td>
</tr>
<tr>
<td>

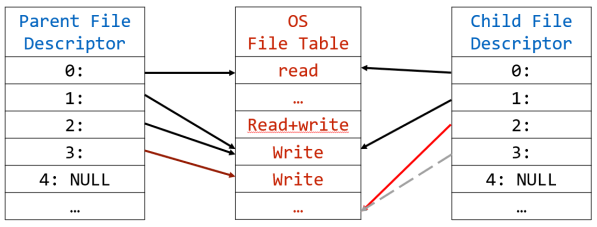
- Child:

```
dup2(2, 3);
close(3); // not yet run
```

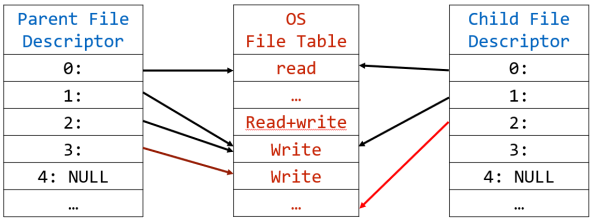
</td>
<td> 
</td>
</tr>
<tr>
<td>

- Child:

```
dup2(2, 3);
close(3); // now running
```

</td>
<td> 
</td>
</tr>
</table>

- Child: exec and the new executable has redirected stderr!
- On exec, open file descriptors are preserved!



Redirecting stdout for a child process

- A process would like to start a new program, and redirect stdout of the new process to a file
- Where & when should the file be opened?
- Select the best options.

a. Before fork(), in parent
b. After fork() in parent
c. Before exec in child (after fork())
d. After exec in child (after fork())
e. None of the above

Summary of Steps in Child Process

- Close FDs that are opened in parent and not needed in child
- open(). Open a file (and save the file descriptor in fd)
- dip2(). Copy FD to the right place
- close(fd)
- Exec

More questions on redirecting stdout?

- Where & when should the new file be opened?
- Where & when should you call close(1)/
- Where & when should you call dup?
- Where & when should close (newfd) be called?

Summary

- A program can direct input/output
  - It is done with open(), close(), dup(), or dup2()
- FDs are preserved on exec
  - Close file descriptors that are not needed

Lecture 19 - P4: Inter-Process Comm. w/ Pipes

Weds. Oct. 16, 2019

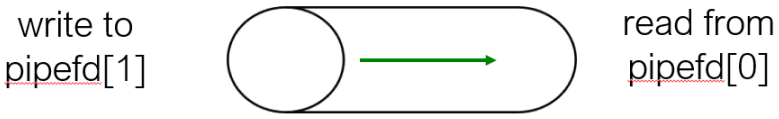
Inter-process communication (IPC)

- Files
- Pipes
- Named pipes
- Sockets
- Message queues
- Shared memory
- Synchronization primitives
    - Semaphores, Signal, etc

pipe()

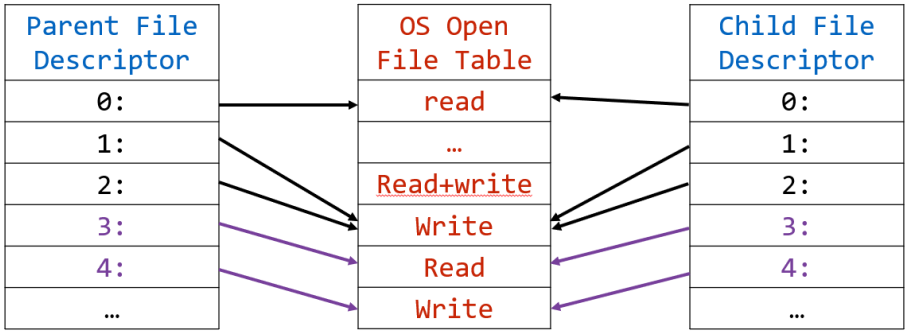
#include <unistd.h>
int pipe(int pipefd[2])

- Create a one-way pipe (a buffer to store a byte stream)
- Two FDs in pipefd. pipefd[0] is the read end, pipefd[1] is the write end
- Return 0 if successful
- Pipes allow IPC. One process writes and the other one reads



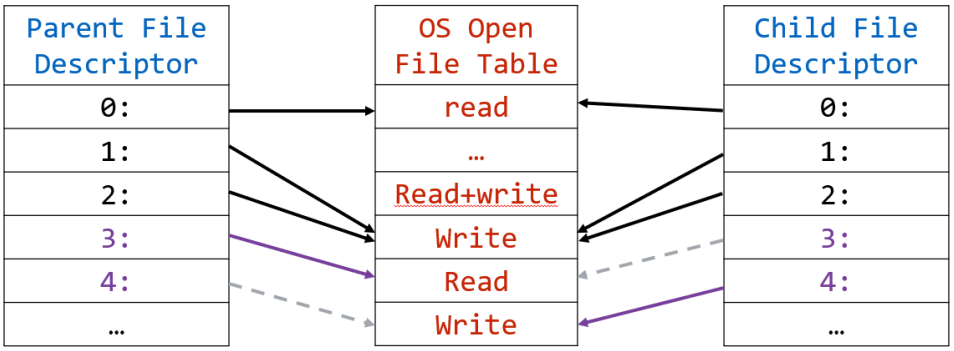
Connecting two processes

- Parent creates a pipe and gets two FDs (e.g. 3 and 4)
- After fork(), the child has 3 and 4, too
- One process can write to FD 3, and the other can read from FD 4
    - Close unused FD!



Closing FDs not in use

- If the pipe is for parent to read and for child to write
  - Parent: close(4)
  - Child: close(3);
- Then the child can write to and parent can read from the pipe. See demo code!



Questions

- What would you do if you need two-way communication between parent and child?
- After exec, the new program gets the file descriptors for the pipe, too
- How can the new program use the pipe?
  - A program is aware of FDs 0, 1, and 2, but not 3 or 4

Pipeline in shell

- Shell supports pipelines

```
cmd | cmd2 arg 21 arg22 | cmd3 arg31
```

- stdout of a command is connected is to stdin of the nex command
  - Done with pipes on Linux/Unix
  - cmd1 writes to a pipe and cmd2 reads from it
- Example:

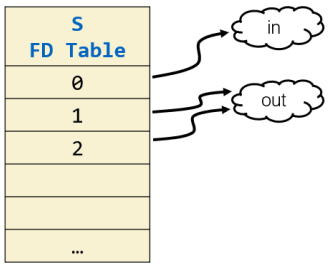
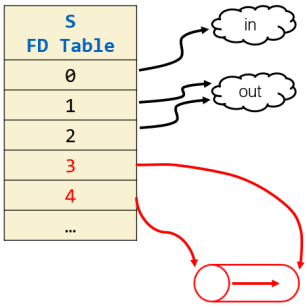
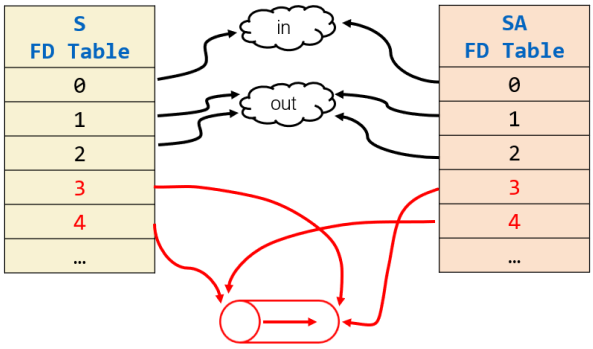
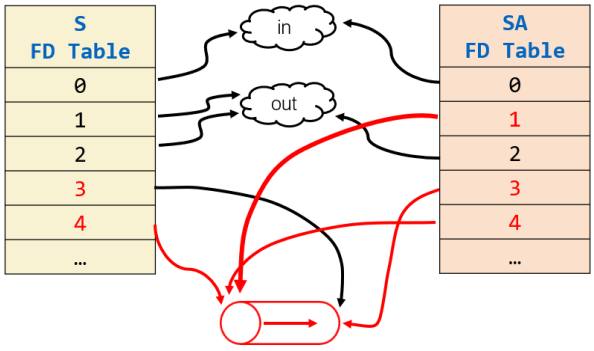
```
ls | tr a-z A-Z | wc
```

Example: connect two programs with a pipe

Start a pipeline in program S (aka, the shell): A | B

- High-level strategy (missing clean up!)
  - Create a pipe
  - Fork #1
    - In child process
      - Redirect stdout to the write end of the pipe
      - Start A, by calling exec
  - Fork #2

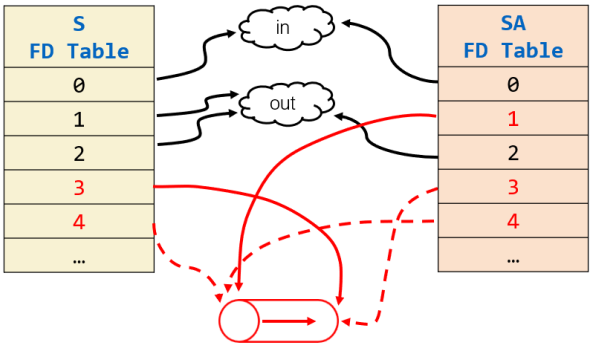
- In child process
  - Redirect stdin to the read end of the pipe
  - Start B, by calling exec

At the beginning • S has only 0, 1, and 2 open	
Pipe Creation • S creates a pipe by calling pipe() ○ A pair of FDs is returned	
Fork #1 • S: fork(0 ○ FD table is duplicated	
Redirect in first child process • SA: dup2(3, 1) ○ Or close(1); dup(4);	

