CS 332/532 Systems Programming

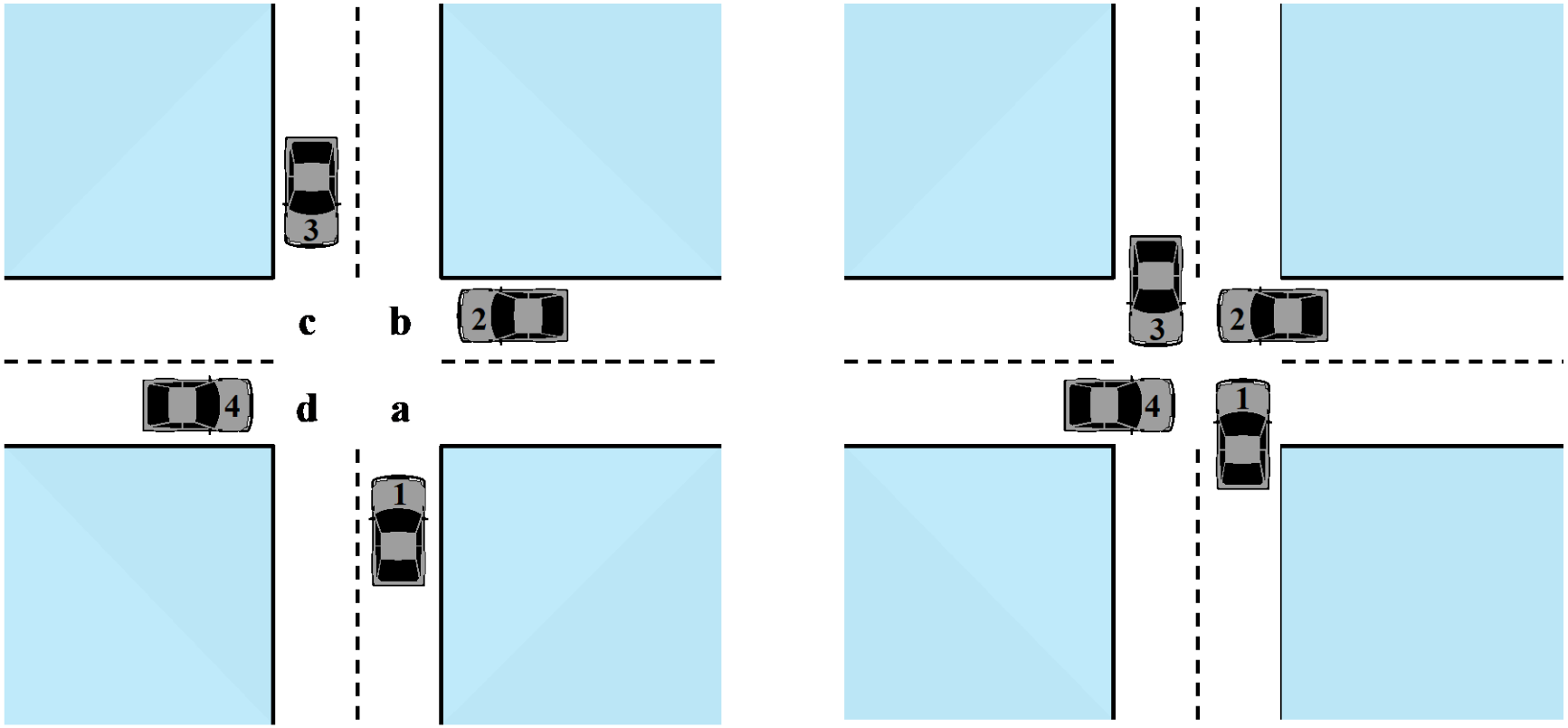
Lecture 39

-REVIEW 3/3-

Professor : Mahmut Unan – UAB CS

Deadlock

- **The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other**
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:

- **Deadlock prevention**
  - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

- **Deadlock avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock

- **Deadlock detection**
  - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
    - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
    - Performing some default action
    - Executing a signal-handler function
    - Ignoring the signal

Signals

- Signals are software interrupts that provide a mechanism to deal with asynchronous events (e.g., a user pressing Control-C to interrupt a program that the shell is currently executing).
- A signal is a notification to a process that an event has occurred that requires the attention of the process (this typically interrupts the normal flow of execution of the interrupted process).
- Signals can also be used as a synchronization technique or even as a simple form of interprocess communication.
- Signals could be generated by hardware interrupts, the program itself, other programs, the OS kernel, or by the user.

- All these signals have a unique symbolic name and starts with SIG. The standard signals (also called POSIX reliable signals) are defined in the header file *<signal.h>*.
- Here are some examples:
- `SIGINT`: Interrupt a process from keyboard (e.g., pressing Control-C). The process is terminated.
- `SIGQUIT`: Interrupt a process to quit from keyboard (e.g., pressing Control-/). The process is terminated and a core file is generated.
- `SIGTSTP`: Interrupt a process to stop from keyboard (e.g., pressing Control-Z). The process is stopped from executing.
- `SIGUSR1` and `SIGUSR2`: These are user-defined signals, for use in application programs.

- ***NOTE***: Please see Section 10.2 and Figure 10.1 in the textbook for a complete list of UNIX System signals. You can also find more details using *man 7 signal* on any CS UNIX system (*man signal* on Mac).
- After a signal is generated, it is delivered to a process to perform some action in response to this signal.
- Since signals are asynchronous events, a process has to decide ahead of time how to respond when the particular signal is delivered.
- There are three different options possible when a signal is delivered to a process:

- Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior then the default action specific to that particular signal will occur.
  - The predefined default signal is specified as `SIG_DFL`.
- Ignore the signal. If a signal is ignored, then the default action is performed.
  - The predefined ignore signal handler is specified as `SIG_IGN`.
- Catch and Handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received.
  - This is usually performed by invoking a signal handler using *`signal()`* or *`sigaction()`* system calls.
- However, the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

signal()

- signal - ANSI C signal handling
-
```
typedef void (*sighandler_t)(int);
```
- `sighandler_t signal(int signum, sighandler_t handler);`

- **signal**() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

Sending Signals Using The Keyboard

- Ctrl-C to send an INT signal (SIGINT) to the running process.
  - This signal causes the process to immediately terminate.
- Ctrl-Z to send a TSTP signal (SIGTSTP) to the running process.
  - This signal causes the process to suspend execution.
- Ctrl-\ to send a ABRT signal (SIGABRT) to the running process.
  - This signal causes the process to immediately terminate.
  - Ctrl-\ doing the same as Ctrl-C but it gives us some better flexibility.

infinite loop

```c
#include<stdio.h>
#include<signal.h>
#include <unistd.h>

void handleSignINT(int sig)
{
    printf("the signal caught =  %d\n", sig);
}

int main()
{
    signal(SIGINT, handleSignINT);
    while (1==1)
    {
        printf("Hello CS332 \n");
        sleep(1);
    }
    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc helloLoop.c -o helloLoop
(base) mahmutunan@MacBook-Pro lecture23 % ./helloLoop
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
^Cthe signal caught =  2
Hello CS332
Hello CS332
^\zsh: quit       ./helloLoop
```

- Now, let's write a program to handle a simple signal that catches either of the two user-defined signals and prints the signal number (see Figure 10.2 in the textbook).
- 1. Create a user-defined function (signal handler) that will be invoked when a signal is received:

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

static void sig_usr(int signo) {
    if (signo == SIGUSR1) {
        printf("received SIGUSR1\n");
    } else if (signo == SIGUSR2) {
        printf("received SIGUSR2\n");
    } else {
        printf("received signal %d\n", signo);
    }
}

```

- 2. Now let’s call above user-defined signal.

