Week 1

Memory

- Memory is about addresses and values.
    - Address is the location in memory of the value.
    - Values are an arbitrary number of the bits.
    - The first bit of a value is stored at the address.

`ssize_t` and `size_t`

`ssize_t` is longest signed integer (`int` or `long`), `size_t` is longest unsigned integer (`unsigned int` or `unsigned long`), their size depend on the machine.

Week 2

Pointer

Strings

> Strings resemble an array of characters, in C, all strings are **Null-terminated**, which means it ends with '\0'.

Pointer Notation

> A pointer is a memory address, find out the address of a variable using the **&** operator.

- Address operator, **&**.
- Indirection operator, *****, is used to unravel the indirection.

Pointers and arrays

The name of the array can be use to obtain the address of the first element of the array.

```
char msg[] = "Hello";
char *str = &msg[0];
//same as
char *str = msg;
```

> a == &a[0]

Pointers arithmetic

```
char *string;
string+=1; //jump to 1 byte after string
int *integer;
integer+=1;//jump to 4 byte after integer
```

`void*` can be use to point at **any type** when is used as a return type or a function arguments.

Keyword const

```
const char *fileheader = "P1";
fileheader[1] = '3';//Can't change the value.

char * const fileheader = "P1";
fileheader = "P3";//Can't change the address value.
```

Floating point types

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, e_{min} \leq e \leq e_{max}$$

Sign	meaning
s	sign
b	base of exponent
e	exponent
p	precision
f_k	nonnegative integer less than b

Enumerated types

> Is a Simple type, it is mapped to and similar to `int`, it associates a name with a value.

```
enum day_name
{
    Sun, Mon, Tue, Wed, Thu, Fri, Sat, day_undef
};//Maps to integers 0 ... 7, can do things like 'Sun++'
```

Week 3

Structure

> Is used to hold a collection of data items of different types.

```
struct date {
    enum day_name day;
    int day_num;
    enum month_name month;
    int year;
} Big_day = {
    Mon, 8, Jan, 2000
};//Declaration after definition

struct date dealine = {Mon, 8, Jan, 2000}; //Conventional declaration
```

- The name of the structure can be omitted.

Accessing Elements of a structure

- A dot used to nominate an element of the structure.

  ```
  struct date bigday;
  int theyear;
  bigday.year = theyear;
  ```

- The `->` operator indicates the element from a pointer of a structure.

  ```
  struct date bigday;
  struct date *mydate;
  int theyear;
  mydate = &bigday;
  mydate->year = theyear;
  ```

Keyword Typedef

> Give a nickname to a structure.

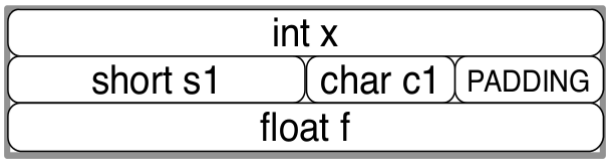
```
typedef struct date {
    enum day_name day;
    int day_num;
    enum month_name month;
    int year;
} Date;

struct Date dealine = {Mon, 8, Jan, 2000};
```

Size of structure

Address of a structure variable will give direct access to bytes of the first members.

Byte aligned should be considered.

```
struct c{
    int x;//4 bytes
    short s1;//2 bytes
    char c1;//1 bytes
    float f;//4 bytes
}
```



Unions

> Can be used for variables that wouldn't appear at the same time.

```
union test
{
    int a;
    double b;
    char c[9];
}; //size of this union is 16
```

- The size of a union will be the size of the biggest element, **byte aligned** should be considered.
- The name of the union can be omitted.

When using `test.a`, the value of b or c is unknown, union always keep the latest modified value.

Bit fields

> Can specify a size, in bits, for elements of a **structure**.
>
> The size is placed after the field name with a colon between.

```
struct tryBitFields
{
    unsigned a: 1;
    unsigned b: 8;
    unsigned c: 3;
}
```

Byte aligned should be considered but different from the normal structure type. For the example above, it will be $1 + 8 + 3 + padding(4) = 16\ bits$

Bit Operations

Operations	Code
shift right	>>
shift left	<<
bitwise AND	&
bitwise OR	|
bitwise XOR	^
bitwise NOT	~

Files

Function	Description
`fopen`	open file
`fscanf`	output
`fprintf`	input
`fread`	binary output
`fwrite`	binary intput
`fclose`	finish off
`feof`	tests the end of file
`ftell`	query position
`fseek`	change position
`fflush`	output: force write all data, input: discard any unprocessed buffered data
`fgets`	reads one line of input and returning a string
`setvbuf`	setting three kinds of method of buffer

- Unbuffered: input/output is passed on as soon as possible.
- Fully buffered: input/output is accumulated into a block then passed.
- line buffered: the block size is based on the new line character.

Week 4

Linked Lists

> Dynamic nature, must allocate and free memory on demand.

Structure

```
struct node{
    int data;
    struct node *next;
};
```

createNewNode

```c
struct node* createNewNode(int data){
    struct node* newNode = malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

- Should apply memory dynamically here.

addToEnd

```c
struct node* addToEnd(struct node* head, int newData){
    struct node* newNode = createNewNode(newData);
    if(head == NULL)
        return newNode;
    struct node* cursor = head;
    while(cursor->next!=NULL){
        cursor = cursor->next;
    }
    cursor->next = newNode;
    return head;
}
```

- Return value is unnecessary here.

deleteFront

```c
struct node* deleteFront(struct node* head){
    if(head == NULL){
        return NULL;
    }
    struct node* next = head->next;
    free(head);
    return next;
}
```

readAll

```c
void readAll(struct node* head){
    if(head == NULL){
        return;
    }
    struct node* cursor = head;
    do{
        printf("%d\n", cursor->data);
        cursor = cursor->next;
    } while (cursor != NULL);
    printf("\n");
}
```

Stack of C

```c
void foo(int n, int m){
    int x = 2;
    int y = 5;
    printf("addr of n is: %p\n", &n);
    printf("addr of m is: %p\n", &m);
    printf("addr of x is: %p\n", &x);
    printf("addr of y is: %p\n", &y);
}
```

For function above, c compiler will create a stack firstly, then **copy** arguments and local variables into this stack, which means the stack doesn't obtain the arguments it-selves, that's why we can't change the value of arguments directly.

What needs attention is, the address of static local variables would not be continuously, it is stored in the memory and always exist after compilation.

Week 5

Function pointer

> An address that refers to an area of memory with executable code, typically the first instruction of the function call.