CSE 3100 Master Notes

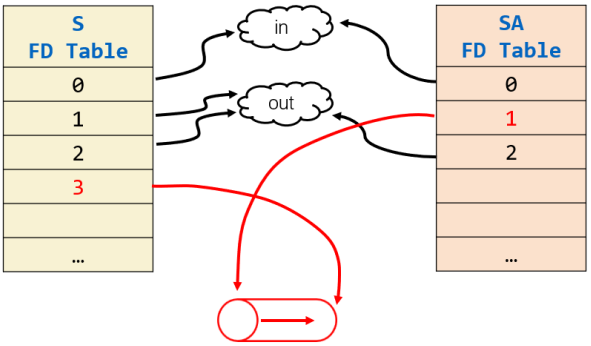
Clean up #1

- S: close(4)
- SA: close(4), close(3)
- SA can then exec into A



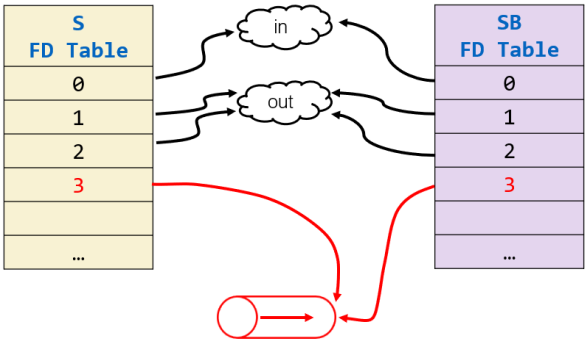
After clean up #1

- S: close(4)
- SA: close(4), close(3)
- SA can then exec into A



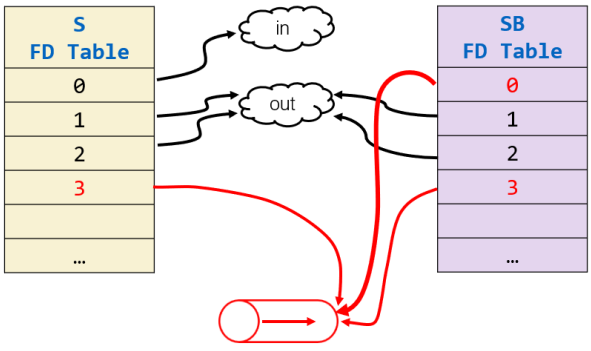
Fork #2

- S: fork()
  - Note that 4 has been closed in S

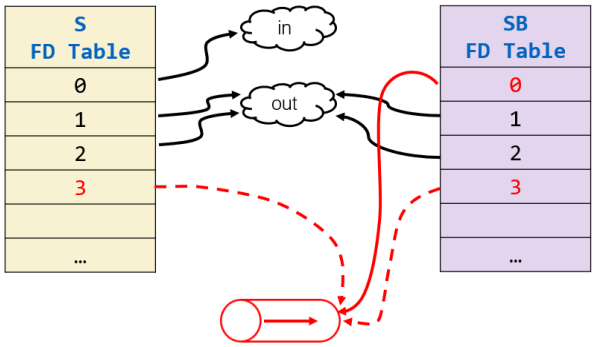


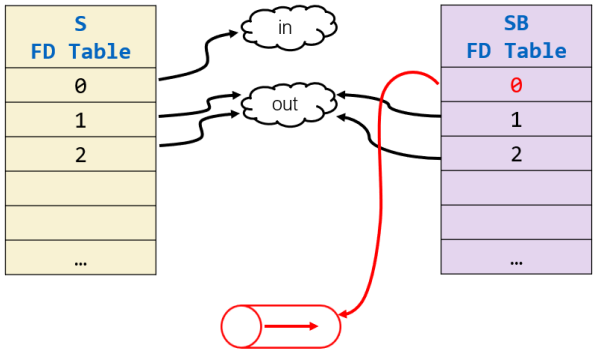
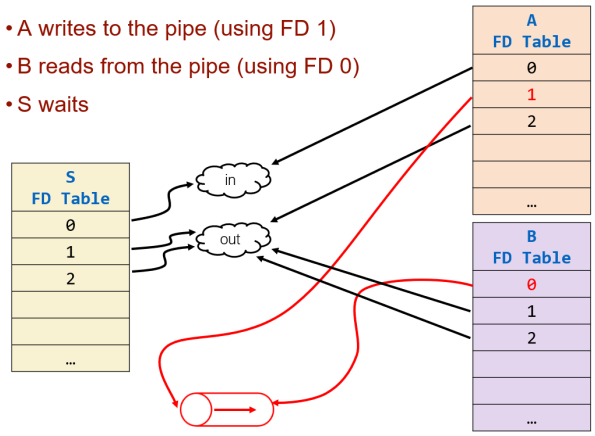
Redirect in second child process

- SB: dup2(3,0)
  - or close(0); dup(3);



<table>
<tr>
<td>Clean Up #2
• S: close(3)
• SB: close(3)

• SB can then exec into B</td>
<td> 
</td>
</tr>
<tr>
<td>After clean Up #2
• S: close(3)
• SB: close(3)

• SB can then exec into B</td>
<td> 
</td>
</tr>
<tr>
<td>Final set up
• A writes to the pipe (using FD 1)
• B reads from the pipe (using FD 0)
• S waits</td>
<td> 
</td>
</tr>
</table>

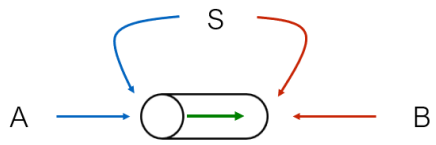
FDs of a dying process

- When a process ends, all its open FDs are automatically closed
- What happens to the processes on the other end of the pipe?

Example:

Assume S does not read or write, but have FDs of the pipe

- If both A and S die, B gets EOF when all buffered data are consumed
- If A dies, B will wait for more data (assuming S may write)
- If both B and S die, A gets an error (SIGPIPE) when writing
- If B dies, A will wait if the pipe is full (assuming S will read)



Close file descriptors
a process does not use!

Going further…

- You can repeat this to create a long pipeline
  - e.g., connect B's stdout to stdin of another process C
- Draw pictures to find how pipes are used
  - And what FDs need to be closed

Remember

- Processes are running in parallel once they are created
  - Although we showed the operations in sequence
- All processes in the pipeline are running concurrently on Linux
  - As soon as data are sent in the pipe…
  - The next process can pick them up on the work

Atomicity of read() and write()

```
n_r = read(fd, buf, N);
n_w = write(fd, buf, N);
```

- wite() and read() returns the # of bytes actually read/written
- The number maybe less than the requested

Atomicity - The degree a program is guaranteed to be isolated from other operations that may be happening at the same time

- Atomicity of write() is guaranteed if the # of bytes < PIPE_BUF
  - The bytes will be consecutive
  - The default value of PIPE_BUF is 4096 on Linx
- For read(), it is fine if all writes and reads are of the same size
  - Otherwise, need special handling

Starting a 2-stage pipeline - 1

```
// A | B

pipe(pipefd)          // pipefd is an array of 2 int's
pid_a = fork()              // for A
if (pid_a == 0){            // child process for A
      dup2();               // setup stdout for A
      close both FDs in pipefd
      exec to start A       // remember to exit from child on error
```

```
}
close(pipefd[WR_END]);      // No need to keep it open in parent
```

Starting a 2-stage pipeline - 2

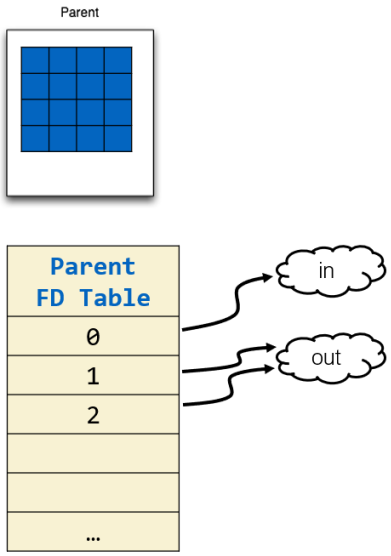
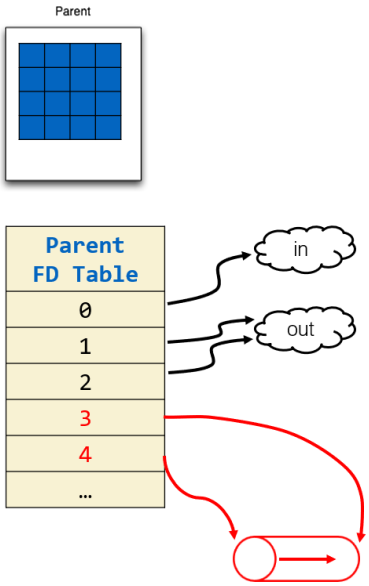
```
pid_b = fork()              // for B
if (pid_b == 0){            // child process for B
      dup2();               // setup stdin for B
      close(pipefd[RD_END]);
      exec to start B       // remember to exit from child on error
}
close(pipefd[RD_END]);      // No need to keep it open in parent
```

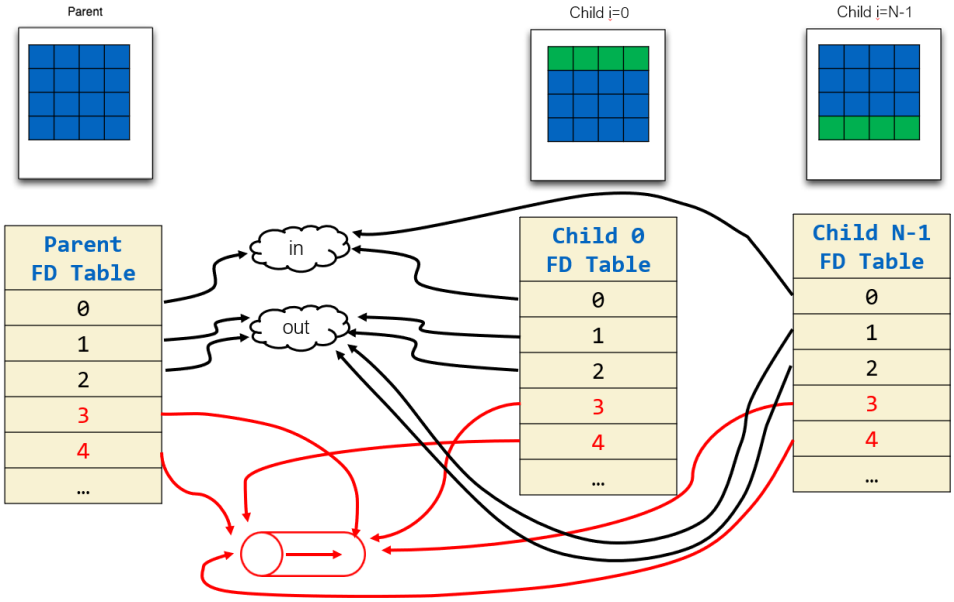
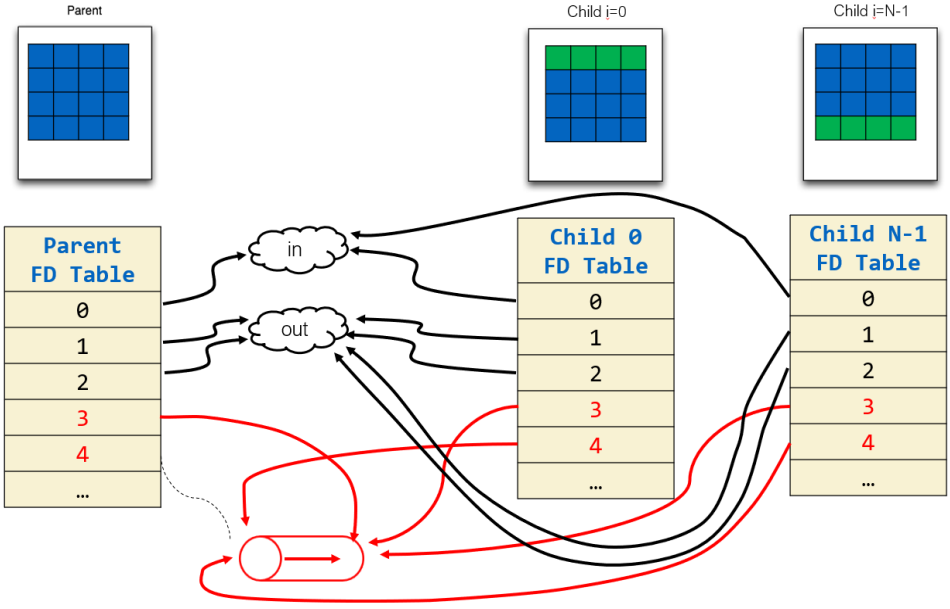
Using Pipes to Sum Matrix Rows Concurrently