```
16  int main(void) {
17      if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
18          printf("can't catch SIGUSR1\n");
19          exit(-1);
20      }
21      if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
22          printf("can't catch SIGUSR2\n");
23          exit(-1);
24      }
25      for ( ; ; )
26          pause();
27
28      return 0;
29  }
```

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigusr.c -o sigusr
(base) mahmutunan@MacBook-Pro lecture23 % ls
sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % ./sigusr &
[1] 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
```

```
(base) mahmutunan@MacBook-Pro lecture23 % kill -SIGUSR1 8122
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -STOP 8122
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
(base) mahmutunan@MacBook-Pro lecture23 % kill -CONT 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -TERM %1
[1]  + terminated  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
(base) mahmutunan@MacBook-Pro lecture23 % _
```

- In the example we used the *kill* command that we used in the previous lecture to generate the signal.
- While we used the kill command in the previous lecture to terminate a process using the TERM signal (the default signal if no signal is specified), the kill command could be used to generate any other signal that is supported by the kernel.

- You can list all signals that can be generated using *kill -l*. For example, on the CS Linux systems you see the following output:

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Exercise 2

- We will now extend the exercise 1 to handle other signal generated from keyboard such as Control-C (SIGINT), Control-Z (SIGTSTP), and Control-\ (SIGQUIT). In this example we will use a single signal handler to handle all these signals using a switch statement (instead of separate signal handlers).

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

static void sig_usr(int signo) {
    switch(signo) {
       case SIGINT:
        printf("received SIGINT signal %d\n", signo);
        break;
       case SIGQUIT:
        printf("received SIGQUIT signal %d\n", signo);
        break;
       case SIGUSR1:
        printf("received SIGUSR1 signal %d\n", signo);
        break;
       case SIGUSR2:
        printf("received SIGUSR2 signal %d\n", signo);
        break;
       case SIGTSTP:
        printf("received SIGTSTP signal %d\n", signo);
        break;
       default:
        printf("received signal %d\n", signo);
    }
}
```

```c
int main(void) {
    if (signal(SIGINT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGQUIT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR1\n");
        exit(-1);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR2\n");
        exit(-1);
    }
    if (signal(SIGTSTP, sig_usr) == SIG_ERR) {
        printf("can't catch SIGTSTP\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro ~ % cd Desktop/lecture23
(base) mahmutunan@MacBook-Pro lecture23 % ls
sighandler      sighandler.c    sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sighandler.c -o sighandler
(base) mahmutunan@MacBook-Pro lecture23 % ./sighandler
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3
^Zreceived SIGTSTP signal 18
^Creceived SIGINT signal 2
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3

_
```

infinite loop

- Notice that we have replaced the default signal handlers to kill this job from the keyboard and the program is in an infinite loop with pause. As a result we cannot terminate this program from the keyboard. We have to login to the specific machine this process is running and kill this process using either *kill* or *top* commands. You can follow the examples from the first part and kill this process.

Default Action Of Signals

- Each signal has a default action, one of the following:
  - **Term:** The process will terminate.
    Core: The process will terminate and produce a core dump file.
    Ign: The process will ignore the signal.
    Stop: The process will stop.
    Cont: The process will continue from being stopped.
- Default action may be changed using handler function. Some signal's default action cannot be changed. **SIGKILL** and **SIGABRT** signal's default action cannot be changed or ignored
- https://linuxhint.com/signal_handlers_c_programming_language/

Basic Signal handler examples

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){

  //Return type of the handler function should be void
  printf("\nInside handler function\n");
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/signal$ ./Example1
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
int main(){
  signal(SIGINT,SIG_IGN); // Register signal handler for ignoring the signal

  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example2.c -o Example2
tump@tump:~/Desktop/c_prog/signal$ ./Example2
1 : Inside main function
2 : Inside main function
^C3 : Inside main function
4 : Inside main function
^C5 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){
  printf("\nInside handler function\n");
  signal(SIGINT,SIG_DFL);   // Re Register signal handler for default action
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example3.c -o Example3
tump@tump:~/Desktop/c_prog/signal$ ./Example3
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
^C
tump@tump:~/Desktop/c_prog/signal$
```

```c
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);
  kill(getppid(),SIGUSR1);
}

int main(){
  pid_t pid;
  if((pid=fork())<0){
    printf("Fork Failed\n");
    exit(1);
  }
  /* Child Process */
  else if(pid==0){
    signal(SIGUSR1,sig_handler_child); // Register signal handler
    printf("Child: waiting for signal\n");
    pause();
  }
  /* Parent Process */
  else{
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
  }
  return 0;
}
```

```c
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);
  kill(getppid(),SIGUSR1);
}

int main(){
  pi
  if
  }
  /*
  el
  }
  /*
  el
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example6.c -o Example6
tump@tump:~/Desktop/c_prog/signal$ ./Example6
Child: waiting for signal
Parent: sending signal to Child
Parent: waiting for response
Child : Received a signal from parent
Parent : Received a response signal from child
tump@tump:~/Desktop/c_prog/signal$
```

Exercise Sigint

- Till now we used the kill command and the keyboard to generate the signals.
- Now we will generate the signal in a program using the C API for *kill* or *raise* system call.
- In this example, we replace the SIGINT signal handler with our own and in the signal handler asks the user if the process should be terminated and then based on the response continue with either terminating the process or let it continue to run.
- We use read function instead of scanf to emphasize that scanf is not a reentrant function

sigint

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

static void sig_int(int signo) {
    ssize_t n;
    char buf[2];

    signal(signo, SIG_IGN); /* ignore signal first */
    printf("Do you really want to do this: [Y/N]? ");
    fflush(stdout);
    n = read(STDIN_FILENO, buf, 2);
    if ( buf[0] == 'Y' ) {
        raise(SIGTERM); // or kill(0, SIGTERM); // or exit (-1);
    } else {
        printf("Ignoring signal, be careful next time!\n");
        fflush(stdout);
    }
    signal(signo, sig_int); /* reinstall the signal handler */
}
```

```c
int main(void) {
    if (signal(SIGINT, sig_int) == SIG_ERR) {
        printf("Unable to catch SIGINT\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}

```

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigint.c -o sigint
(base) mahmutunan@MacBook-Pro lecture23 % ./sigint
^CDo you really want to do this: [Y/N]? N
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]?
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]? Y
zsh: terminated  ./sigint
(base) mahmutunan@MacBook-Pro lecture23 %
```

recall -forkexecvp.c

```c
int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execvp");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}
```

- Let us now consider how signals impact child processes created using fork/exec. We will be running a c program that consist of fork and execvp system calls (hw1.c)
- This code illustrates the use of dynamic memory allocation to create contiguous 2D-matrices and use traditional array indexing. It also illustrate the use of gettime to measure wall clock time.