```c
type (*f)(param declaration...)
```

```c
void hello(int x){
    printf("%d\n", x);
    printf("Hello, World!\n");
}

void goodbye(int x){
    printf("%d\n", x);
    printf("Goodbye, World!\n");
}

void saySomething(void (*sayHello)(int), void (*sayGoodbye)(int), int sayWhat){
    if(sayWhat){
        sayHello(sayWhat);
    } else {
        sayGoodbye(sayWhat);
    }
}
```

Signals

> A process can communicate with another using a signal, can also be generated by the operating system.

Library: `signal.h`

Send Message

```
int kill (pid_t pid, int sig);
```

This function send signal `sig` to process whose id is `pid`.

Receive Message

```
sighandler_t signal(int signum, sighandler_t handler);
```

This function waiting for signal `signum`, once it receive that signal, function `handler` will be called.

- Some signals can be caught and handled by a user supplies function while some signals cannot.

Keyword volatile

A contrary keyword to `const`, meaning it can be changed by some unknown reason like OS, others processes, so every time program need to obtain this variable it will be read from the **memory** to ensure its stability.

Errno

> An error reporting mechanism using a global variable, typically is a integer value, a companion function `strerror` and `perror` will print a textual description.

- `errno` is set by the last function call that will set `errno`, there is only one `errno` value.

Library: `errno.h`

```
#include <errno.h>
#include <stdio.h>
int main() {
    FILE *fp = fopen("doesn't exist", "r");
    printf("errno: %d\n", errno);
    return 0;
}
```

Low Level File I/O

File Descriptors

> A file descriptor is an integer indicating an open file.

- 0 is standard input.
- 1 is standard output.
- 2 is standard error output.
- File that opened with be set a number after that.

Form

```
ssize_t read(int fd, void *buf, size_t n);
//read n character from fd to buf
ssize_t write(int fd, const void *buf, size_t n);
//write n character in buf to fd
```

Standard Input/Output

```
int n = read(0, c, 10);
write(1, c, n);
```

Read Write File

```
void readWriteFile(){
    int fd = open("test.txt", O_RDWR);
    write(fd, "this is a test A", 16);
    lseek(fd, 0, SEEK_SET);
    char buffer[14];
    read(fd, buffer, 14);
    for(int i = 0; i < 14; i++){
        printf("%c", buffer[i]);
    }
}
```

Error Checking

```
int main()
{
    signal(SIGINT, impatient);
    char buffer[100];
    ssize_t result = read(0, buffer, 10);
    int error_val = errno;
    if(0 != error_val){
        printf("read() was interrupted by signal\n");
    }
    printf("%d\n", error_val);
    signalTest();
    return 0;
}
```

`read()` can be interrupt by signal, on error, -1 is returned and `errno` is set appropriately.

Catching Signals

`sigaction()` can be used to catch new signal, stabler than `signal`. It works with `struct sigaction`.

Structure `sigaction`

```
/* Structure describing the action to be taken when a signal arrives.  */
struct sigaction
  {
    /* Signal handler.  */
#if defined __USE_POSIX199309 || defined __USE_XOPEN_EXTENDED
    union
      {
    /* Used if SA_SIGINFO is not set.  */
    __sighandler_t sa_handler;
    /* Used if SA_SIGINFO is set.  */
    void (*sa_sigaction) (int, siginfo_t *, void *);
      }
    __sigaction_handler;
# define sa_handler __sigaction_handler.sa_handler
# define sa_sigaction   __sigaction_handler.sa_sigaction
#else
    __sighandler_t sa_handler;
#endif
    /* Additional set of signals to be blocked.  */
    __sigset_t sa_mask;
    /* Special flags.  */
    int sa_flags;
    /* Restore handler.  */
    void (*sa_restorer) (void);
  };
```

- `sa_handler`: Function which will be called after signal is received. Also can be:
  - `SIG_DFL` for the default action.
  - `SIG_IGN` to ignore this signal.
- `sa_mask`: A set of signal that would not be received by the process **when the handler is running**, it will be received after this handler is finished.
- `sa_flag`: The behavior after receiving signal.
  - `SA_NOCLDSTOP`: Child process wouldn't release `SIGCHLD` signal when it is stopped.
  - `SA_RESETHAND`: Restore the behavior of the signal after call (just use the new behavior once).
  - `SA_NODEFER`: Wouldn't block the signal while executing the signal handler function (without this flag, core will block the signal until the function is finished).
  - `SA_RESTART`: If signal terminate a function, restart it and not giving `EINTR` error.

Function `sigaction`

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

- `sig`: Signal waiting for.
- `act`: Function executed after receive the signal.
- `oact`: Store the previous action of this signal.

Others

Function `ioctl`

> Manipulates the underlying device parameters of special files.

Function `umask`

> User file-creation mode mask, affect how the file permissions are changed.

Function `fcntl`

> Manipulate file descriptors.

Week 6

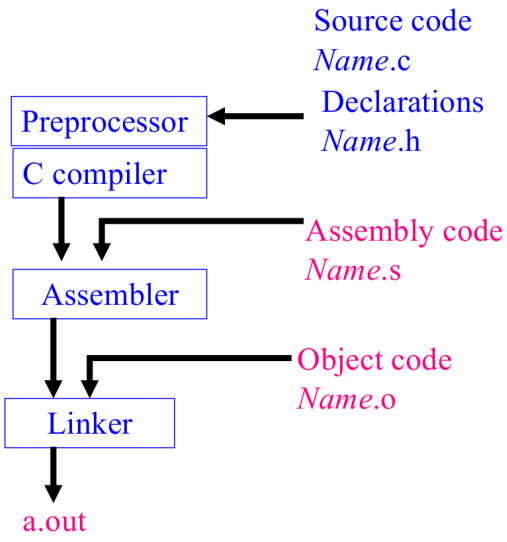
The C Preprocessor

Stage of The Compilation

- Preprocessing.
- Syntax checking.
- Translation to assembly.
- Compose object files.
- Link symbols between object files.
- Produce binary executable.

Object Code Files

End in `.o`, produced from `.c` file, object code files are in machine language **but not linked with other parts of the program**.



To produce the `.o` file: `gcc -c file.c`.

To combined several `.c` or `.o` files: `gcc fileA.c fileB.o -o program`.

Preprocessor commands

> The C preprocessor interprets those lines starting with `#`

`#include`

> `#include` is used to include text from another file, the names of included files end in `.h`

- `externs`, variables can be access from the outside of the header files.

If the file name is enclosed in `<>`, the file is searched for in `/usr/include`.

The preprocessor can be instructed to look in other directories using the `gcc -Ipath`.

`#define`

> `#define` (macro) can be used to replace a number, a char, a string or an expression with a nickname.