- See the complete code in the demo repo.

```
int main(void)
{
   int i, row_sum, sum = 0, pd[2], a[N][N] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};

   if (pipe(pd) == -1) error_exit("pipe() failed");  /* create pipe */

   for (i = 0; i < N; ++i)
      if (fork() == 0) {            /* create a child process for each row */
         row_sum = add_vector(a[i]); /* compute the sum of a row */
         if (write(pd[1], &row_sum, sizeof(int)) == -1) /* write to pipe */
            error_exit("write() failed");
         return 0;                                    /* exit from child */
      }
   /* better to close the write end in the parent */
   for (i = 0; i < N; ++i) {
      if (read(pd[0], &row_sum, sizeof(int)) == -1) /* read from pipe */
         error_exit("read() failed");
      sum += row_sum;                     /* calculate the total */
   }
   printf("Sum of the array = %d\n", sum);
   /* wait for child processes*/
```

26

Parent Process	
Pipe Creation	

After forking (without cleanup)	
After forking (with cleanup)	

Lecture 20 - T1: Instruction & Basic Management

Mon. Oct. 21, 2019

Overview

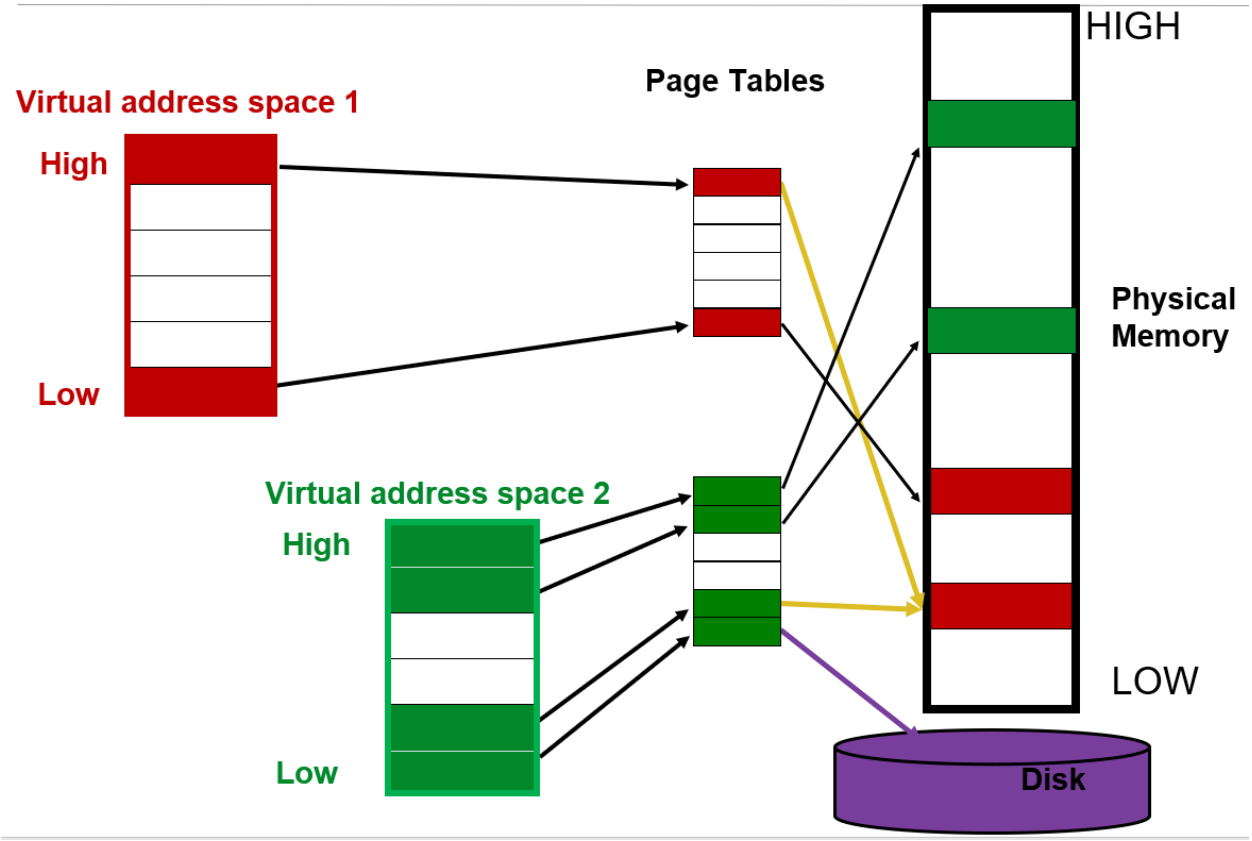
- Motivation & Concurrency

    - Thread abstraction
- Thread creation & termination

Motivation

- Processes for concurrency
    - Process do run concurrently on O.S.
        - On multi-core or single-core processors, through time-sharing
- Processes also provide protection

Each process has its own virtual address space



Why?

- Well…
    - Nobody can interfere
    - Nobody can move his “stuff”
    - Nobody can play with his “stuff”
    - Nobody can break his toys
- It is very very safe!

Downsides?

- Nobody to play with
    - You can get bored quite quickly
- Nobody to do your chores
    - Do everything yourself!
- You can't leave the castle
    - It is also your prison
- Communication with the outside is tricky and very limited (but safe)
    - Banging on a pipe…
    - Smoke signals…
    - Sockets…

Where are her buddies (e.g., processes)?

- In other castles!
    - Equally alone / isolated
    - You cannot get together easily
    - Sharing is limited

Threads?

- Inviting other 'living creatures" iinside your castle
    - These "creatures" can
        - Move around independently of you
        - Do work / chores on your behalf
        - Communicate on your behalf
        - Play with you
        - Use/share your toys
- Essentially
    - No limits on their abilities in their address space
    - As powerful as the "Lord of the Castle"

Refining our definitions

- Process is a bundle grouping
    - A virtual address [memory]
    - A collection of files / sockets [IO]
    - A collection of concurrent threads [execution units]
- Thread is a light-weight entity
    - Can run concurrently w/ other threads
    - Share resources in the process [having same rights]

- Confined to a single process [cannot “move” to another]
- Can be created and destroyed [different life cycles]

Lightweight ‘processes” that share the address space (and other resources in a process)

The Circle of Life

- How is a thread created
  - By another thread! `pthread_create()`
  - It is given a stack to execute and a function to run
- What about the 1st thread?
  - When a process is created, there is a single thread [starts alone]
- How does a thread die?
  - Voluntarily, after completing its task `pthread_exit()`
  - Requested by another thread `pthread_cancel()`
  - When the process dies (along with all thread in it)
    - Any thread calls exit()
      - Note that exit() is called when main() returns!

!!! Any thread can bring down the whole castle !!!

The pthreads API

- ANSI/IEEE POSIX 1003.1 - 1995 standard
- These types of routines:
  - Thread management: create, terminate, join, and detach
  - Mutexes: mutual exclusion, creating, destroying, locking, and unlocking mutexes
  - Condition variables: event driven synchronization

The Pthreads API naming convention

Routine Prefix	Function
pthread_	General pthread
pthread_attr_	Thread attributes
pthread_mutex_	mutex
pthread_mutexattr	Mutex attributes
pthread_cond_	Condition variables
pthread_condattr	Conditional variable attributes
pthread_key_	Thread specific data keys

The Pthreads API

```
# include <pthread.h>
```

- Add '-pthread' option to compile on Linux

```
cc -pthreads a.c
```

- PThread functions do not set errno on errors
- Many types are defined in pthread library. They are opaque objects
  - Cannot make assumptions on the representation/implementation

For ex.): should use pthreads_equal() to compare two thread IDs

main() function of thread

void * thread_main(void * arg)

- The "main" function of a thread
  - Can be any name you like
- Takes a pointer as the only parameter
  - It can point to anything, int, char, string, or a structure
- Return a pointer
  - It can point to anything, int, char, string, and a structure
  - **However, do not point to local variables on stack -**

Threads creation

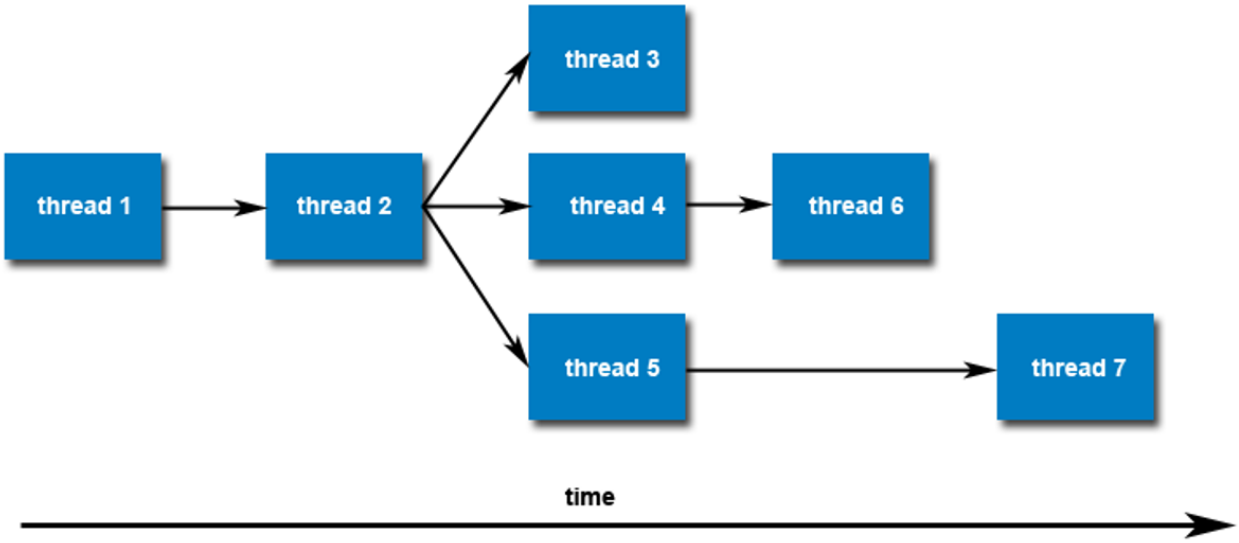
```
int pthread_create(
                pthread_t *  thread,
                pthread_attr_t * attr,
                void * (*start_routine)(void*),
                void * arg);
```

- Returns 0 if successful, and non-zero (> 0) if error
- *thread is the returned thread ID, if successful
- attr specifies the attribute for the thread. NULL for defauly
- start_routine()
- void * arg is passed to start_rountine()

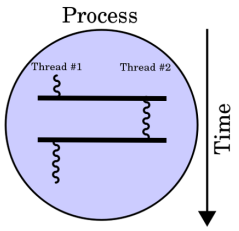
- Thread equivalent of fork(), but it does create (not "clone")

Thread creation

- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency
  - All threads are equal!

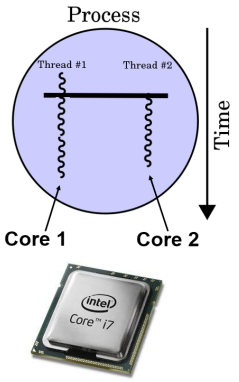


What about concurrency?



- Single-core CPU: Timesharing
  - When one thread is waiting for an IO to complete…
  - … another thread can use the CPu

What about thread concurrency?