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^C
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.3  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19320  0.0  0.0 161588  1868 pts/0    R+   10:06   0:00 ps -u
$
```

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^Z
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.1  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri     19352 29.0  0.6  27800 23844 pts/0    T    10:08   0:01 /home/UAB/puri/
puri     19353  0.0  0.0 161588  1864 pts/0    R+   10:08   0:00 ps -u
$ kill -CONT 19352
$
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.0  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri     19352 17.9  0.6  27800 23844 pts/0    R    10:08   0:04 /home/UAB/puri/
puri     19355  0.0  0.0 161588  1872 pts/0    R+   10:08   0:00 ps -u
$ Time taken for size 1000 = 32.274239 seconds
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ kill -CONT 19351
$ Child process exited with status = 0
[19351]: Exiting program .....
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ jobs
$
```

Linux I/O Streams

- Before we discuss I/O redirection, let us review how I/O is handled in Linux systems.
- Each process in a Linux environment has three different file descriptors available when a process is created: standard input (*stdin – 0*), standard output (*stdout – 1*), and standard error (*stderr – 2*).
- These three file descriptors are created when a process is created.
- We use the *stdin* file descriptor to read input from a keyboard or from another a file or from another program.
- Similarly, we use the *stdout* and *stderr* file descriptors to write output and error messages, respectively, to the terminal.

- Input and output in the Linux environment is distributed across three streams. These streams are:
- **standard input** (**stdin**)
- **standard output** (**stdout**)
- **standard error** (**stderr**)

- The streams are also numbered:
- **stdin** (**0**)
- **stdout** (**1**)
- **stderr** (**2**)

During standard interactions between the user and the terminal, standard input is transmitted through the user's keyboard. Standard output and standard error are displayed on the user's terminal as text. Collectively, the three streams are referred to as the *standard streams*.

https://www.digitalocean.com/community/tutorials/an-introduction-to-linux-i-o-redirection

cat command

```
(base) mahmutunan@MacBook-Pro lecture26 % cat
1
1
2
2
3
3
(base) mahmutunan@MacBook-Pro lecture26 % _
```

Stream Redirection

- **Overwrite**

  > - standard output

  < - standard input

  2> - standard error

- **Append**

  >> - standard output

  << - standard input

  2>> - standard error

```
(base) mahmutunan@MacBook-Pro lecture26 % cat > outputFile.txt
1
2
3
4
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat >> outputFile.txt
a
b
c
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
a
b
c
```

```
(base) mahmutunan@MacBook-Pro lecture26 % ls >> listOffiles.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat listOffiles.txt
error.txt
forkexecvp
forkexecvp.c
forkexecvp2.c
hw2.c
input.txt
ioredirect.c
lab7_solution.c
listOffiles.txt
myprog
myprog.c
output.txt
output2.txt
outputFile.txt
```

stdout stderr

```
(base) mahmutunan@MacBook-Pro lecture26 % echo "some output using stdout"
some output using stdout
(base) mahmutunan@MacBook-Pro lecture26 % echo

(base) mahmutunan@MacBook-Pro lecture26 % _
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat nonexistingfile.txt
cat: nonexistingfile.txt: No such file or directory
(base) mahmutunan@MacBook-Pro lecture26 % _
```

- You have already been using these file descriptors when you wrote the insertion sort program in C.
- You read the number of elements and the elements to be sorted from the keyboard using the *scanf* function.
- The *scanf* function was using *stdin* file descriptor to read your keyboard input. In other words, the following two functions are equivalent:

```
scanf("%d", &N);
fscanf(stdin, "%d", &N);
```

- Similarly when you use the *printf* function to print the output of your program you are using the *stdout* file descriptor.

- The two functions below are equivalent:

```
printf("%d\n", N);
fprintf(stdout,"%d\n", N);
```

- The file descriptors *stdin*, *stdout*, and *stderr* are defined in the header file *stdio.h*. We typically use the *stderr* file descriptor to write error messages.

- If we need to save the output or error message from a program to a file or read data from a file instead of entering it through the keyboard, we can use the I/O redirection supported by the Linux shell (such as bash).
- In fact, we already used this in one of the earlier labs when we used insertion sort to sort large input values.
- The following examples show how to use I/O redirection in the bash shell to read input from a file (instead of entering it from the keyboard) and send the output and error messages to different files (instead of the terminal)

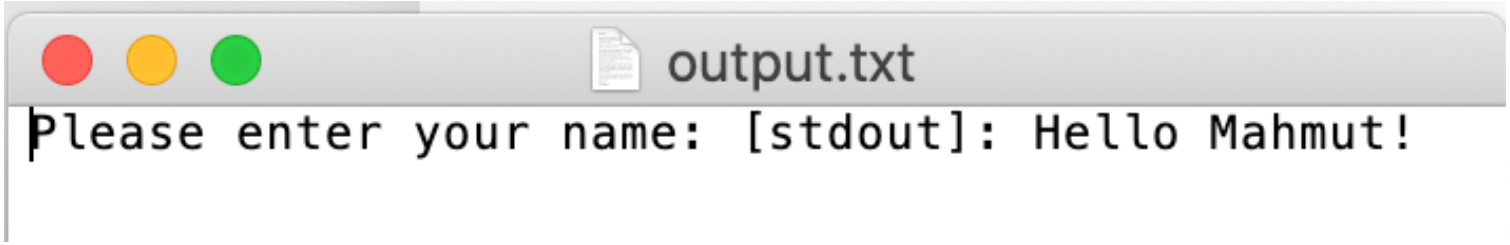
Exercise 1

```c
#include <stdio.h>

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

    printf("Please enter your name: ");
    scanf("%s", name);
    printf("[stdout]: Hello %s!\n", name);
    fprintf(stderr, "[stderr]: Hello %s!\n", name);

    return 0;
}
```

Compile the program myprog.c,
create a file called *input.txt*, type you name in the file *input.txt*

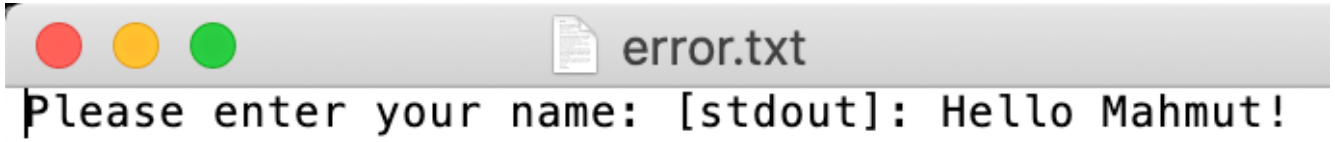
compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % touch input.txt
(base) mahmutunan@MacBook-Pro lecture26 % echo "Mahmut" > input.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat input.txt
Mahmut
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall myprog.c -o myprog
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt > output.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % _
```



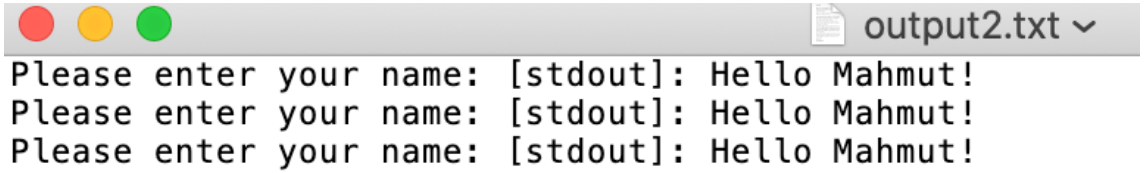
compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt
Please enter your name: [stdout]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt >output.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >output2.txt 2> error.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt & >output2.txt 2> error.txt
[1] 91593
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
[1]  + done       ./myprog < input.txt
^C
(base) mahmutunan@MacBook-Pro lecture26 %
```



compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 %
```



You can replace **>** with **>>** if you like to append to the file instead of overwriting the file

Sharing between parent and child processes

- When we created a new process using fork/exec in the previous labs, we noted that the child process is a copy of the parent process and it inherits several attributes from the parent process such as open file descriptors.
- This duplication of descriptors allowed the child processes to read and write to the standard I/O streams (note that the child process was able to read input from the keyboard and write output to the terminal).
- As a result of this sharing both parent and child processes share the three standard I/O streams: *stdin*, *stdout*, and *stderr*.

Exercise 2

- Let us use the example from the previous lectures to illustrate this by adding the following lines in the parent process:

```
    char name[BUFSIZ];

        printf("Please enter your name: ");
        scanf("%s", name);
        printf("[stdout]: Hello %s!\n", name);
        fprintf(stderr, "[stderr]: Hello
%s!\n", name);
```

Exercise 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
    perror("exec");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
```

Exercise 2

```
23          char name[BUFSIZ];
24
25          printf("[%d]: Please enter your name: ", getpid());
26          scanf("%s", name);
27          printf("[stdout]: Hello %s!\n",  name);
28          fprintf(stderr, "[stderr]: Hello %s!\n",  name);
29
30          wait(&status); /* wait for the child process to terminate */
31          if (WIFEXITED(status)) { /* child process terminated normally */
32              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33          } else { /* child process did not terminate normally */
34              printf("Child process did not terminate normally!\n");
35              /* look at the man page for wait (man 2 wait) to determine
36                 how the child process was terminated */
37          }
38      } else { /* we have an error */
39          perror("fork"); /* use perror to print the system error message */
40          exit(EXIT_FAILURE);
41      }
42
43      return 0;
44  }
45
```

compile & run

- If we compile and execute the program by using *myprog* (used earlier) as the child process, we will notice that the prompt to enter the name is printed twice. If we enter the name, which process is reading the keyboard input?

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[86530]: Please enter your name: Please enter your name: mahmut
[stdout]: Hello mahmut!
[stderr]: Hello mahmut!
```

Exercise 2

```
23          char name[BUFSIZ];
24
25          printf("[%d]: Please enter your name: ", getpid());
26          scanf("%s", name);
27          printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28          fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30          wait(&status); /* wait for the child process to terminate */
31          if (WIFEXITED(status)) { /* child process terminated normally */
32              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33          } else { /* child process did not terminate normally */
34              printf("Child process did not terminate normally!\n");
35              /* look at the man page for wait (man 2 wait) to determine
36                 how the child process was terminated */
37          }
38      } else { /* we have an error */
39          perror("fork"); /* use perror to print the system error message */
40          exit(EXIT_FAILURE);
41      }
42
43      return 0;
44  }
```

compile & run

- We could add the PID in the printf statement to make this explicit. We can update the code above to print the PID and test the program. In any case, this illustrates the result of sharing of the standard I/O streams between the parent and the child processes

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[85953]: Please enter your name: Please enter your name: mahmut
[85953-stdout]: Hello mahmut!
[85953-stderr]: Hello mahmut!
```

- If we like to change the behavior of all child processes to use separate files instead of the standard I/O streams we have to replace the standard I/O file descriptors with new file descriptors.
- We will use the dup2() system call to create a copy of this file descriptors and associate separate files to replace the standard I/O streams.

dup2()

- **Copying file descriptors – dup2() system call**
- The dup2() system call duplicates an existing file descriptor and returns the duplicate file descriptor.
- After the call returns successfully, both file descriptors can be used interchangeably. If there is an error then -1 is returned and the corresponding errno is set (look at the man page for dup2() for more details on the specific error codes returned).
- The prototype for the dup2() system call is shown below:

```
#include <unistd.h>

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

dup() vs dup2()

- These system calls create a copy of the file descriptor *oldfd*.
- **dup**() uses the lowest-numbered unused descriptor for the new descriptor.
- **dup2**() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary, but note the following:
  - If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
  - If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.
- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. T

Exercise

- This example illustrate how to use dup2 to replace the stdin and stdout file descriptors with the files stdin.txt and stdout.txt, respectively.
- We use the file forkexecvp.c from the previous lab and the new lines of code are highlighted.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv) {
    pid_t pid;
    int status;
    int fdin, fdout;

    /* display program usage if arguments are missing */
    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    /* open file to read standard input stream,
       make sure the file stdin.txt exists, even if it is empty */
    if ((fdin = open("stdin.txt", O_RDONLY)) == -1) {
       printf("Error opening file stdin.txt for input\n");
       exit(-1);
    }
```

```
26
27      /* open file to write standard output stream in append mode.
28         create a new file if the file does not exist. */
29      if ((fdout = open("stdout.txt", O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
30         printf("Error opening file stdout.txt for output\n");
31         exit(-1);
32      }
33
34      pid = fork();
35      if (pid == 0) { /* this is child process */
36          /* replace standard input stream with the file stdin.txt */
37          dup2(fdin, 0);
38
39          /* replace standard output stream with the file stdout.txt */
40          dup2(fdout, 1);
41
42          execvp(argv[1], &argv[1]);
43          /* since stdout is written to stdout.txt and not the terminal,
44             we should write to stderr in case exec fails, we use perror
45             that writes the error message to stderr */
46      perror("exec");
47      exit(-1);
```

```
48      } else if (pid > 0) { /* this is the parent process */
49          /* output from the parent process still goes to stdout :-) */
50          printf("Wait for the child process to terminate\n");
51          wait(&status); /* wait for the child process to terminate */
52          if (WIFEXITED(status)) { /* child process terminated normally */
53              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
54              /* parent process still has the file handle to stdout.txt,
55                 now that the child process is done, let us write to
56                 the file stdout.txt using the write system call */
57              write(fdout, "Hey! This is the parent process\n", 32);
58              close(fdout);
59              /* since we opened the file in append mode, the above text
60                 will be added after the output from the child process */
61          } else { /* child process did not terminate normally */
62              printf("Child process did not terminate normally!\n");
63              /* look at the man page for wait (man 2 wait) to determine
64                 how the child process was terminated */
65          }
66      } else { /* we have an error */
67          perror("fork"); /* use perror to print the system error message */
68          exit(EXIT_FAILURE);
69      }
70
71      return 0;
72  }
```

compile & run

- Compile and run this program using myprog as the child process. Note that you have provide input to *myprog* in the file *stdin.txt* and the output of *myprog* will be written to *stdout.txt*. As we did not do anything with the *stderr* stream, the output to *stderr* stream goes to the terminal. You can add the PID in the print statement to confirm which output is sent to the terminal and which output is sent to the files. Here is a terminal session that illustrates this interaction:

recall myprog.c

```c
#include <stdio.h>

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

    printf("Please enter your name: ");
    scanf("%s", name);
    printf("[stdout]: Hello %s!\n", name);
    fprintf(stderr, "[stderr]: Hello %s!\n", name);

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o myprog myprog.c
(base) mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o ioredirect ioredirect.c
(base) mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt
Mahmut
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello Mahmut!
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
```

```
(base) mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt
World
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello World!
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
(base) mahmutunan@MacBook-Pro lecture27 % _
```

```
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect uname -a
Wait for the child process to terminate
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Mon Aug 31 22:12:52 PDT 2020; roo
t:xnu-6153.141.2~1/RELEASE_X86_64 x86_64
Hey! This is the parent process
(base) mahmutunan@MacBook-Pro lecture27 % _
```

Pipes

- We have seen the Linux shell support pipes.
- For example:

```
$ ps -elf | grep ssh
```

- The above example redirects the output of the program ps to another program grep (instead of sending the output to standard output).
- Similarly, the program grep uses the output of ps as the input instead of a file name as the argument. The shell implements this redirection using pipes.
- The system call pipe is used to create a pipe and in most Linux systems pipes provide a unidirectional flow of data between two processes.