```
#define NUMBER 123*123
#define CHAR 'a'
#define STRING "string"
#define min(a, b) ((a)<(b)?(a):(b))
```

Function of macro also can be optimize from another file.

```
//headerA.h
#define ISay(string) saysomething("Me", string)
//headerB.h
extern void saysomething(char* who, char* what){
    printf("%s say %s\n", who, what);
}
```

Controlling the preprocessor from the gcc command

`gcc -Didentifier`

It can be overridden by `#define` or `#undef` within the program.

Conditional Inclusion

> the preprocessor allows to select text to be included or not. It's useful for debugging.

macro	description
`#ifdef`	If a symbol is defined
`#ifndef`	If a symbol is not defined
`#if`	Same as `if()`
`#else`	Same as `else()`
`#elif`	Same as `else if()`

Pre-defined Symbols

- `__LINE__` contains the current line number at any point.
- `__FILE__` contains the name of the current program file.

Make

- The `make` program will **automate the process of recompiling**.
- Stored in `Makefile` or `makefile`.

How to write the file

`makefile` contain the dependence files and command used as usual, **may contain several rules**.

In general, make rules have:

- A `target`
- One or more `dependencies`
- An action: a shell command that creates the target

```
target: dependenciesA
    action
```

Example:

```
target.o: dependenciesA.c dependenciesB.h
    gcc -c dependenciesA.c
```

How It Works

1. After command `make`, it will read the `makefile` from the top to the bottom, if there are several target, it will choose the first one as its final target file.
2. If the final target file doesn't exist or it's dependence files' latest modified time is later than final target file's, it will re-execute the action.
3. If the final target file's dependence files doesn't exist, it will read the rest part of the file to find out how to generate the dependence files.

Default Rules

By default:

```
filename.o : filename.c
    gcc -c filename.c
```

`make` knows the default rules between `filename.o` and `filename.c`, so don't have to specify a rule for `filename.o` **except** some new headers are linked into the object code files:

```
filename.o : header.h
```

Combining rules

When some targets have common dependencies and actions, `makefile` can be written like:

```
ocf1.o ocf2.o: header.h
```

Rules without dependencies

Rules that don't have dependencies, always active

```
clean:
    rm *.o
```

Can be called by `make clean`.

Make variables

Form:

```
variable_name = any character sequence
```

The value of the variable is substituted with:

```
$(variable_name)
```

Predefined variables

Can be changed like a variables

- `CC` the default C compiler.
- `CFLAGES` flags passed to the compiler.

Libraries

- A large system might consist of several programs that share a number of functions.
- When using function from others libraries, need to link the libraries.

How To Link Standard Libraries

```
gcc -lm myprog.o -o myprog
```

The C compiler will search for the library in standard directories: `/lib`, `/usr/lib`.

- `-lm`: link the `libm.so` for `math.h`
- `-lc`: link the `libc.so`, contain the `printf` and so on, compiler will link this libraries by default.
- `-lz`: link the `libz.so`
- `-lpthread`: link the `libpthread.so`

How To Create Own Libraries

```
ar c mylib.a ocf1.o ocf2.o
gcc myprog.o mylib.a -o myprog
```

Week 7

Processes and Memory

- OS manages memory for processes, devices and communication.
- Memory contains both data and instructions.

- System memory is divided into:
  - Kernel
    - only processes with certain privileges can r/w/x (read/write/execute).
    - OS functions and data live here.
    - protect the hardware by only accessing through this layer.
  - User
    - all user created processes privilege depends on who created.
    - These programs are treated as rogue/untrusted that can run and die.
    - processes in this space are independent.

  User space processes use `system calls` to access Kernel space.
- Mode bit:
  - Kernel mode (the bit is set)

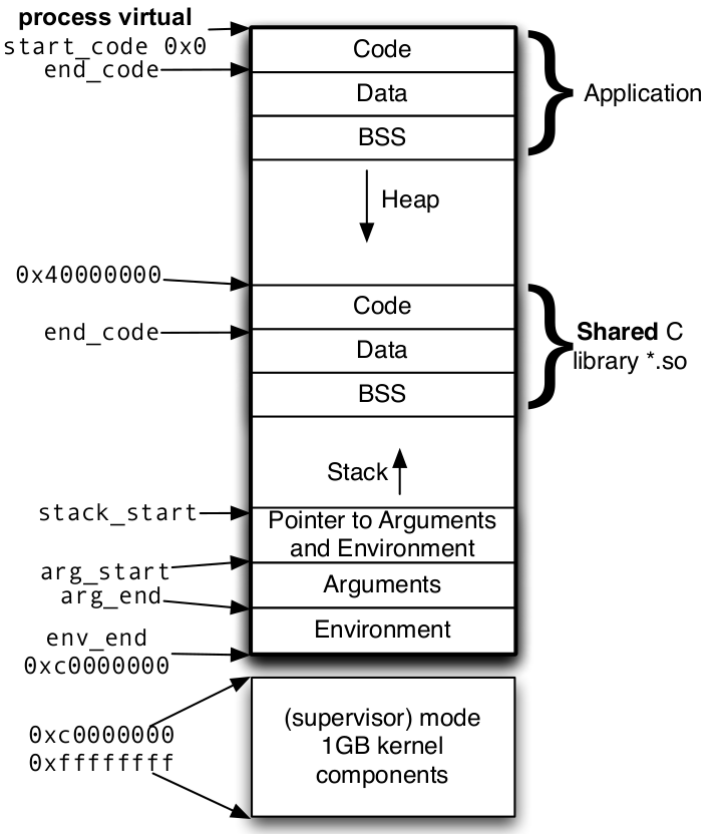
    Allowed to execute **privileged instructions** and reference code or data in the **kernel area**.
  - User mode (the bit is not set)

    A process running application code is initially in this mode.

  The only way to change user mode to kernel mode is via an **exception**, the handler runs in kernel mode.

- User creating a new process:
  - OS creates a **new image of memory** that will be used by the process by cloning an existing
    - This new image of memory is the **copy of parent's user-level virtual address space**, including code and data segments, heap, shared libraries, user stack, and open file descriptors, different between child and parent is **different PIDs**.
  - This has permissions associated with r/w/x of user/group.
- The memory inside the process is assigned a virtual address range.
  - Pieces of virtual memory get mapped to physical memory **when they are needed during execution**.

Virtual Memory



Why Need it

Provides each process with the illusion that it has exclusive use of the main memory.

How It Works

Virtual Memory of a process is mapped to physical memory.

How It Is Stored

1. Multiple processes share the same finite memory resources, so the physical primary memory is easily exhausted.
2. Whatever cannot fit is stored in **secondary memory**, so a process can potentially have more memory than system supports.

Structure (From top to bottom)

- Memory of a process is divided into several parts.
- Size of a new process's virtual memory address space varies with OS.

1. Program code and data

   Code begins at the same fixed address for all processes.

   Data area is for global C variables.

   Initialized directly from the contents of an executable object file.

2. Run-time heap

   Contains data that generate dynamically at run time, created by `malloc`.

3. Shared libraries

   Holds the code and data for libraries.

4. User stack

   Created at run time, to store the data and executing function's information, the top of the stack is `%esp`, the bottom of the stack is `%ebp`.

5. Kernel virtual memory

   Application not allowed to read, write or directly call functions from this area.

Processes

- The standard C library includes functions that invoke Unix `system calls`.
- A set of these functions allows to **initiate and manage** the running of other programs or processes.
- The shell uses these functions to start the programs that correspond to the commands.

The `main` function

> When a program is started the main function is called.