- Multi core CPU: true concurrency
  - MIMD architecture: multiple instructions, multiple data
  - Threads can execute in parallel, one on each core
  - OS can still preempt threads
  - Useful when #threads >> #cores → timesharing is still used!

Thread Termination

- Return from the start_routine function, or
- Call pthread_exit()

```
void pthread_void(void* status)
```

- The function always succeeds and does not return
- Status can be obtained by other threads
- Similar to process termination
  - `main()` returns, or `exit()` called by any thread

Joining a thread

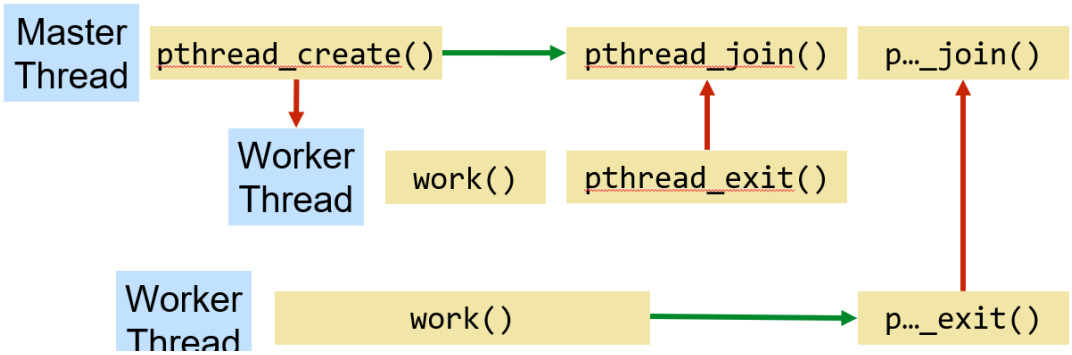
```
int pthread_join(pthread_t tid, void** status)
```

- Wait for a thread to complete
  - Blocks the calling thread until thread id terminates

- Can obtain the exit status of the thread
  - Pass NULL to ignore the return value
  - Why is the type of status void (**)? - A pointer to a pointer that can be changed
- Equivalent of **`waitpid()`** for processes

Joining a thread - 2

- Joining is a simple way to accomplish synchronization
  - The calling thread can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit()



Reading

- Book "Programming with POSIX Threads"
  - Chapter 1, 2, 3, 4,
  - Chapter 5, Section 1- 4
- Nice to read too
  - Chapter 6, Sectoins 1-5
  - Chapter 8 [debugging]
  - Chapter 9 [reference]
  -

Passing Arguments and Getting Results Back

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

#define NUM_THREADS   8

struct thread_data
{
   int  thread_num;
   char* message;
   int   len;
};

void* PrintHello(void* threadarg)
{
   struct thread_data* my_data = (struct thread_data*) threadarg;

   sleep(1 + 5*(my_data->thread_num % 2) );

   my_data->len = strlen(my_data->message);
   printf("Thread #%d: %s  length=%d\n", my_data->thread_num, my_data->message,
    my_data->len);

   pthread_exit(NULL);
}
```

```
int main(int argc, char* argv[])
{
   pthread_t threads[NUM_THREADS];
   struct thread_data thread_data_array[NUM_THREADS];
   char* messages[NUM_THREADS];
   int rc, t;

   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvuyte, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";
```