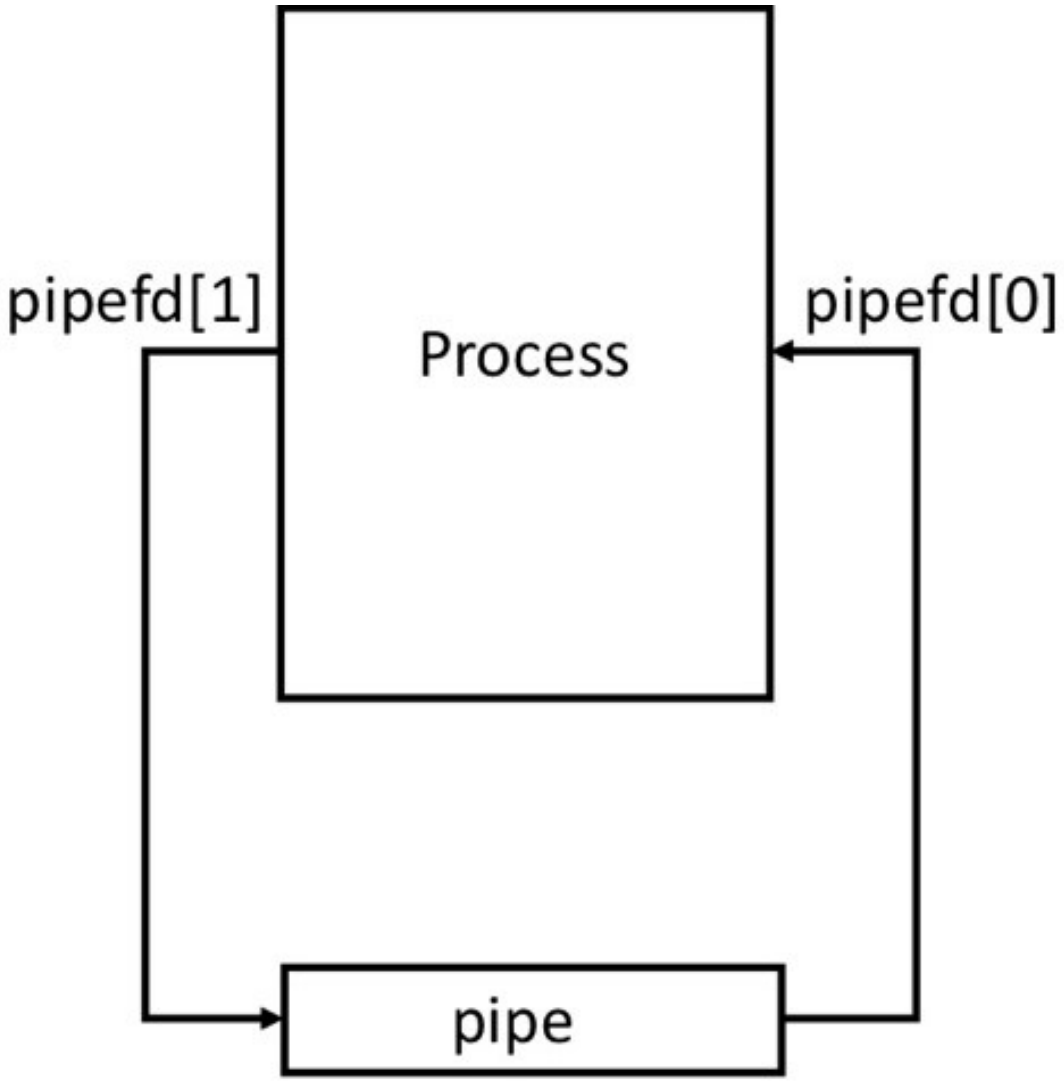
The C API for the pipe function

- The C API for the pipe function is shown below:

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

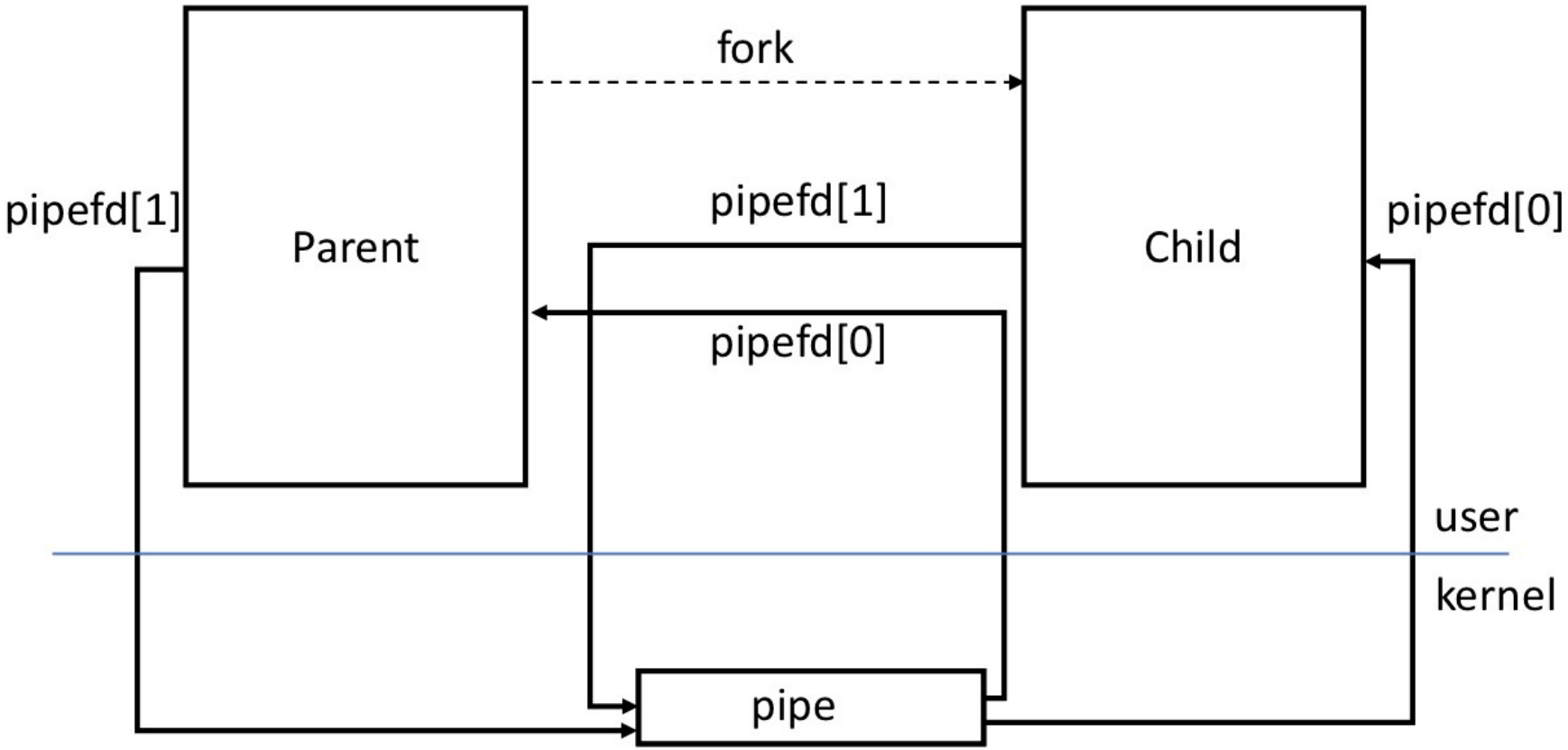
- The pipe call returns two file descriptors corresponding to the read and write ends of the pipe.
- The first file descriptor (*pipefd[0]*) refers to the read end of the pipe (can be used for reading data from the pipe) and the second file descriptor (*pipefd[1]*) refers to the write end of the pipe (can be used for writing data to the pipe).
- The kernel buffers the data written to the pipe until it is read from the read end of the pipe. When there is an error in creating the pipe, it returns -1 and sets the corresponding *errno*, otherwise it returns 0 on success.