```
int main(int argc, char *argv[], char *envp[])
```

- `argc` is the number of arguments passed.
- `argv` is the array of pointers to strings containing the arguments.
- `envp` is an array of pointers to strings containing the environment variables.

`execl` Functions

- Can be used to switch current program to another program, PID wouldn't be changed (like `JUMP`, not starting a new process).
- Current program will be terminated.
- If `exec` failed, it will return and execute current program continually.
  - If it return `-1`, the program was not found.
  - If it return `0` or greater, the function itself has failed.

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

The `fork` Function

- Both the parent and the child programs run in parallel.
- The return value from the fork function is different for the parent and the child.
  - **0** in the child process.
  - **The PID of the child** in the parent process.
  - **-1** in the parent process if the fork failed.

How It Works

It create a new process which starts executing the following program with the original process together, because the new process's virtual memory is the same as it's parent's.

The Return Value

The return value for child process and parent process is different, so it can be used like:

```
int pid;
if((pid = fork()) == 0){
        printf("inside the child and pid = %d\n", getpid());
        fflush(stdout);
        sleep(1);
        exit(0);
}
while(1){
    printf("inside the parent and pid = %d\n", getpid());
    printf("**************************************\n");
    fflush(stdout);
    sleep(1);
}
```

- For the parent process, it receive the PID of the child as the return value, so it wouldn't go into the if statement.
- For the child process, it receive 0 as the return value, so it goes into the if statement.

The `wait` Function

Library: `types.h`

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

- `pid`: the process waiting for, `-1` represent all of the child process.
- `status`: a integer variable that will store the reason why target process is ended, there are some function helps interpreting this variable.
- `options`: can be used to modify the behavior.
    - `WNOHANG`: Return immediately, if the target processes are not end, return 0, otherwise return the PID.
    - `WUNTRACED`:Return if the target processes either terminated or stopped, return the PID.
    - `WCONTINUED`: Return if the target processes are terminated or a target process has been resumed by the receipt of a `SIGCONT` signal.

Sleep and Stop

> `pause()` will turn process into **sleep state** while `SIGSTOP` signal will turn process into **stop state**.

To resume the process:

- `pause()`:

- Any catchable signal, need to catch by a handle function.

```
if((pid = fork()) == 0){//child process
    signal(SIGALRM, handler);
    pause();
} else {//parent process
    kill(pid, SIGALRM);
}
```

- `SIGSTOP` signal:
  - `SIGCONT` signal, no need handle function to work with `SIGSTOP` signal.

```
if((pid = fork()) == 0){//child process
    kill(pid, SIGSTOP);
    pause();
} else {//parent process
    kill(pid, SIGCONT);
}
```

- Both:
  - `SIGKILL` also known as `kill -9`, the signal that uncatchable and will terminate the process.

Week 8

Interprocess Communication

Pipe

> Pipe is a queue, it has a read end (read only) and a write end (write only), indicated by two file descriptors.

```
int pipe(int fd[2]);
```

- If a process attempts to read from an **empty pipe**, then `read()` will block until data is available.
- If a process attempts to write to a full pipe, then `write` blocks until sufficient.
- If all file descriptors referring to the write end have been closed, then `read()` will see end-of-file, return 0.
- If all file descriptors referring to the read end have been closed, then `write()` will cause a `SIGPIPE` signal, if process is ignoring this signal, then it fails with the error `EPIPE`.

Pipe Capacity

A pipe has a limited capacity, If the pipe is full, then a `write` will block or fail, depending on the `O_NONBLOCK` flag.

File

Function `select`

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
struct timeval *timeout);
```

- `fd_set` is a set which contains file descriptor, can be modify by follow function:
  - `FD_ZERO(fd_set *)`, empty the set.
  - `FD_SET(int, fd_set *)`, add specific file descriptor to the set.
  - `FD_CLR(int, fd_set *)` delete specific file descriptor from the set.
  - `FD_ISSET(int, fd_set *)` check file descriptor can write or read or not.
- `struct timeval` is a structure that use to represent time, it contains two member, one is `tv_sec`, one is `tv_usec`.
  - If `timeval` is `NULL`, it will block until one of the monitored file descriptor is changed.
  - If `timeval` is 0, it will not block and return immediately.
  - If `timeval` is bigger than 0, it will block until the time out or some of the monitored file descriptor is changed.
- `maxfdp` should be bigger than the largest file descriptor in the set, so it should be `largest fdp + 1`.