```
   for( t=0; t<NUM_THREADS; t++ ) {
      thread_data_array[t].thread_num = t;
      thread_data_array[t].message = messages[t];
      printf("Creating thread #%d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, &thread_data_array[t]);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   int grand_total = 0;
   for( t=0; t<NUM_THREADS; t++ ) {
      printf("Joining thread #%d\n", t);
      rc = pthread_join( threads[t], NULL );
      if( rc ){
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
      }
      grand_total += thread_data_array[t].len;
   }
   printf("Grand total = %d\n", grand_total);
   pthread_exit(NULL);
}
```

28

Common ways to use threads

- Pipeiline
    - A task is broken into a series of sub-operations, each of which is handled in series, but concurrently, by a different thread (think automobile assembly line)
- Manager/worker

    - A single thread, the manager assigns work to other threads, the workers. The manager handles all input and parcels out work to workers
    - Two common forms: static worker pool and dynamic worker pool
- Peer
    - Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work

Applications of threads

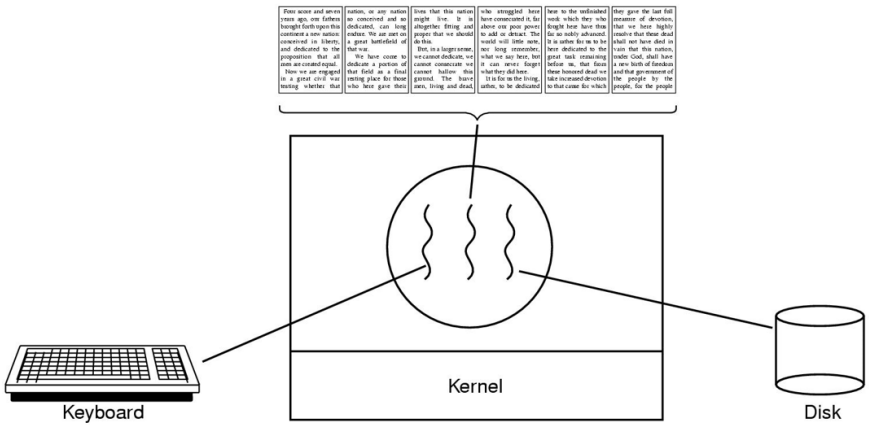
- Parallel computing
    - On multi-core machines, threads can be executed at the same time
- Overlap CPU work with I/O
    - While one thread is waiting for an I/O, others can perform CPU work
- Asynchronous event handling
    - e.g., a web server can both transfer data from previous requests and manage the arrival of new requests
- Priority/real-time scheduling
    - Important tasks can be scheduled with higher priority
- Computer games
    - Each thread controls the movement of an object

Why is it hard for the princess to do everything by herself?

- Imagine an application like Microsoft Word
- When you are tying, the process needs to
    - Listen for keystrokes and display the text on screen
    - Save data periodically to disk so you do not lose data in case of crash
    - Reform the document while edit the text
- How can a single process (i.e., one princess) can do all these concurrently?

The answer is she cannot without affecting application performance

We solve it by creating multiple threads!



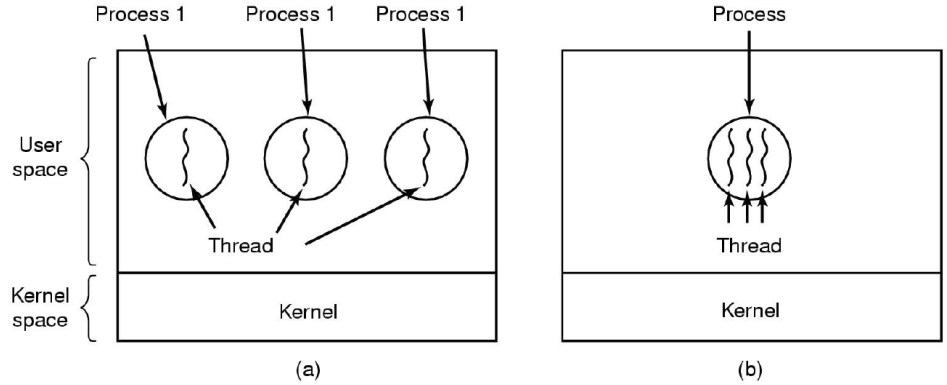
A word processor with three threads.

Usage Examples?

- We will create multiple threads, one thread for each sub tasks
  - One thread controls a GUI
  - One thread reformats the document
  - One thread does background tasks (e.g., savings!)
- Other examples
  - Worker threads do parallel computations
    - e.g., in matrix-vector multiplication: do all the rows in parallel
    - e.g., simulate agents in parallel [think agents in games!]

The Classical Thread Model(1)

a. Three processes each with one thread
b. One process with threads



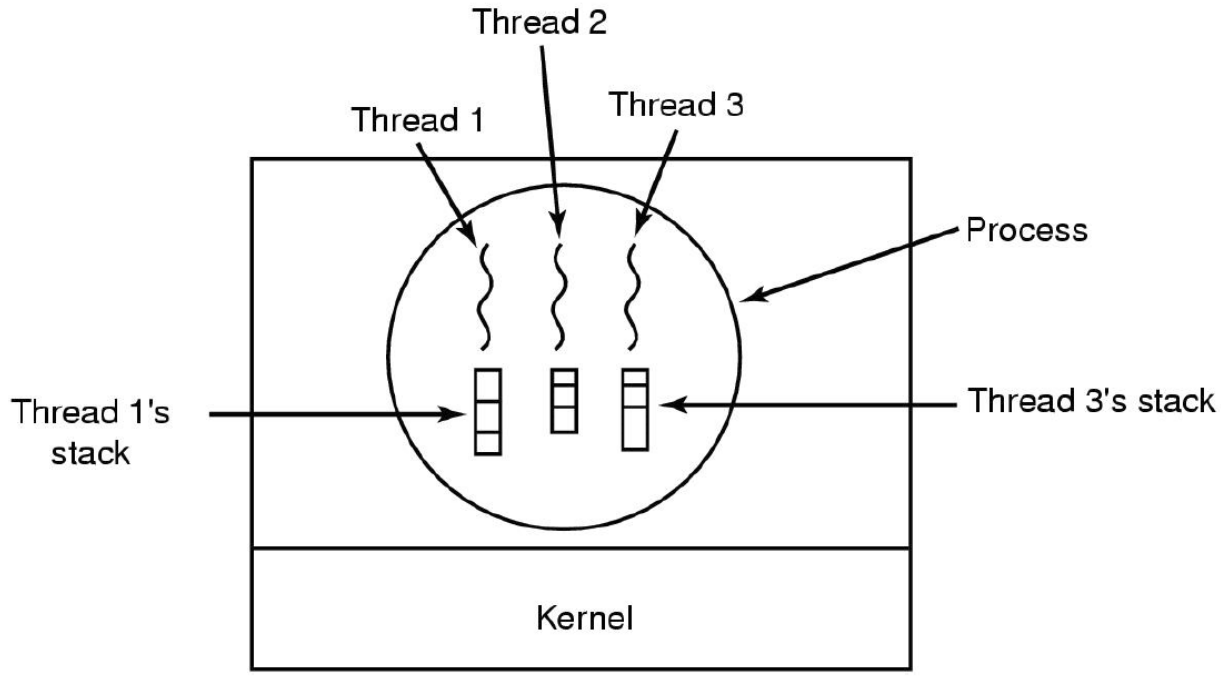
Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

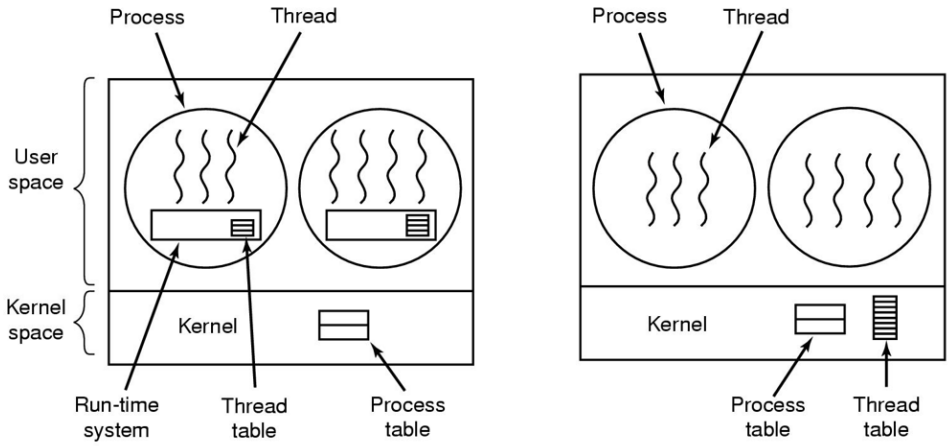
Shared by all threads in a process.

Private to each thread.

Classical Thread Model (3)



Implementing Threads



a. A user-level threads package
b. A threads package managed by the kernel

The Plot

- What you wish to share (the toy) is
    - The variable x
    - it exists somewhere in the shared virtual address space
    - It has no special status (compared to other regions!)
- The protagonist who both want to play with the toy are
    - thread 1
    - thread 2

Your Objective

- Protect the toy(x)
    - Make sure nothing “bad” happens to it
    - In particular, make sure it has the correct value at the end
- Recall
    - x exists in memory
    - So you wish to protect memory (an integer)

Idea

- You cannot protect the memory hold “x”
- But…
- You can specify a protocol for everyone using “x”
    - If a thread wishes to increase “x” …
        - It must grab a specific “lock” [lock]

        - If it has the “lock”, it can do what it wants to x [critical section]
        - When does with x it must release the “lock” [unlock]
    - If it does not have the lock, it must wait

You do not protect x directly
Instead, you discipline the code that touches x

Idea 1 - Is this slide necessary? - Maifi

- Put x in a “mini-prison”
    - If thread 1 wishes to increment “x”
        - It must take “x” out of its prison, increment it, and put it back
    - If thread 2 wishes to increment “x” and the cell is empty…
        - It must wait until “x’ is back in its cell!
- Does this work? [conceptually]
- Does this work? [practically]

32-bit Implication

- Virtual address Space size on 32-bit OS
    - Linux: 2G
    - Windows: 1G
- Typical stack size per thread
    - 8 Megs
- Memory usage goes to
    - Executable: ~ 1 to 50 megs
    - Heap: ~ 1 to 200 megs
    - Stacks: ~ # of threads * 8 → 100 threads yield 800 megs
- Total near the limit of the address space size.

Circumventing the limit?

- Several “ways” to hop along
    - Make smaller stacks! [but beware of recursion]
    - Separate tasks in several process that
        - Communicate via pipes
        - Communicate via shared virtual memory
    - Use a 64-bit OS!
        - Remember 8 megs = 2^{23}
        - Address space size = 2^{64}
            - What’s left is still: 2^{63}
        - How many stacks can you have? ~2^{40}

CSE 3100 Master Notes

Threads and MIMD Architectures

- **MIMD** - Multiple Instruction Multiple Data
- Process has
    - Multiple execution units
    - All executing independently
    - All executing diff. instructions
    - All operating on diff. pieces of data

Threads and MIMD Architecture

- Threads
    - Are OS abstractions that capture computation streams
    - Can be scheduled on a MIMD processor
    - All concurrent threads execute diff. instructions on diff. data

Timesharing

- Idea is simple
    - Diff. pattern of interaction [work / IO] for threads
    - When one thread is “waiting” for an IO to complete…
    - … another thread can use the CPU for some computing.
    - At any one point in time, CPU is used by only ONE thread
- True concurrency
    - There are > 1 CPUs
    - Threads are executing truly in parallel, one on each CPU
    - OS can still preempt threads
    - Useful when # threads >> # cpus! → time sharing is also used!

Threads?

- **definition [wikipedia]** - In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically as part of an operating system).

User Level Thread

- Can be implemented on top of OS that does not support threading
- Each process maintains thread table
- Per process run time system for thread switching
- Each process can have its own customized algorithm

- But
    - Blocking system call in one thread can be a problem

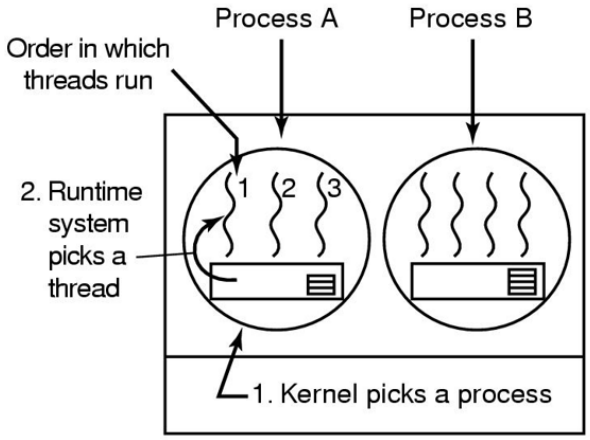
- Change all block to non-block call by changing OS
- Within a single process, there is no interrupt. If a thread does not give up voluntarily, other threads in the same will not get a chance

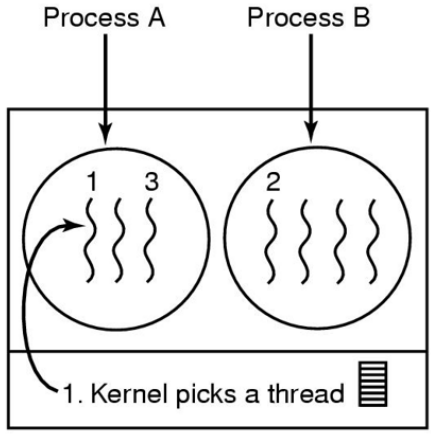
- Threads are mainly for processes that often blocks for I/O
  - User level thread does not make much sense as the goal was to avoid expensive kernel switch. Once switch is done, not much work to switch between threads

Kernel Level Thread

- Kernel has a thread table for all threads
  - Hold thread specific register, stack, state
- Thead recycling
  - Instead of deleting thread data structure, preserve it and reuse it
- But
  - What happens when multithreaded process forks?
  - Signals are per process. Which thread should receive the signal?

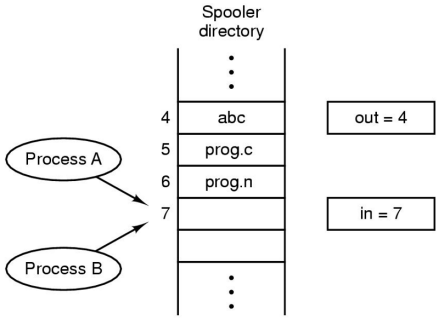
Let's check...





Threads & Castles

- In reality
  - When a process is created there is only 1 living creature (1 thread)
  - The first thread no diff. from other threads
  - All threads have equal rights (unless otherwise specified)



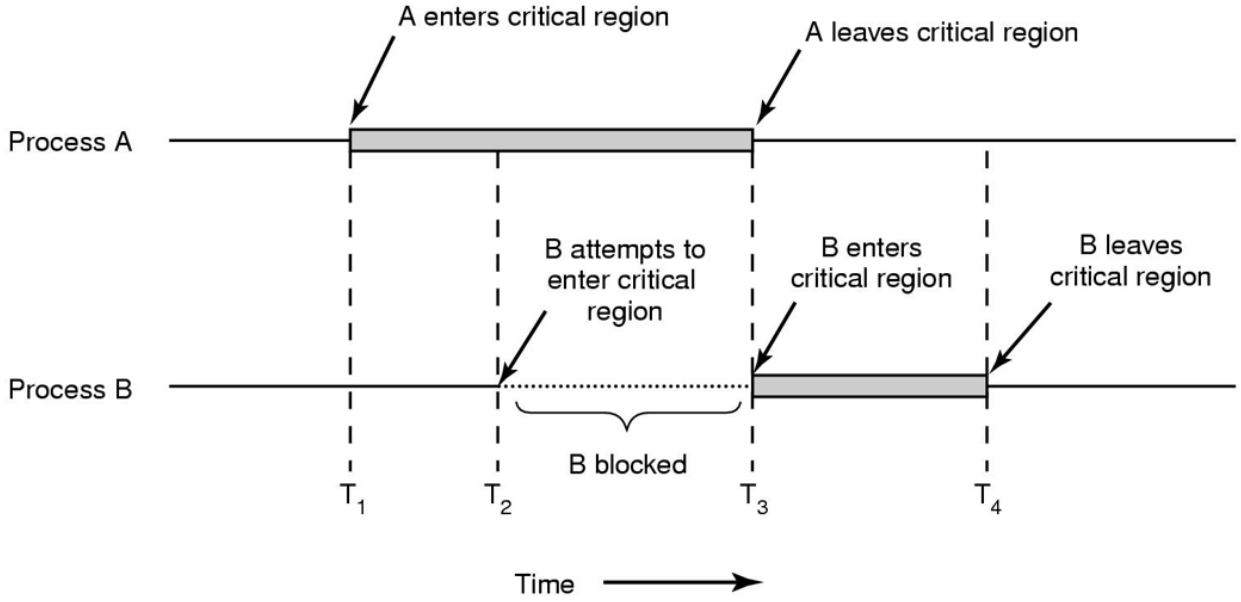
Race Conditions

- Two processes want to access shared memory at the same time

Critical Regions (1)

Conditions required to avoid race condition:

- No two threads may be simultaneously inside their critical regions
- No assumptions may be made about speed or the # of CPUs
- No threads running outside its critical region may block other processes
- No threads should have to wait forever to enter its critical region



Critical Regions (2)

- Mutual exclusion using critical regions.

Lecture 21 - T2: Resource Sharing

Weds. Oct. 23, 2019

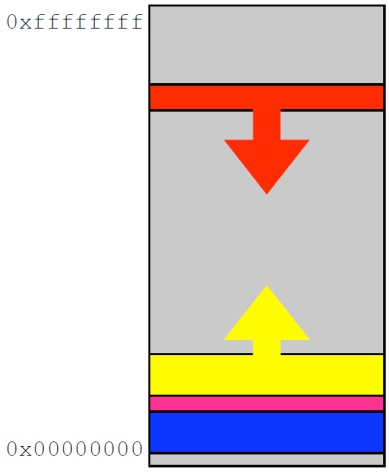
Review

```
#include <pthread.h>
// Compile and link with '-pthread'
```

```
int pthread_create(   pthread_t* thread,
                      pthread_attr_t* attr,
                      void* (*start_routine)(void*),
                      void* arg);
// Terminating itself or return from start_routine()
voidpthread_exit(void *retval);