- The diagram below illustrates the creation of a pipe in a single process.

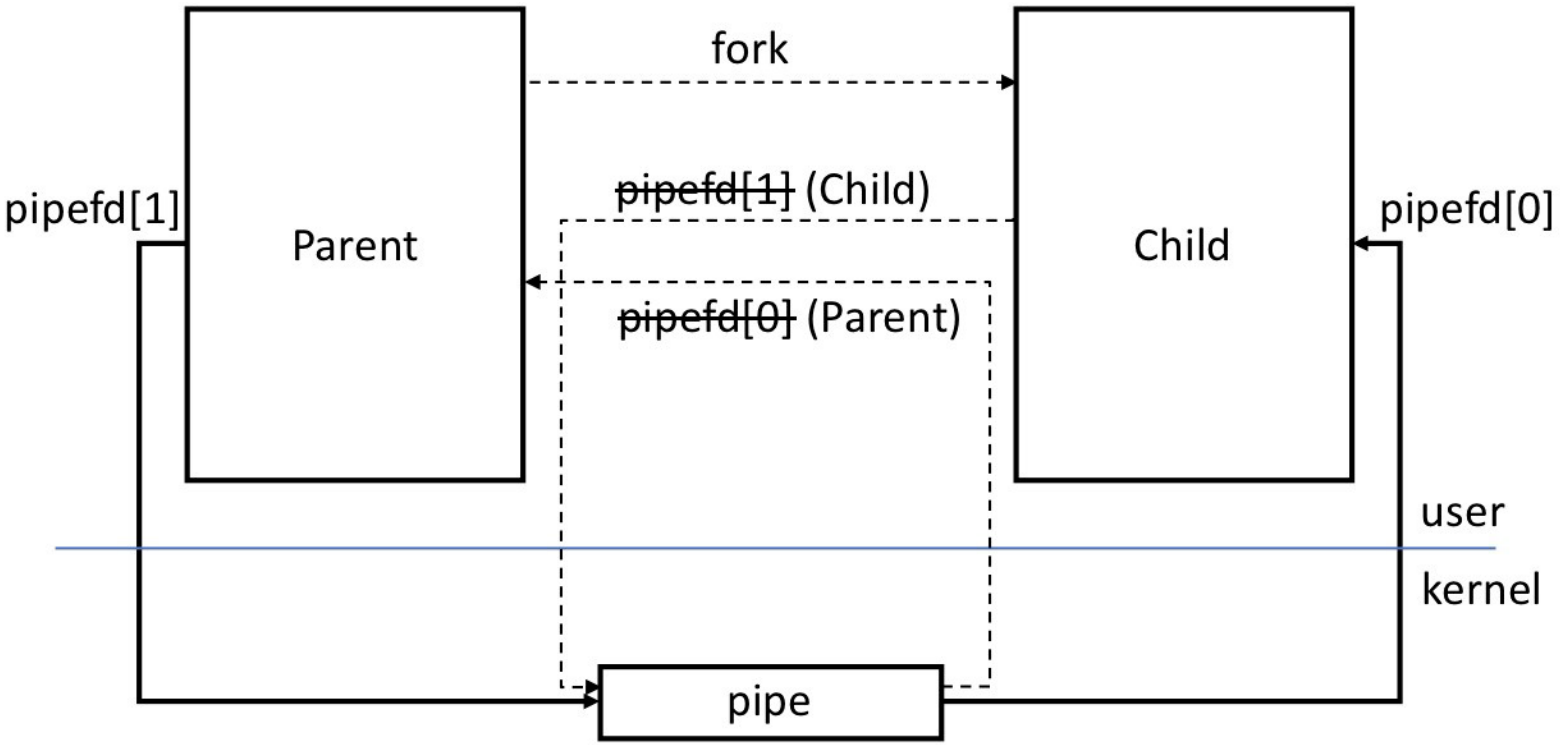


- We really don't need to create a pipe to communicate within the same process, typically we use pipes to communicate between a parent process and a child process.
- In such a case, first a pipe is created by the parent process and then it creates a child process using the fork command.
- Since fork creates a copy of the parent process, the child process will also inherit the all open file descriptors and will have access to the pipe

parent - child



- To provide a unidirectional data channel for communication between the two process, the parent process closes the read end of the pipe and the child process closes the write end of the pipe as shown in the diagram below.



Exercise 1

- The following example shows the steps involved in creating a pipe, forking a child process, closing the file descriptors in the parent and child process, and communication between the parent and child process.
- The parent process writes the string passed as the command-line argument to the pipe and the child process reads the string from the pipe, converts the string to uppercase, and prints it to the standard output.
- The read and write functions that operate on files are used to read and write data from the pipe.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
  pid_t pid;
  int status;
  int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
  char c;

  if (argc != 2) {
     printf("Usage: %s <string>\n", argv[0]);
     exit(-1);
  }