The return value of `select`

- 0: Time out, no file can be read or written.
- Positive: Success, some file can be read or written.
- Negative: Function failed.

Shared Memory

- Parent process's shared memory will inherited to children process.

`ftok`

```
key_t ftok(const char* pathname, int proj_id)
```

An **unique id** should be indicated when system establish a IPC communication.

- `pathname` should be the path of an existed and accessible file.
- `proj_id` can be arbitrary any integer between 0 and 255.

`shmget`

> This function return a valid shared memory identifier if the function successes, otherwise it return -1.

```
int shmget(key_t key, size_t size, int shmflg)
```

- `key` is generated from `ftok`, an unique IPC id in the system.
- `size` is the storage of shared memory.

- `shmflg` indicated the behaviors and authorities of function `shmget`.
  - `IPC_CREAT`: If segment doesn't exist, create a new one.
  - `IPC_EXCL`: If segment exist, function fails.

`shmat`

> This function return the address in current process.

```
void *shmat(int shmid, const void *addr, int flag)
```

- `shmid`: is generated from `shmget`.
- `addr`: is the address in current process that expected to attach to shared memory, if it's `NULL`, system will select a suitable position to store.
- `flag`: indicated the authority of read, write and etc.

`shmdt`

> Detach shared memory from current position, make it unaccessible.

```
int shmdt(const void *shmaddr)
```

`shmctl`

> This function can control the shared memory.

```
int shmctl(int shm_id, int command, struct shmid_ds *buf)
```

`struct shmid_ds`

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

Use to control the shared memory.

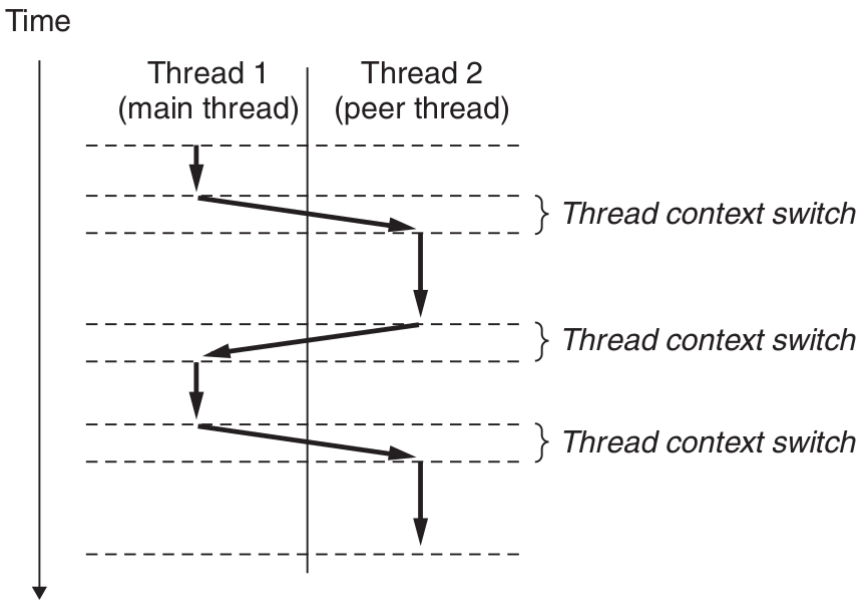
- `command`: instruction that process will do:
  - `IPC_STAT`: get information from `shm_id` to `buf`.
  - `IPC_SET`: set the value of `shm_id` by `buf`.
  - `IPC_RMID`: destroyed the shared memory.

Week 9

Parallelism

- **Task-parallelism**: executing different tasks at the same time.
- **Data-parallelism**: performing the same task to different data-items at the same time.
- **Dependencies**: an execution order between two tasks Ta and Tb, Ta must complete before Tb can execute, notation: $Ta \rightarrow Tb$.

Two tasks can execute in parallel if there is not dependencies from both side.



- A thread context is small so it can switch faster.
- Threads are not organized in a rigid parent-child hierarchy.

Threads Memory Model

- A pool of concurrent threads runs in the context of a process.
- Each thread has its own **separate thread context**:
  - A thread ID
  - **Stack and stack pointer**
  - Program counter
  - Condition codes
  - General-purpose register values
- Share the rest of the process context with the other threads:
  - Read-only text (code), read/write data
  - **The heap**
  - Any shared library code and data areas
  - Same set of open files
  - Shared virtual memory

POSIX threads

> API, Application Programming Interface, a set of functions procedures or classes.

- `pthread.h` should be compile and link with `-lpthread` because it isn't a default library in Linux.

`pthread_create`

> It will create a new thread.

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

- `pthread_t *thread` is the id of the new thread (tid).
- `const pthread_attr_t *attr` is the attributes of the new thread.
- `void *(*start_routine) (void *)` is the function will be executed once the thread is created.
- `void *arg` are arguments that will be passed to `start_routine()` at run-time.
    - Arguments should be passed as a void pointers. It can be an arrays, structure or anything else.

Joinable or detached

At any point in time, a thread is joinable or detached.

Joinable thread:

- Can be reaped and killed by other threads.
- Its memory resources are not freed until it is reaped by another thread.

Detached thread:

- Can **not** be reaped and killed by other threads.
- Its memory resources freed automatically.

`pthread_detach`

> If a thread is detached it could release its memory resources automatically.

```
int pthread_detach(pthread_t tid);
```

same as:

```
pthread_t tid;
pthread_attr_t attr;
pthread_attr_init(&attr);//necessary
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, THREAD_FUNCTION, arg);
```

`pthread_join`

> It will block current thread until the thread waiting is done.

```
int pthread_join(pthread_t thread, void **retval);
```

- `pthread_t thread` is the id of the thread waiting for.
- `void **retval` is the return value of the thread waiting for.

Thread termination

A thread can be terminate by:

1. Returning from `start_routine()`.
2. Calling from `pthread_exit()`.
    - Similar to `pthread_join()`, it will exit after all other peer threads to terminate, and store the return value into the address given as the argument.
3. Canceling by any thread with `pthread_cancel()`.
    - Will cancel thread immediately.
4. Calling `exit()` or the main thread performs a return from `main()`.
    - `exit()` terminates the whole program including all threads immediately.