// avoid zombie threads
int pthread_join(pthread_tthread, void ** retval);
```

Virtual Address Space

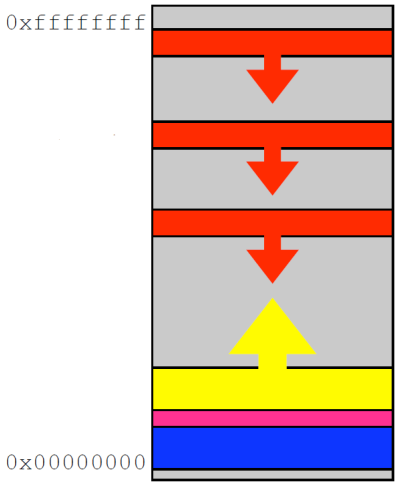
Process with one thread



Each thread gets a stack - Stacks have a max size -- kepss stacks separated to avoid accidental overlap

Process with 3 threads

- **Global data is shared**



32-bit Implication

- Virtual address Space size on 32-bit OS
  - Linux: 2G
  - Windows: 1G
- Typical stack size per thread
  - 8 Megs
- Memory usage goes to
  - Executable: ~ 1 to 50 megs
  - Heap: ~ 1 to 200 megs
  - Stacks: ~ # of threads * 8 → 100 threads yield 800 megs
- Total near the limit of the address space size.

Circumventing the limit?

- Several “ways” to hop along
  - Make smaller stacks! [but beware of recursion]
  - Separate tasks in several process that
    - Communicate via pipes
    - Communicate via shared virtual memory
  - Use a 64-bit OS!
    - Remember 8 megs = 2^{23}
    - Address space size = 2^{64}
      - What’s left is still: 2^{63}
    - How many stacks can you have? ~2^{40}

Advantages of Threads

- Simpler programming model
- Easier to coordinate (shared address space, and data)
- Lighter weight than processes
  - Takes less resources than process to manage
- In case of substantial CPU and I/O, thread improves performance

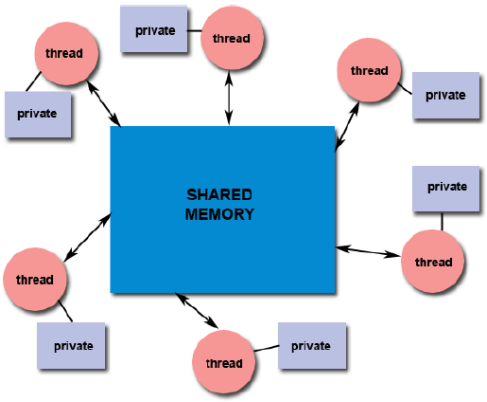
- Light-weight
  - Lower overhead for thread creation
  - Lower context switching overhead
  - Fewer OS resources

Time (sec.) for creating 50,000 processes/threads

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.8	27.7	1.8	0.7	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.1	48.6	47.2	2.0	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	55.0	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.3	0.7

https://computing.llnl.gov/tutorials/pthreads

- Shared State
  - Simpler programming model
  - Don't need IPC-like mechanism to communicate between threads



The larger, the better.

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Example: array sum

- Use two threads to compute the sum of integers in an array

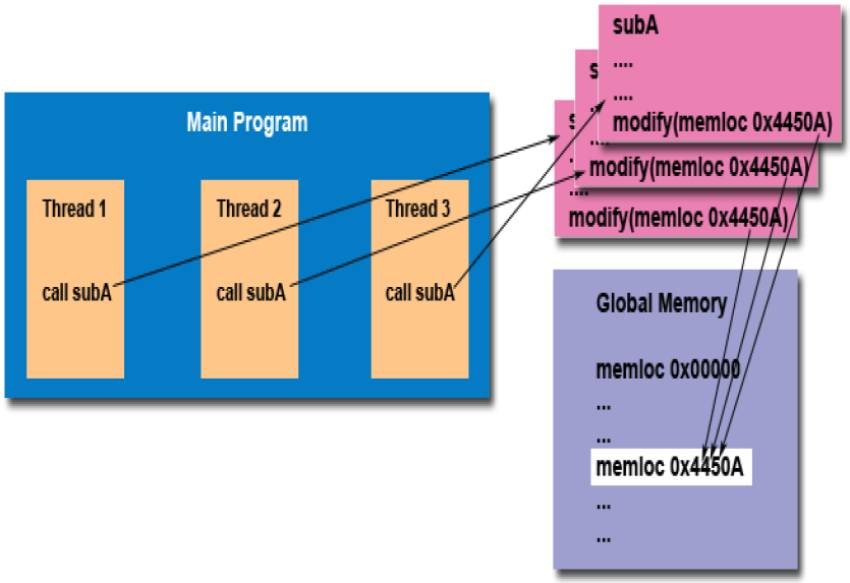
Disadvantages of threads: What if:

- A thread something stupid like
  - A division by zero?
  - Dereferencing a null pointer?
  - Corrupting a block of memory?

- Access a bad file descriptor?

The entire process crashes and burns!

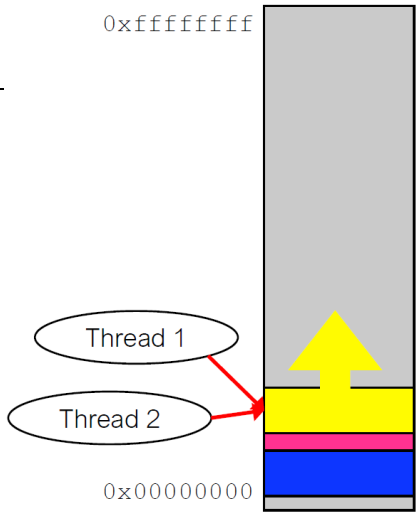
Disadvantages of threads: Shared State.



You may not be aware of shared resources!

Shared data: global, heap, and even local

```
int a[100];
// a, defined outside of functions,
// can be accessed in all threads.

int main(void)
{
      int i;      // stack
      char * p = malloc(1000); // heap
      void * arg = &i;
      // pass p or arg to threads
}
```



However, Sharing is Unsafe - A simple counting program…

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

long count = 0;

void* increase(void *arg){
    long i, inc = *(long *)arg;
    for (i=0; i<inc; i++)
        count++;
    pthread_exit(NULL);
}

int main(int argc, char* argv[]){
    pthread_t tid1, tid2;
    long inc = atol(argc >= 2 ? argv[1] : "100");
    pthread_create(&tid1, NULL, increase, &inc);
    pthread_create(&tid2, NULL, increase, &inc);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter is %ld\n", count);
    return 0;
}
```

Example

- Consider the two threads each doing the following

```
int x = 0;
```

```
void increase(int cnt) {
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

```
void increase(int cnt) {
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

What will happen?

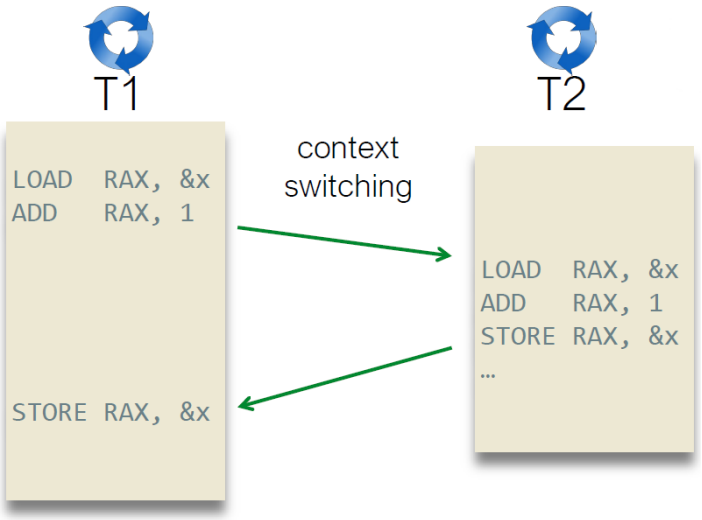
What is happening?

- The addition
  - Becomes more complex in assembly
  - Something like (pseudo-code)

```
void increase(int cnt)
{
   int i;
   for(i=0;i<cnt;i++)
      x = x + 1;
}
```

```
// x = x + 1;

LOAD RAX, &x
ADD  RAX, 1
STORE RAX, &x
```

- Threads execute this concurrently
- Even on a single core, the execution of a thread can be interrupted



Lesson

- Even sharing a single integer can go wrong!
- What to do?
- We need coordination!
    - RULES and PROTOCOLS
    - To establish how share data safely and keep everyone happy

The Road Ahead

- What we will do
    - Define PROTOCOLS and DATA STRUCTURES to safely share
- Examples
    - Mutexes / Spinlocks
    - Conditions
    - Semaphores
    - Barriers
    - Producer / Consumer
    - Reader / Writer
    - …

How unsafe can this be?

- Lots of subtle issues
  - It is very easy to get it wrong
- Good multi-threading programming must be disciplined

Detaching a thread

`int pthread_detach(pthread_t tid)`

- The “parent thread doesn’t need to wait
- A thread can detach another thread
- When detached thread terminates, its resources are automatically released
- A thread can detach itself:

  `pthread_detach(pthread_self());`
- Only threads created as joinable and not detached can be joined

Passing Arguments to Threads -1

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

#define NUM_THREADS  8

struct thread_data {
   int  thread_num;
   char* message;
};

void* PrintHello(void* threadarg) {
   struct thread_data* my_data = (struct thread_data*) threadarg;

   sleep(1 + 5*(my_data->thread_num % 2) );

   printf("Thread #%d: %s  length=%zd\n", my_data->thread_num, my_data->message,
    strlen(my_data->message));

   pthread_exit(NULL);
}
```

Passing Arguments to Threads -2

```
int main(int argc, char* argv[])
{
   pthread_t threads[NUM_THREADS];
   static struct thread_data thread_data_array[NUM_THREADS];
   char* messages[NUM_THREADS];
   int rc, t;
   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvuyte, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";

   // continue on the next slide
```

Passing Arguments to Threads -3

```
  for( t=0; t<NUM_THREADS; t++ ) {
      //set up struct for thread t
      thread_data_array[t].thread_num = t;
      thread_data_array[t].message = messages[t];

      printf("Creating thread #%d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello,
                  (void*)&thread_data_array[t]);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
      printf("Detaching thread #%d\n", t);
      rc = pthread_detach( threads[t] ); // detach a thread
      if( rc ){
         printf("ERROR; return code from pthread_detach() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Create a thread as detached

- When calling pthread_create(), one of the attributes defines whether the thread is joinable or detached

- By default (NULL attribute) threads are created as joinable

```
pthreads_ tid; void * arg     // thread id and argument
pthread_attr_t attr;          // an attribute variable

pthread_attr_init(&tattr);    // initialize with default attributes
// set detach state
pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &tattr, start_routine, arg);     // create thread
pthread_attr_destroy(&attr);                    // destroy the attribute
```