  if (pipe(pipefd) == 0) { /* Open a pipe */
    if ((pid = fork()) == 0) { /* I am the child process */
      close(pipefd[1]);    /* close write end */

```

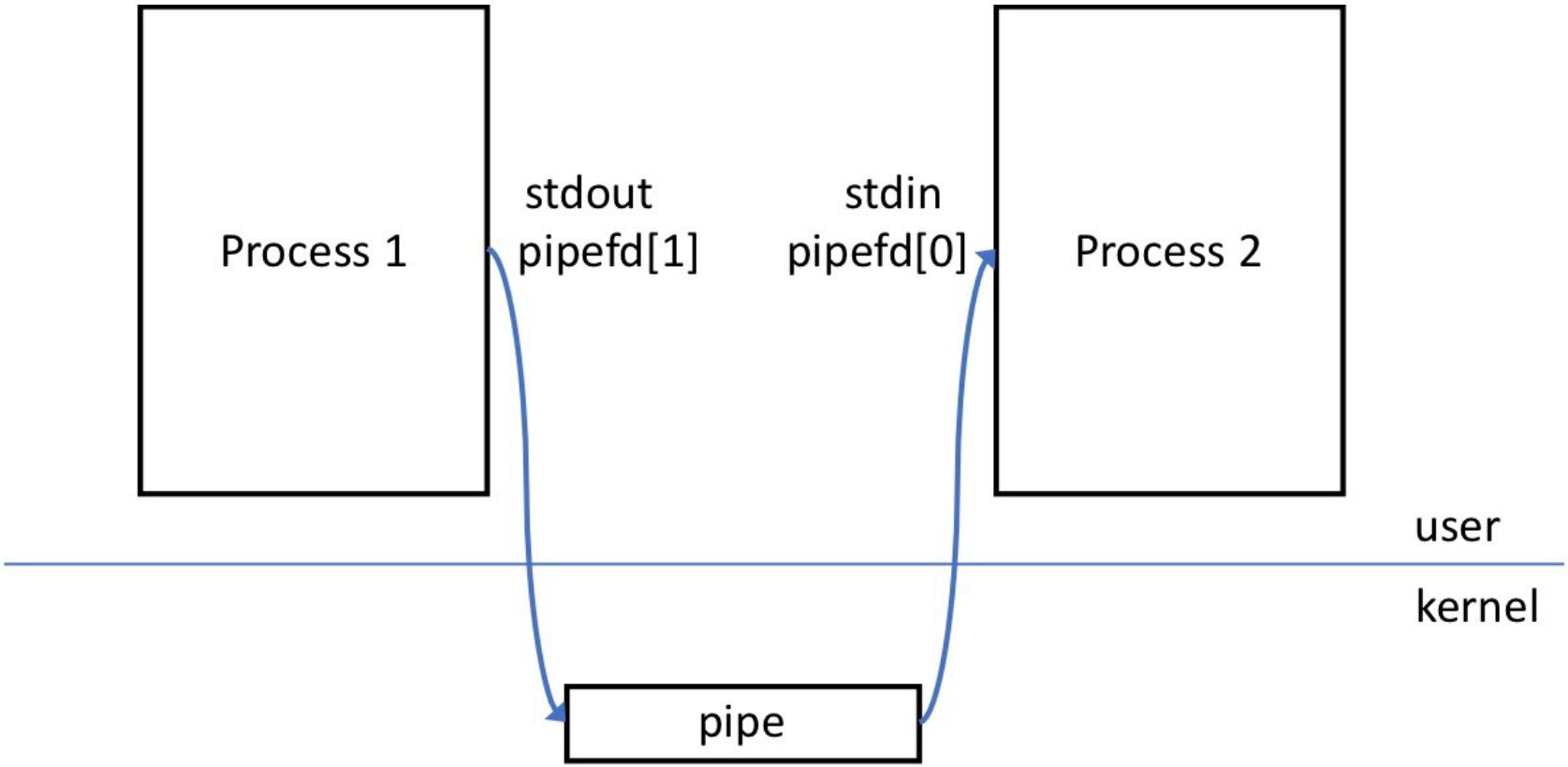
```
23
24          while (read(pipefd[0], &c, 1) > 0) {
25              c = toupper(c);
26              write(1, &c, 1);
27          }
28          write(1, "\n", 1);
29          close(pipefd[0]);
30
31          exit(EXIT_SUCCESS);
32      } else if (pid > 0) { /* I am the parent process */
33          close(pipefd[0]);   /* close read end */
34
35          write(pipefd[1], argv[1], strlen(argv[1]));
36          close(pipefd[1]);
37
38          wait(&status);      /* wait for child to terminate */
39          if (WIFEXITED(status))
40             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
```

```
41          else
42              printf("Child process did not terminate normally!\n");
43        } else { /* we have an error in fork */
44          perror("fork");
45          exit(EXIT_FAILURE);
46        }
47      } else {
48        perror("pipe");
49        exit(EXIT_FAILURE);
50      }
51
52      exit(EXIT_SUCCESS);
53    }
54
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc -Wall pipe1.c -o pipe1

(base) mahmutunan@MacBook-Pro lecture28 % ./pipe1 cs332lowercase
CS332LOWERCASE
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture28 %
```

- The above example shows how the pipe is used by a parent and a child process to communicate.
- We can further extend this to implement pipes between any two programs such that the output of one program is redirected to the input of another program (e.g., ps -elf | grep ssh).
- In order to do this, we have to replace the standard output of the first program with the write end of the pipe and replace the standard input of the second program with the read end of the pipe.
- We have seen in the previous lecture/lab that this can be done using the dup2 system call.
- We will use the dup2 to perform this redirection and implement the pipe operation between two processes

- The pipe operation between two processes as shown in the diagram below.



- We have several options to create the two processes, some of the possible options include:

1. The parent process creates a child process, the child process uses exec to launch the second program, and the parent process will use exec to launch the first program.
2. The parent process creates two child process, the first child process uses exec to launch the first program, the second child process uses exec to launch the second program, and the parent process waits for the two child processes to terminate.
3. The parent process create a child process, the child process creates another child process which in turn uses exec to launch the first program, then the child process uses exec to launch the second program, and the parent waits for the child process to terminate.

In this example, we will use the first option

Exercise 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
    pid_t pid;
    int pipefd[2]; /* fildes[0] for read, fildes[1] for write */

    if (argc != 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == 0) { /* Open a pipe */
      pid = fork(); /* fork child process to execute command2 */
      if (pid == 0) { /* this is the child process */
        /* close write end of the pipe */
    close(pipefd[1]);

        /* replace stdin with read end of pipe */
    if (dup2(pipefd[0], 0) == -1) {
           perror("dup2");
           exit(EXIT_FAILURE);
        }
```

Exercise 2

```
28
29          /* execute <command2> */
30          execlp(argv[2], argv[2], (char *)NULL);
31          perror("execlp");
32          exit(EXIT_FAILURE);
33        } else if (pid > 0) { /* this is the parent process */
34          /* close read end of the pipe */
35      close(pipefd[0]);
36
37      /* replace stdout with write end of pipe */
38      if (dup2(pipefd[1], 1) == -1) {
39              perror("dup2");
40              exit(EXIT_FAILURE);
41          }
42
43          /* execute <command1> */
44          execlp(argv[1], argv[1], (char *)NULL);
45          perror("execlp");
46          exit(EXIT_FAILURE);
47        } else if (pid < 0) { /* we have an error */
48          perror("fork"); /* use perror to print the system error message */
49          exit(EXIT_FAILURE);
50        }
51      } else {
52        perror("pipe");
53        exit(EXIT_FAILURE);
54      }
55
56      return 0;
57  }
```

compile & run

pipe1.c | pipe0.c | p2.c | p1.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int a = 15, b = 25;
    printf("%d\n", a+b);

}
```

pipe1.c | pipe0.c | p2.c | p1.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x;
    printf("Enter an int \n");
    scanf("%d",&x);
    printf("square of your number is %d \n", x*x);

}
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture28 % ./p1
40
(base) mahmutunan@MacBook-Pro lecture28 % gcc p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture28 % ./p2
Enter an int
5
square of your number is 25
(base) mahmutunan@MacBook-Pro lecture28 % gcc pipe0.c -o pipe0
(base) mahmutunan@MacBook-Pro lecture28 % ./pipe0 ./p1 ./p2
Enter an int
square of your number is 1600
(base) mahmutunan@MacBook-Pro lecture28 %
```

- We follow the same approach as the example above and create a pipe in the parent process.
- First we create a child process, close the read end of the pipe, replace the standard output stream with the write end of the pipe using dup2 system call, and then use exec to launch the first program.
- Then we create another child process, close the write end of the pipe, replace the standard input stream with the read end of the pipe using dup2 system call, and then use exec to launch the second program.
- The parent then closes both ends of the pipe and uses the waitpid system call to wait for both child process to terminate.
- The name of the two programs to be executed is provided as command-line arguments to this program.

pipe2.c

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
    pid_t pid1, pid2;
    int pipefd[2]; /* fildes[0] for read, fildes[1] for write */
    int status1, status2;

    if (argc != 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == 0) { /* Open a pipe */

      pid1 = fork(); /* fork first process to execute command1 */
      if (pid1 == 0) { /* this is the child process */
        /* close read end of the pipe */
    close(pipefd[0]);