Terminating a thread **does not** release:

- dynamic memory allocated by the thread(`malloc`).
- close files that have been opened by the thread.

Week 10

Race Conditions

> A situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution.

Critical section:

> Two or more code-parts that access and manipulate shared data.

Mutexes

There are two kinds of mutexes in Linux, one is kernel mutex, one is user space mutex, the mutex in this article is about user space mutex, which is based on futex.

The `Pthread` library provides mutexes to synchronize access to shared global variables.

```
typedef union
{
  struct __pthread_mutex_s __data;
  char __size[__SIZEOF_PTHREAD_MUTEX_T];
  long int __align;
} pthread_mutex_t;
```

A mutex has two possible states:

- Unlocked
- Locked

The `Pthread` library does not guarantee fairness, so starvation (some thread may be block forever) may appear.

Initialization

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;//or
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);
```

Dynamic creation

```
pthread_mutex_t *mylock;
mylock = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(mylock, NULL);
...
pthread_mutex_destroy(mylock);
free (mylock);
```

Three Types of Mutexes

1. `PTHREAD_MUTEX_DEFAULT`, `PTHREAD_MUTEX_NORMAL` and `PTHREAD_MUTEX_TIMED_NP`
2. `PTHREAD_MUTEX_RECURSIVE`
3. `PTHREAD_MUTEX_ERRORCHECK`

Set type with function:

```
int pthread_mutexattr_settype (pthread_mutexattr_t *__attr, int __kind)
```

Lock

```
int pthread_mutex_lock(pthread_mutex_t *mutex));
```

- If the mutex is currently unlocked, it becomes locked and owned by the calling thread.
- If the mutex is already locked by another thread, it suspends the calling thread (block and add into a waiting queue).
- If the mutex is already locked by the calling thread:
    - Default type will cause deadlock.

- Recursive type succeeds recording the number of times the calling thread has locked the mutex. An equal number of `pthread_mutex_unlock` operations must be performed before the mutex returns to the unlocked state.
- Error check type returns immediately with the error code `EDEADLK`.

Unlock

```
int pthread_mutex_unlock (pthread_mutex_t *__mutex)
```

- If the mutex is of the default kind, it always returns it to the unlocked state.
- If it is of the recursive kind, it decrements the locking count of the mutex and only when this count reaches zero is the mutex actually unlocked.
- If it is of the error checking type, it will return an error code if:
  - Mutex is not locked on entrance.
  - Mutex is not locked by current thread.

Try lock

```
int pthread_mutex_trylock (pthread_mutex_t *__mutex)
```

- If it is not locked, same behavior as `pthread_mutex_lock`.
- If it is locked, return immediately with the error code `EBUSY`.

Destroy

```
int pthread_mutex_destroy (pthread_mutex_t *__mutex)
```

- It destroy a mutex object and freeing the resources.
- the mutex must be unlocked on entrance.

Create Monitor by Mutex

```
void producer(){
    pthread_mutex_lock(&mylock);
    if(count != MAX)
        produce();
    pthread_mutex_unlock(&mylock);
}

void consumer(){
    pthread_mutex_lock(&mylock);
    if(count != 0)
        consume();
    pthread_mutex_unlock(&mylock);
}
```

- At any time, at most one thread can execute any monitor function.

What data needs protection?

- Shared data

Semaphores

> Semaphores are non-negative integer synchronization variables.

- Two basic operations on a semaphore s:
  - ```
    P(s) : {
      while(s == 0)
        wait();
      s--;
    }
    ```
  - ```
    V(s) : {
      s++;
    }
    ```
- `P(s)` and `V(s)` are atomic operation.
- Semaphore invariant: $s \geq 0$.

`sem_init`

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- If `pshared` is 0, then semaphore is shared between the threads of a process, then semaphore should be located at some address that is visible to all threads (global or on the heap).
- If `pshared` is nonzero, then semaphore is shared between processes, it should be located in a region of shared memory.
- `Value` specifies the initial value for the semaphore.

`sem_wait`

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

- `sem_wait` locks the semaphore if the semaphore's value is greater than zero, otherwise it blocks until it is not zero or a signal handler interrupts the call.
- `sem_trywait` is the same as `sem_wait` except that if the decrement cannot be performed it will returns `EAGAIN` instead of blocking.
- `sem_timedwait` is the same as `sem_wait` except that if the time out it will return `ETIMEDOUT`.

`sem_post`

```
int sem_post(sem_t *sem);
```

- `sem_post` increments the semaphore, no upper limitation.

`sem_destroy`

```
int sem_destroy(sem_t *sem);
```

- Destroying a semaphore that other processes or threads are currently blocked on produces undefined behavior.

Semaphores vs Mutexes vs Monitor

Semaphores vs Mutexes

Mutexes are less flexible than semaphores:

- Semaphores doesn't have a owner.
- Mutexes is initially unlocked, semaphores can be initialized to 0.
- Mutexes only have two states.
- Semaphores is a counter.

Monitor vs (Semaphores and Mutexes)

Mutexes and Semaphores are voluntarily:

- Programmer might forget to use them, then introduce a race condition.
- Deadlock.

Monitor = shared data + mutual exclusion/synchronization mechanism

- Monitors are safer.
- Monitors are flexible.
- Monitors are slightly more inefficient.

Deadlocks

> Condition that involving one or more threads, such that each thread is waiting for one of the resources but all of them are already held.

- Advise: Multiple mutexes must always be obtained in the same order.

Lock including semaphores, mutexes and monitor.

Self-Deadlock

> A thread attempts to acquire a mutex that it already holds.

