CSI 400: Lecture 08

Synchronization and Buffering
(data not ready indication).

non-blocking I/O is an advanced topic—the system call might return with a

blocking operations are one mechanism of synchronization

returning

wait for the input before copying it into the calling process (memory) and then

For example a (blocking) read() system call to obtain terminal input MUST

order (in time) consistent with their dependencies.

synchronization: The need or mechanisms for doing computation steps in an

hardware, and/or time-shared on common hardware.

each other. They might be running actually at the same time on separate

appears as active simultaneously (typically) interacting or communicating with

concurrency: Multiple "tasks", "processes", "threads", "activities", etc.

University at Albany Computer Science Dept.
supports data transfer in both directions. Cache: denotes a similar idea; the difference is that a so-called cache usually
in RAM (to receive and hold the data before it is requested.

The buffering mechanism uses extra storage, called a buffer (typically an array and/or data is available before a dependent operation begins or request the data.

and/or systems are (often radically) different.

Synchronization (of course) when the speeds of individual communicating

Buffering: a mechanism for improving overall system speed while maintaining
The data from or to the disk (via the interface controller),

The kernel allocates and uses kernel memory buffers to receive, hold or transmit

subroutines (typically).

Is typically controlled by software device drivers which are kernel

CPU’s activity: running machine language code.

Computer transfers the data directly to/from RAM concurrently with the

interface hardware (typically IDE or SCSI controller) makes the

at a time.

Transfers data in batches of one or more sectors (typically 512 bytes/sector).

Takes a long time to start (several milliseconds),

(1) Disk hardware operation

Buffering Examples,
to process a substring based on how that substring starts.

In this application, the buffering allows the scanner to "back up" and choose how your code scans the input line character by character.

*String object* that receives each line of input (using `getLine()`, e.g.) before

You (C/C++/Java application programmer) can declare an array of chars (or a

Buffering Example (2):
Unix terminal system. See Haviland ch. 9.

Separation between the "line discipline" and "device driver" layers of the

and/or other configurable conditions are met (eg, MIN and TIME settings).

If buffers input characters and makes read() is blocked until a newline is received

The terminal driver is within the kernel:

Buffering Example (3):
process, except for the first process or for forks that fail.  

The Unix fork() system call creates a new process. Each new process is created 
call instruction at a particular time.  

System calls in the dynamic sense is the action of a process executing a system 
that cause interrupts.

Machine instructions (such as int 0x80 on x86/Linux) whose use can be coded in programs. System calls are coded by "syscall" type.

Similarly a system call in the static sense is a kind of operation or subroutine 
without an output operation (assuming the program runs to completion without 

38 dynamic instances of the 

When this program is compiled, linked, loaded into a process's virtual memory 

This program has ONE static instance of a certain "output statement". This 

```c
{
    while(c--){
        cout << "Hi" << endl;
        int c = 38;
    }
    main
```
(virtual) memory

sharing the same execution, each using a separate set of register values, all separate threads mean separate sequences of machine instruction

In general, separate threads mean separate sequences of machine instruction processes.

the entity that is scheduled. Concurrent systems support multi-threaded Linux and some other systems, the word “process” signifies the memory (and processes run the first time). immediately before the kernel’s process switcher makes the process run again (or)

including the PC (contains) is copied back into the CPU (hardware) registers

register data belongs to the entity that the kernel schedules. The register data

fetches or stored when the process runs) together with register data. The

addresses into segments, and the data stored at every legal address which can be

As such, a process has its virtual memory resource (organization of legal

We introduced process: One run of one program from start to finish, including
In Linux, a process is called a "task". A process is created by a common system call named `clone()`. A clone()’s parameters determine whether or not the new scheduling entity "task" shares its virtual memory with the parent process.

In the Linux kernel, the entities that are scheduled are called tasks. The "process control block" (generic term used in AOS) is realized by the Linux system call `getrusage()`. You can find it in

```
/usr/src/linux/include/linux-versions.h
```

The structure of a task is defined here.
Parent and child execute concurrently.

blocks, **or**

- Parent waits until child runs and exits (Unix system) function, parent
- Request others, after the child is created.

*Alternative:* Parent and child separately release some resources and

create a new thread.

(e.g. Linux `clone`(): one resource is virtual memory, so clone()) can

- Arguments to the process creator call `special` which resources are shared.

- Executable file is an argument to the process creator system call.

Variations:

- **Other Variations:**

  - memory with a **CO** of the parent process' virtual memory
  - `malloc` initializes the new process' virtual
    - Unix innovation (late 70's): `fork()`.
    - Special kernel code.

How did the first process get created?

- **Special kernel code**
- called that system call `parent`hood defines the tree of processes.
  - A process created by a `system` call must have a `parent` the process that
  - `fork`, `clone` does it. (Linux has the more general `clone`.)

Typically a process is created by a `system` call for that purpose.
In the background, by defaulting the command with & etc., you can
make this happen by commanding the job run
under Unix shells. Under Unix shells, you can make this happen by commanding the job run
in the background.

5. Or, the shell continues to execute concurrently with your process.

is blocked (your process is running in the foreground).

Typically, your process interacts with you via the terminal while the shell
receives a signal, typically caused by special characters like control-c or control-z.

4. The shell might block until your program's process finishes (or it receives a
somewhat specialized by you). The shell calls the system call(s) to create a new process to run the program
named for redirection, etc., and then,
based on the command line arguments (e.g., command line args.), opens

Unix/Linux desktops, etc.)

2. The shell scans, checks, gathers arguments (e.g., command line args.), opens
courses, MS-DOS/DOS shell, MS Windows, Macintosh,

Interactive shells can be text terminal (A) and non-Unix-based C shell

Interactive shell program.

I. When you log in, the "log in system" creates a process that runs an

How do ordinary people get processes created for them?
Using a shell to make the computer run several of your programs concurrently:

```
$ a.out
$ myproject.cpp
$ g++ myproject.cpp
$ Edit out bugs, save myproject.cpp. Then interact with your program... your process actually!
$ a.out
$ myproject.cpp
$ g++ myproject.cpp
$ emacs myproject.cpp
$ a.out
```
You can give new commands now.
{ 
    return gg;
    cout << "done!" >> endl;
    cout >> [1]argv >> " argument" >> [0]argv >> " argument"
    (argv < 1)
    cout >> [0]argv >> " running " >> argv[0] >> " process with pid=" >> getpid() () << "
    }

main(int argc, char *argv[])
{
    using namespace std;
    <unistd.h>
    #include <sys/types.h>
    #include <iostream>
    #include <sstream>
}
Happy Part 1 Days

Part 2 Days is done!

Happy Part 2 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!

Happy Part 1 Days is done!
{ return 0;
{
  cout >> endl;

  if (arc < 1) cout >> " on " >> arcv[1];
  if (arc < 1) cout >> arcv[0];
}
while (1)
} 

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

/* NEVER RETURNS */
exit(0);
print("child complete");
wait(NULL);
/* parent process */ }
else
{ }

/* NEVER RETURNS */
exit(-1);
printf("fork failed");
/* error occurred */ } (0 > pid)
/* fork() never has an argument!*/
if (pid = fork())
/* fork another process */
pid = pid -1 ?

(void) main(int argc, char *argv[])
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

unix fork and exec READ Haveland Chapter 5.
University at Albany Computer Science Dept.