Practice Exam 2

Mon. Oct. 28, 2019

Practice Exam 2 Starter Code

```
//In this practice exam, we use multiple processes to simulate a word game named hangman.
//In the original hangman game, the number of guesses are limited and some graphical
display of a hangman
//is involved to indicate the progress of the game.
//We simplify the game to not to draw a hangman, and not limit the number of guesses.
//The simplified game works as follows.
//Player one chooses a word and indicates the length of the word
// and let the second player to guess the word. For example, the first player
// shows the following string to indicate that the word has 4 letters.
// ----
//The player two suggests a letter, for example, the letter 'e'.
//The player one responses by displaying the correct guess at the right places in the word.
//For example, player one tells player two the following string
// --ee
//Player two suggests another letter, a letter 'f' this time.
//The player one displays the same string
// --ee
//This is because the letter 'f' is not in the word.
//Player two suggests another letter 't'.
//The first player displays
// t-ee
//Then the second player suggests another letter 'r'.
//Player one displays
// tree
//Now since all the hidden letters are displayed. The game is over.
//In this practice exam, we need to use two processes to simulate the two players.
//Also, we need to use pipes for the communications between the two players.
//To be more specific, the child process is player one; the parent process is player two.
//A user will type guesses from the standard input to play the game.
```

```
// Search TODO to find the location where the code needs to be completed.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
#include <sys/wait.h>
#include <errno.h>
#include <ctype.h>

#define     PFD_READ    0
#define     PFD_WRITE   1

#define MAX_WORD_COUNT 60000            //we have less than 60000 words
#define MAX_WORD_LENGTH 80              //each word is less than 80 letters

void die(char *s)
{
   if (errno)
      perror(s);
   else
      fprintf(stderr, "Error: %s\n", s);
   exit(EXIT_FAILURE);
}

char words[MAX_WORD_COUNT][MAX_WORD_LENGTH];       //2-d array to hold all the
words
int count = 0;                  //number of words, initilized to 0

//read words from the file to the array words declared above
//also update the number of words (update variable count)
//We could have avoided using global variables. Try to revise it yourself.
void read_file_to_array(char *filename)
{
   FILE *fp;

   //open the file for reading
   fp = fopen(filename, "r");
   if(fp==NULL)
      die("Cannot open the word list file.");

   // TODO
   // make sure when each word is saved in the array words,
   // There is no white space in words, we can use fscanf().
   // We could also use fgets(). Need to remove '\n' at the end.
   fclose(fp);
```

```
}

// write a character to a pipe
void write_char(int pd, char value)
{
   if (write(pd, &value, sizeof(char)) != sizeof(char))
      die("write()");
}

// write a string to FD pd , add '\n' at the end
void write_word(int pd, char *word)
{
   size_t len = strlen(word);

   if (write(pd, word, len) != len)
      die("write()");
   write_char(pd, '\n');
}

// read a char from FD pd and save the result in *pc
// return the return value from read()
int read_char(int pd, char *pc)
{
   return read(pd, pc, sizeof(char));
}

// read a line from FD pd
//
void read_word(int pd, char buffer[], int sz)
{
   char c;
   int count = 0;

   while (read_char(pd, &c) > 0)
   {
      if (count >= sz)
         die("line is too long in read_word().");
      if (c == '\n') {
         buffer[count] = 0;
         return;
      }
      buffer[count ++] = c;
   }
   // could handle error better
   die("read() faild in read_word");
}
```

```
//check if the character guess is in the word
//if it is in the word, update the string so_far in the right places
//for example, if guess is 'e', so_far is "----", and word is 'tree'
//then so_far will be updated to be '--ee'
//if guess is 't', so_far is '--ee', and word is 'tree'
//then so_far will be updated to be 't-ee'
// Return value:
// 0:   guess is not in the word
// 1:   guess is in the word
int check_guess(char guess, char *so_far, const char *word)
{
   // TODO
   return 0;
}

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

   if(argc!= 2)
   {
      printf("Usage: %s seed\n", argv[0]);
      return -1;
   }
   int seed = atoi(argv[1]);
   assert(seed > 0);

   int pdp[2];
   //pipe creation
   if(pipe(pdp) == -1)
   {
      perror("Error.");
      return -1;
   }

   int pdc[2];
   //pipe creation
   if(pipe(pdc) == -1)
   {
      perror("Error.");
      return -1;
   }

   pid_t pid;
   pid = fork();
   if(pid == 0)
   {
      // TODO
```

```
    //  close some file descriptors

    // read the list in child process
    read_file_to_array("dict.txt");

    char *my_word;
    char so_far[MAX_WORD_LENGTH];

    srand(seed);
    my_word = words[rand() % count];
    fprintf(stderr, "Child: debugging: the word is %s\n", my_word);

    // Note that so_far should have enough space
    size_t  len = strlen(my_word);
    for(int i = 0; i<len; i++)
       so_far[i] = '-';
    so_far[len] = 0;

    //TODO
    //repeatly doing the following
    //      send so_far to parent
    //      receive a guess (a character) from parent
    //      exit from the loop if it was not successful
    //      check_guess
    //Do some clean up before exit from the process
    return 0;
}
else
{
    char guess;
    char so_far[MAX_WORD_LENGTH];

    // TODO
    // close some file descriptors
    // Then do the following in a loop:
    //      read a word from child
    //      print it to stdout
    //      if there is no '-', exit from the loop
    //      read a character from stdin until a ltter is found
    //      report error if EOF found.
    //      turn the character to lower case
    //      send it to child
    // close file descriptors before exit from the process

}

//wait for the child process to finish
```

```
    waitpid(pid, NULL, 0);
    return 0;
}

//below is a sample output
//it can also be found in the file sample-output.txt
/*
   ./hangman 8
   To help debugging: the word is pride
   -----
   e
   ----e
   a
   ----e
   t
   ----e
   i
   --i-e
   r
   -ri-e
   p
   pri-e
   d
   pride
   The word is pride.
   */
```

Lecture 23 - T2: Mutual Exclusion (mutex)

Weds. Oct. 30, 2019

Review

- So far
    - You learned how to create threads (pthread_create)
    - You learned how to terminate thread execution (pthread_exit)
    - You learned how to wait on threads (pthread_join)
- You can carry out independent computations with threads
- However, sharing is a problem
    - Remember the shared counter?

What if the shared counter is the balance on your bank account?

CSE 3100 Master Notes

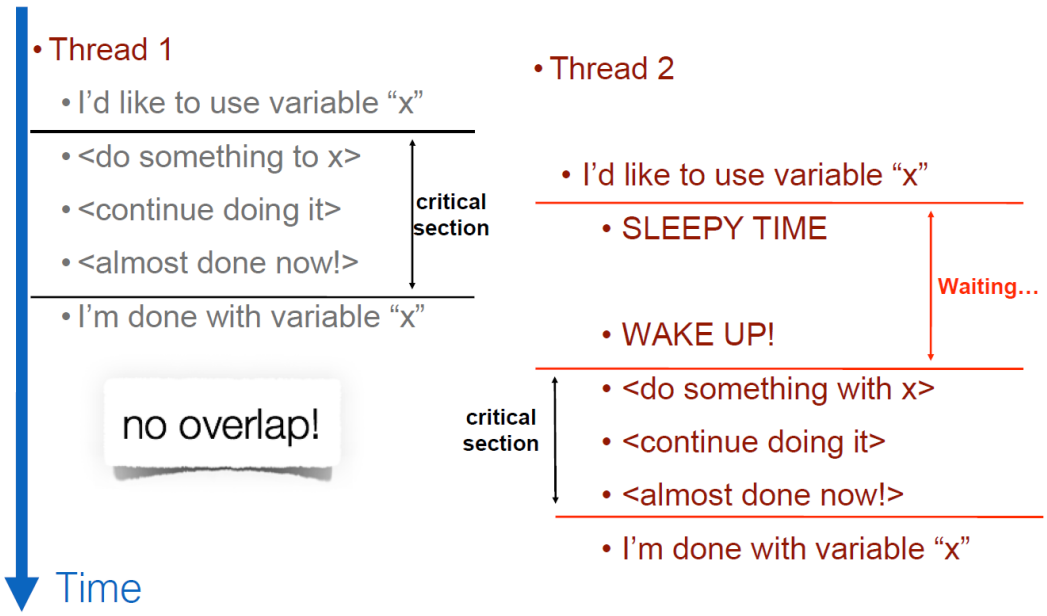
Overview

- Basic Sharing
  - Mutual exclusion
  - Critical selection
  - Mutex
- Application
  - Concurrent access to shared data structures
  - Counter arrays
  - ...

Mutual Exclusion

- Objective
  - Protect shared resources
  - Only one thread can access the resources
- Protocol
  - A set procedures for accessing shared resources
- Example:
  - Lock the resource while using it. Cannot lock if it is already locked
  - Wait if the resource is already lock
- Fact
  - Ask everyone nicely, and expect that everyone behaves

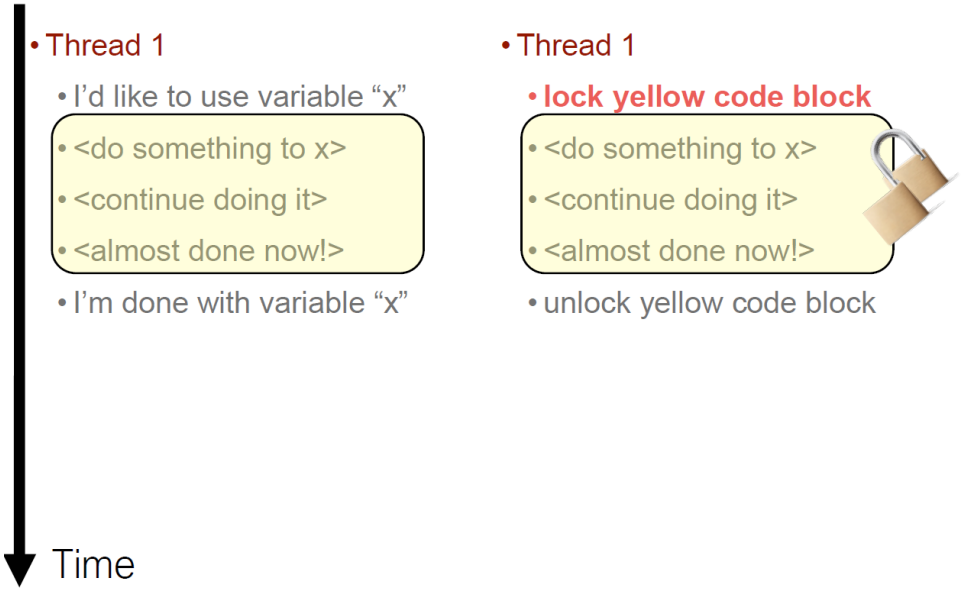
Protocol



Race to the critical section

- Multiple threads race each other to get to critical section
  - Critical section is a code segment that accesses shared resources
- One of them “wins” the race
  - Winner
    - acquire a “lock” first
    - executes critical section
    - release the lock (unlock)
  - Loser
    - waits as the resource is locked
    - wakes up when it is its turn to get the lock
    - executes critical section
    - releases the lock

Protocol



Mutex

- The Lock is a POSIX abstraction
  - Called a **Mutex** (for **MUT**ual **Ex**clusion)
  - Provided by the operating system
- Semantics
  - At most one thread can acquire the lock at any time
  - If a thread “loses” a race, it **falls asleep**
  - Sleepy threads **wake up** when the lock is released

pthread Mutex types and API

```
pthread_mutex_t; // define a mutex
```

- Functions
  - Initialize the Mutex
  - Destroy a Mutex we no longer need
  - Lock a Mutex
  - Unlock a Mutex
  - [Attempt to Lock a Mutex]

Creation

```
# include <pthread.h>
int pthread_mutex_init(pthread_mutex_t * mutex,
                       const pthread_mutexattr_t * attr);
```

- Initialize a mutex (i.e., allocate OS resources
- Return 0 on success

- Example

```
typedef struct MyRecord {
    pthread_mutex_t  myLock;
    int              myValue;
} MyRec;

MyRec* makeARecord() {
   MyRec* rec = (MyRec*)malloc(sizeof(MyRec));
   pthread_mutex_init(&rec->myLock,NULL);
   rec->myValue = 0;
   return rec;
}
```

mutex attribute

Destruction

```
# include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

- Release resources for lock
- Return 0 on successes

- Example

```
typedef struct MyRecord {
    pthread_mutex_t  myLock;
    int              myValue;
} MyRec;

void freeARecord(MyRec* rec) {
   pthread_mutex_destroy(&rec->myLock);
   free(rec);
}
```

Lock / Unlock

```
# include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

- Enter/leave the critical sectoin. Wait (asleep!) if mutex is locked
- Return Value 0 on success (=/= 0? something went horribly wrong)

- Example

```
typedef struct MyRecord {
    pthread_mutex_t  myLock;
    int              myValue;
} MyRec;

void incrementRecordValue(MyRec* rec) {
   pthread_mutex_lock(&rec->myLock);
   rec->myValue = rec->myValue + 1;
   pthread_mutex_unlock(&rec->myLock);
}
```

Key Observations

- Lock and unlock some pairs
- You need ONE MUTEX per “thing” you wish to protect
    - That’s why we package the integer & the mutex in a struct
    - The mutex protects just that integer
    - And nothing else!
- Lock is **BLOCKING**
- Critical sections should be **short**
    - Why?
- Locking incurs a **big** cost
    - Why?