```
pthread_mutex_lock(&mylock);
pthread_mutex_lock(&mylock);
```

ABBA-Deadlock

> A thread acquire lock held by B, B thread acquire lock held by A.

Necessary conditions for a deadlock

1. Mutual exclusion: a resource can be assigned to at most one thread.
2. Hold and wait: threads both hold resources and request other resources.
3. No preemption: a resource can only be released by the thread that holds it.
4. Circular wait.

Deadlock Prevention

- Preventing **circular waits**.
    - Locking hierarchy:
        - Impose an ordering on mutexes.
        - Require that all threads acquire mutexes in the same order.
            - Example 1: Both thread A, B acquire mutex a, b, c, if thread B find A is locked, wait until it is unlocked.
            - Example 2: Thread A acquire mutex a, b, c, and thread B only acquire b, c.
            - Example 3: Both thread A, B acquire mutex a, b, c, if thread B find A is locked, unlock mutex b and c.
- Preventing **starvation**: `Pthread` is fairless.
- Do not double-acquire the same lock.
- Complexity in a locking scheme invites deadlocks, design for simplicity.
- Remember to free mutex, do not enter an endless loop and never leave.

Lock Contention and Scalability

Lock Contention

- A highly contended lock has many threads waiting to acquire it.
- A highly contended limiting parallelism.

Lock Scalability

- Measurement how well a system can be expanded.
- A highly contended lock limits scalability.
- Not all programs are "embarrassingly" parallel.
- Programs have sequential parts and parallel parts.

Locking Granularity

- Locking granularity describes **the amount of data** that a lock protects.
- A **coarse-grained lock** protects a large amount of data, likely to be highly contended.
- A **find-grained lock** protects only a small amount of data.

Livelocks

> Two or more threads are busy synchronizing and do not make progress in what they actually want to compute.

```
start:
P1 lock A
P2 lock B
P1 lock B fail    context switch
P2 lock A fail    context switch
P1 release A
P2 release B
goto startF
```

Starvation

> One thread is never allowed into the critical section because lock is fairless.

Amdahl's law

> The performance improvement to be gained from some faster mode of execution is limited by the fraction of the time that the faster mode can be used

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1-p) + \frac{p}{n}}$$

- p is fraction of work that can be parallelized in the old system.
- n is the number of threads executing in parallel.
- Linear speed-up: if the number of threads/cores doubled, execution time is expected to drop a half.
- However, Linear speed-up is often not achievable.

Further Thread Synchronization

Barriers

> A synchronization point at which a certain number of threads must arrive before any participating thread can continue.

- Barriers are of type `pthread_barrier_t`.

Initialization

```
int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t
*restrict attr, unsigned count);
```

- `count` specifies the number of threads that must call `pthread_barrier_wait` before any of them successfully return from the call.

Barrier's attributes

Before setting barrier's attributes, initialization function should be called.

```
int pthread_barrierattr_init (pthread_barrierattr_t *__attr)
```

`pthread_barrierattr_setpshared`

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

- `pshared` has two values:
  - `PTHREAD_PROCESS_PRIVATE`: Barrier is only used inside process.
  - `PTHREAD_PROCESS_SHARED`: Barrier can be use between processes, but it should be located at shared memory.

`pthread_barrierattr_getpshared`

```
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
int *restrict pshared);
```

The attribute will be set into `pshared`.

`pthread_barrierattr_destroy`

```
int pthread_barrierattr_destroy (pthread_barrierattr_t *__attr)
```

Wait

```
int pthread_barrier_wait (pthread_barrier_t *__barrier)
```

- When the **required number of threads** have called `pthread_barrier_wait()` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to one unspecified thread (arbitrary) and 0 is returned to each of the remaining threads.

Destroy

```
int pthread_barrier_destroy (pthread_barrier_t *__barrier)
```

- Use this function to destroy the barrier referenced by barrier and release any resources used by the barrier.

Condition Variables

It can be use to block a thread, this thread will keep waiting until the specific condition is signaled.

Initialization

```
int pthread_cond_init (pthread_cond_t *__restrict __cond, const
pthread_condattr_t *__restrict __cond_attr); //or

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Condition Variables' attributes

`pthread_condattr_setpshared` and `pthread_condattr_getpshared`

These function are the same as barrier's, no talking anymore on it.

`pthread_condattr_setclock` and `pthread_condattr_getclock`

```
int pthread_condattr_setclock(pthread_condattr_t* attr,clockid_t clock_id);
```

- `clockid_t` can be :
  - `CLOCK_REATIME`, real time.
  - `CLOCK_MONOTONIC`, time pass after system start.
  - `CLOCK_PROCESS_CPUTIME_ID`.
  - `CLOCK_THREAD_CPUTIME_ID`.

Wait

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t
*restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
mutex);
```

These functions atomically **release mutex** and cause the calling thread to **block on the condition variable**.

Upon return of function (condition has been signaled), mutex is **locked again**.

For `pthread_cond_timedwait`, if timeouts occur, it will introduce `ETIMEOUT` and re-acquire mutex.

Signal