Example 1

- Fix the example we had issue

Example 2: Counter ADT

- A shared counter
  - That was the driving example
  - Make an ADT pairing mutex and counter
  - Have functions to manipulate the ADT
    - increase
    - decrease
    - reset to Zero

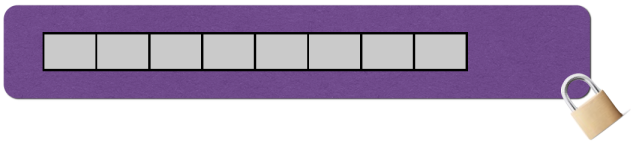
See the Demo Code under t3.counter-adt

Example 3: array

- Sharing an array
- Key decision: Locking Granularity
  - Lock the entire structure?
  - Lock individual parts?

Option 1

- Idea
  - Wrap up the array in a structure
  - Have a single lock for the whole thing
- Issues?



Option 2

- Idea
  - Wrap up each value in a structure (value + lock)
  - Make an array of structures
- Issues?

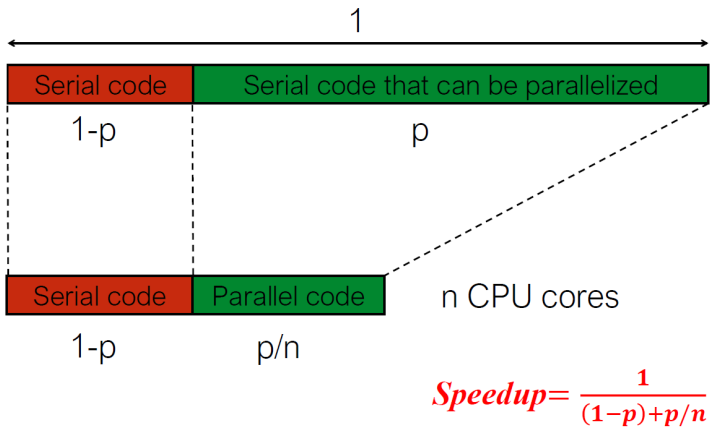
mutex_trylock()

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  - If the mutex is not locked, it will succeed and lock

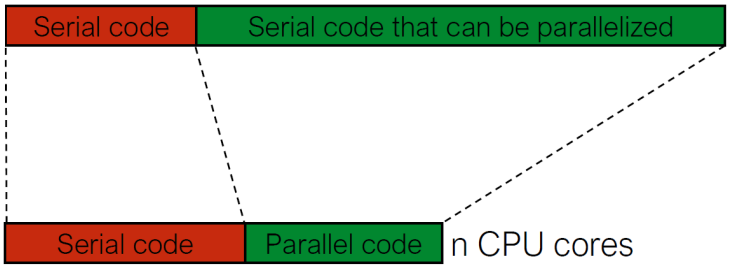
    - If the mutex is already locked
        - It will NOT block
        - It will return an error code
- Purpose
    - Mix breed. It allows polling before locking
        - sometimes, it is fine to use the shared resource later

Amdahl's Law



$$Speedup = \frac{1}{(1-p)+p/n}$$

Practical speedup lower due to overhead



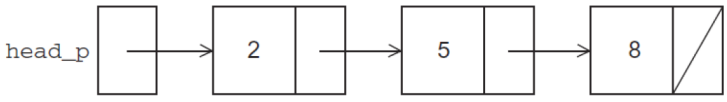
Mutex Attributes?

- A value of type pthread_mutexattr_t
    - An optimal argument to create create a mutex
- APIs to
    - Initialize attribute record (pthread_mutex_attr_init())
    - destroy attribute record (pthread_mutexattr_destroy())
    - modify an attribute in attribute record (set type, get type, etc.)
- Key property: mutex TYPE
    - How to deal with recursive lock?
    - What if other threads try to unlock?

Mutex Type (from the DOC)

- PTHREAD_MUTEX_NORMAL
  - This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behaviour. Attempting to unlock an unlocked mutex results in undefined behaviour.
- PTHREAD_MUTEX_ERRORCHECK
  - This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
- PTHREAD_MUTEX_RECURSIVE
  - A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
- PTHREAD_MUTEX_DEFAULT
  - Attempting to recursively lock a mutex of this type results in undefined behaviour. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behaviour. Attempting to unlock a mutex of this type which is not locked results in undefined behaviour. An implementation is allowed to map this mutex to one of the other mutex types.

Sorted Linked List Example

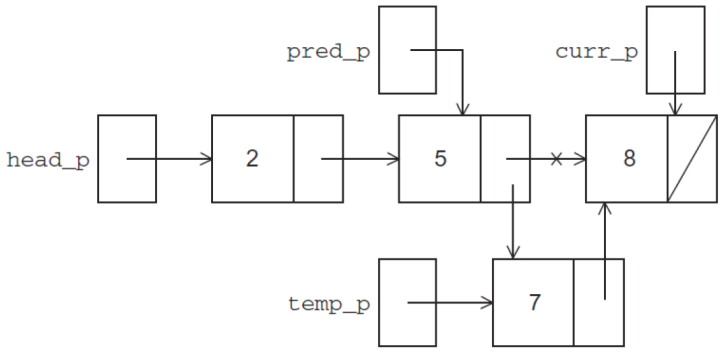



```
typedef struct list_node_s {
   int data;
   struct list_node_s* next;
} LNode;
```

Testing Membership

```
int member( int value, LNode* head_p) {
   LNode* curr_p = head_p;

   while( curr_p != NULL && curr_p->data < value )
      curr_p = curr_p->next;

   if( curr_p == NULL || curr_p->data > value )
      return 0;
   else
      return 1;
}
```

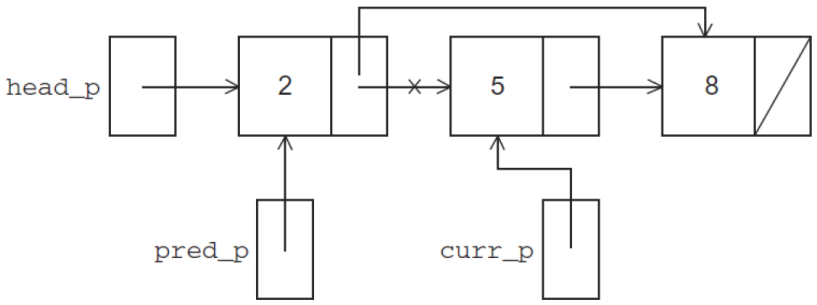
Inserting a new value



Inserting a new value

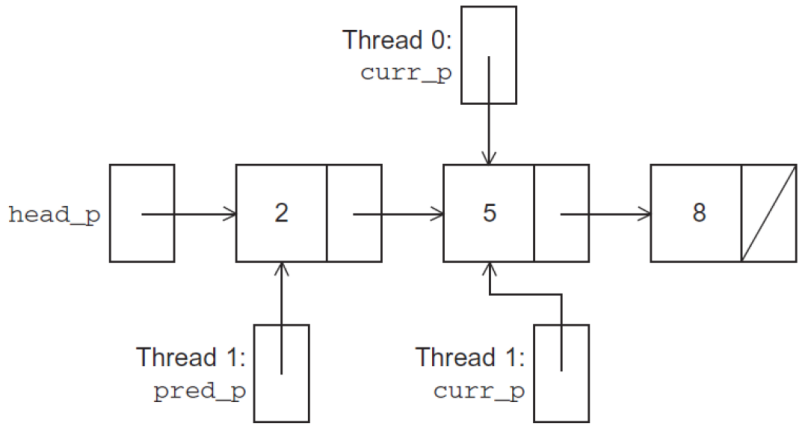
```
int insert( int value, LNode** head_pp) {
   LNode* curr_p = *head_pp;
   LNode* pred_p = NULL;
   LNode* temp_p;

   while( curr_p != NULL && curr_p->data < value ) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }
   if( curr_p == NULL || curr_p->data > value ) {
      temp_p = (LNode*)malloc( sizeof(LNode) );
      temp_p = value;
      temp_p = curr_p;
      if(pred_p == NULL)  /* new first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else  /* value already in list */
      return 0;
}
```

Deleting a value




```
int delete( int value, LNode** head_pp) {
   LNode* curr_p = *head_pp;
   LNode* pred_p = NULL;

   while( curr_p != NULL && curr_p->data < value ) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }
   if( curr_p == NULL || curr_p->data > value ) {
      if(pred_p == NULL) {  /* deleting first node in list */
         *head_pp = curr_p->next;
      } else {
         pred_p->next = curr_p->next;
      }
      free(curr_p);
      return 1;
   } else  /* value not in list */
      return 0;
}
```

Simultaneous access by two threads?



Solution #1

- Lock the entire list any time attempts to access it
  - A call to each of the three functions protected by a mutex e.g.,

```
pthread_mutex_lock(&list_mutex);
member( value, head_p);
pthread_mutex_unlock(&list_mutex);
```

- Issues?

Solution #2

- Instead of locking the entire list, we cold try to lock individual nodes
  - A “finer-grained” approach

```
typedef struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
} LNode;
```

Implementation of member with one mutex per node

```
int member( int value, LNode* head_p) {
  LNode* curr_p;

  pthread_mutex_lock( &head_p_mutex );
  curr_p = head_p
  while( curr_p != NULL && curr_p->data < value ) {
    if( curr_p->next != NULL )
      pthread_mutex_lock( &(curr_p->next->mutex) );
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
    pthread_mutex_unlock( &(curr_p->mutex) );
    curr_p = curr_p->next;
     }
```

```
  if( curr_p == NULL || curr_p->data > data ) {
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
    if( curr_p != NULL )
      pthread_mutex_unlock( &(curr_p->mutex) );
    return 0;
  } else{
    if( curr_p == head_p )
      pthread_mutex_unlock( &head_p_mutex );
      pthread_mutex_unlock( &(curr_p->mutex) );
    return 1;
  }
}
```

Issues

- More complex to implement
- More memory to store the list
    - One mutex per node
- Slower
    - Each time a node is accessed, a mutex must be locked and unlocked
- Using more than on lock creates opportunities for **DEADLOCK**!