```
int pthread_cond_signal (pthread_cond_t *__cond);
int pthread_cond_broadcast (pthread_cond_t *__cond)
```

- `pthread_cond_signal` wake up one thread, must be called after mutex is locked and must unlock mutex thereafter.
- `pthread_cond_broadcast` wake up all threads, **only one** can get resources and others have to go back to blocking state.
- If no thread waiting for the signal, the signal is lost.

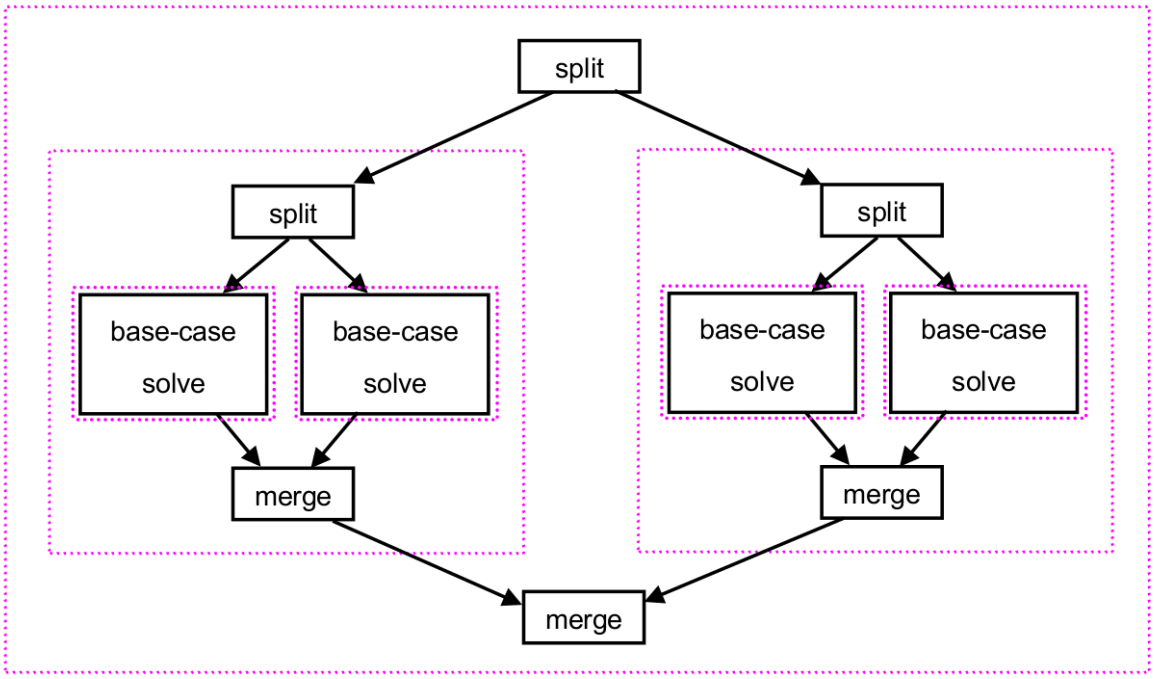
Week 11

Recursion

> 1. Procedure **calls itself** to solve a simpler version of the problem.
> 2. At some stage, the problem is so simple that we can solve it (**base-case**, a fundamental situation) and then recursion terminates.

Divide and Conquer

> **Divide** the problem into several subproblems, **conquer** subproblems and **combine** solutions to create solution to the original problem.



- Each dashed-line box represents a thread.
  - Example: Merge-sort

- Need to mention that: computer don't have so much processor for unlimited number of threads.

Reduction Operation

> A reduction operation reduces a collection of data items to a single data item by repeatedly combining the data items pairwise with a binary operator.

For example: sum, product, maximum.

Week 12

Load Balancing Problem

- Threads that finish early have to wait for the thread with the largest amount of work to complete.
- Static assignment = assignment at program writing time.
- Dynamic assignment = assignment at run-time.

Measuring Performance

Wallclock time: time from start to termination of our program.

- Because of the principal problem (different program run at the same time), wallclock time will be larger than the sum of user time and system time.

```
int gettimeofday (struct timeval *__restrict __tv, void *__restrict __tz)
```

Sources of Performance Loss

False sharing

- Main memory access is extremely slow compared to the processing speed of today's CPUs, so between them we have **cache**.
- Cache composed of **cache line**, if two individual elements in two threads are in the same cache line, one of them is changed, the cache line should be updated and it costs time, this situation is called **false sharing**.
- To reducing false sharing, we can put the individual elements into a structure, and fulfill this structure with the useless element, make it the same large as cache line.

```
struct padded_int{
    int value;
    char pardding[60];
}private_sum[12];
```

Summary of Performance Bottlenecks

- Huge sequential part of a program.
- Highly contended lock.
- High communication overhead, little computation.
- Poor load balancing.
- Scalar code.

Thread pool

A thread pool is a collection of N threads that will perform a general computation task.

A thread pool should contain:

1. A queue for waiting list of function.
2. Some threads used to execute those function.
3. Mutex and condition variable.

A thread pool should implement:

1. Initialization function:
    1. Return a thread pool.
    2. Create space for queue and threads.
2. Threads routine function:
    1. When there's no work item inside queue, it is blocked using condition variable.
    2. When there's works item inside queue, it take it out and update queue, executes the function.
3. Destroy function.

Sources of Performance Loss

Linear speedup with parallel processing frequently not attainable for following reasons:

1. Overhead.

    > Any cost caused in parallel solution but not required by sequential solution is called overhead.

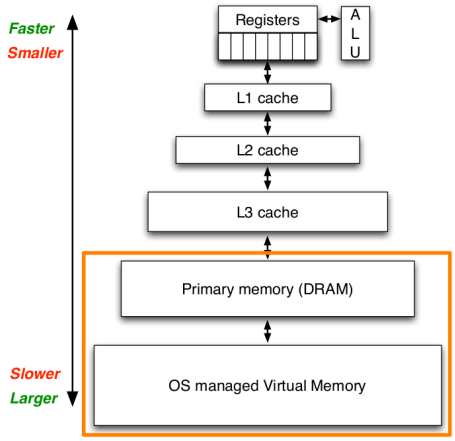
    1. Communication
    2. Synchronization
    3. Computation
    4. Memory
2. Non-parallelizable computations.
3. Idle times.
    1. Lack of work.
    2. Waiting for external event.
    3. Load imbalance.
    4. Memory-bound computations.
4. Contention for Resources.

Performance Trade-offs

1. Communication vs Computation
    1. Redundant computations
    2. Overlapping communication and computation
2. Memory vs Parallelism
3. Overhead vs Parallelism
    1. Parallelize Overhead
    2. Load balance vs Overhead
    3. Granularity Trade-offs

Memory

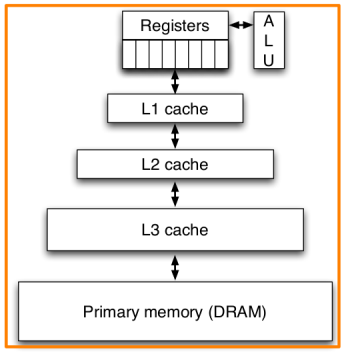
- Virtual memory of a process is mapped to physical memory.
- Physical memory is mix.
- Memory exists at different levels.



- OS does most of this work in software.
- Memory moves to closest part of CPU for computation.
- Stays closer with higher frequency access

machine level

- **Cache protocol** decide when memory moves up and down the hierarchy above primary memory.
- Must be done in hardware, software is too slow to manage memory at this level.



L1 cache

- Instruction cache
  - Amount of instructions to execute is proportional to the size of code.
  - The size of code is proportional to the complexity of problem.
  - Complexity of problem is fixed.
  - Programmers design the data handling, but don't have to think about instructions.
  - Access pattern of instructions much more predictable than data

Data access pattern

- Sequential access
- Random access

Profiling

Profiling = Program Instrumentation + Execution.

Heisenberg effect: measuring can perturb a program's execution time!