    /* replace stdout with write end of pipe */
    if (dup2(pipefd[1], 1) == -1) {
            perror("dup2");
            exit(EXIT_FAILURE);
        }
```

```
31
32          /* execute <command1> */
33          execlp( file: argv[1],  arg0: argv[1], (char *)NULL);
34          printf("If you see this statement then exec failed ;-(\n");
35          perror("execlp");
36          exit(EXIT_FAILURE);
37
38        } else if (pid1 < 0) { /* we have an error */
39          perror("fork"); /* use perror to print the system error message */
40          exit(EXIT_FAILURE);
41        }
42
43        pid2 = fork(); /* fork second process to execute command2 */
44        if (pid2 == 0) { /* this is child process */
45          /* close write end of the pipe */
46      close(pipefd[1]);
47
48          /* replace stdin with read end of pipe */
49      if (dup2(pipefd[0], 0) == -1) {
50             perror("dup2");
51             exit(EXIT_FAILURE);
52          }
53
54          /* execute <command2> */
55          execlp( file: argv[2],  arg0: argv[2], (char *)NULL);
56          printf("If you see this statement then exec failed ;-(\n");
57          perror("execlp");
58          exit(EXIT_FAILURE);
```

```
60        } else if (pid2 < 0) { /* we have an error */
61          perror("fork"); /* use perror to print the system error message */
62          exit(EXIT_FAILURE);
63        }
64
65        /* close the pipe in the parent */
66        close(pipefd[0]);
67        close(pipefd[1]);
68
69        /* wait for both child processes to terminate */
70        waitpid(pid1, &status1, 0);
71        waitpid(pid2, &status2, 0);
72      } else {
73        perror("pipe");
74        exit(EXIT_FAILURE);
75      }
76
77      return 0;
78  }
79
```

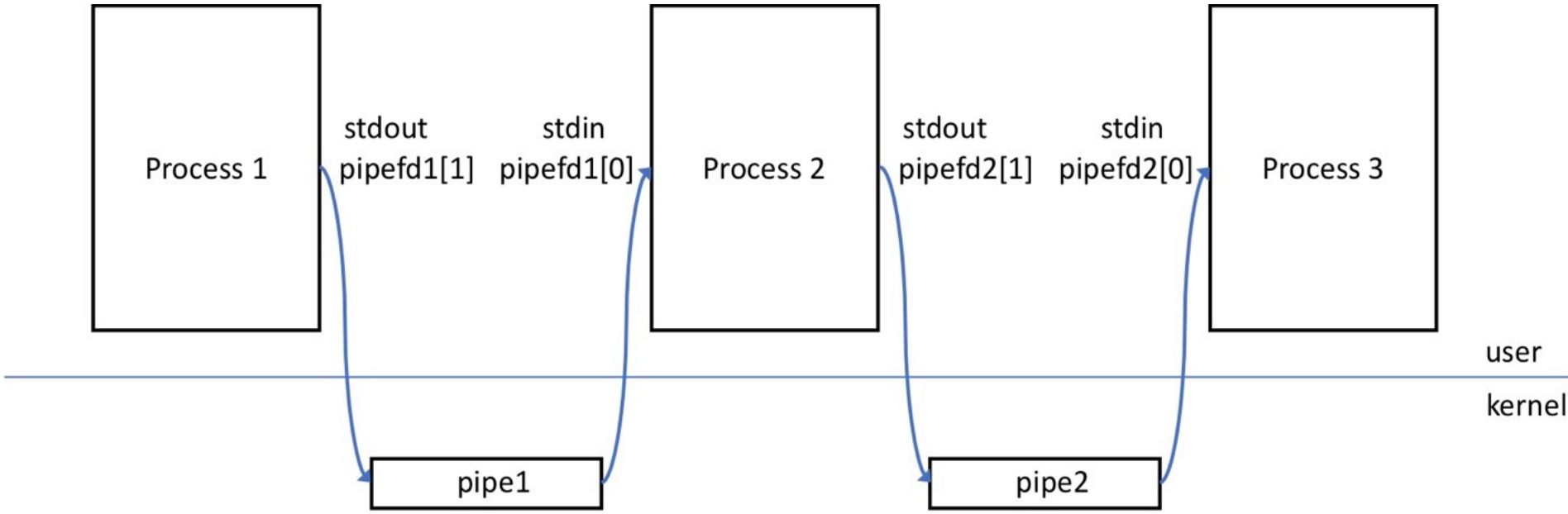
compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ls
p1              pager.c         pipe1           pipe3.c
p1.c            pager2.c        pipe1.c         popen.c
p2              pipe0           pipe2.c
p2.c            pipe0.c         pipe2a.c
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe2.c -o pipe2

(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2 ls sort
p1
p1.c
p2
p2.c
pager.c
pager2.c
pipe0
pipe0.c
pipe1
pipe1.c
pipe2
pipe2.c
pipe2a.c
pipe3.c
popen.c
(base) mahmutunan@MacBook-Pro lecture29 % _
```

extend to three process

- We can extend this to three processes if we like to implement something like: *ls | sort | wc.*
- We will create three processes and use two pipes - one for communication between the first and second process and one for communication between second and third process.
- The code for the first and third children will be similar to the example above while the second child has to replace both standard input and standard output streams instead of just one. T

The diagram below illustrates this



pipe3.c

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
    pid_t pid1, pid2, pid3;
    int pipefd1[2]; /* pipefd1[0] for read, pipefd1[1] for write */
    int pipefd2[2]; /* pipefd2[0] for read, pipefd2[1] for write */
    int status1, status2, status3;

    if (argc != 4) {
        printf("Usage: %s <command1> <command2> <command3>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd1) != 0) { /* Open pipefd1 */
      perror("pipe");
      exit(EXIT_FAILURE);
    }

    if (pipe(pipefd2) != 0) { /* Open pipefd2 */
      perror("pipe");
      exit(EXIT_FAILURE);
    }

    pid1 = fork(); /* fork first process to execute command1 */
    if (pid1 == 0) { /* this is the child process */
      /* close read end of the pipefd1 */
      close(pipefd1[0]);

      /* close both ends of pipefd2 */
      close(pipefd2[0]);
      close(pipefd2[1]);
```

```
39          /* replace stdout with write end of pipefd1 */
40          if (dup2(pipefd1[1], 1) == -1) {
41      perror("dup2");
42      exit(EXIT_FAILURE);
43          }
44
45          /* execute <command1> */
46          execlp( file: argv[1],  arg0: argv[1], (char *)NULL);
47          perror("execlp");
48          exit(EXIT_FAILURE);
49
50      } else if (pid1 < 0) { /* we have an error */
51          perror("fork"); /* use perror to print the system error message */
52          exit(EXIT_FAILURE);
53      }
54
55      pid2 = fork(); /* fork second process to execute command2 */
56      if (pid2 == 0) { /* this is child process */
57          /* close write end of the pipefd1 */
58          close(pipefd1[1]);
59
60          /* replace stdin with read end of pipefd2 */
61          if (dup2(pipefd1[0], 0) == -1) {
62      perror("dup2");
63      exit(EXIT_FAILURE);
64          }
65
66          /* close read end of pipefd2 */
67          close(pipefd2[0]);
68
69          /* replace stdout with write end of pipefd2 */
70          if (dup2(pipefd2[1], 1) == -1) {
71      perror("dup2");
72      exit(EXIT_FAILURE);
73          }
74
75          /* execute <command2> */
76          execlp( file: argv[2],  arg0: argv[2], (char *)NULL);
77          perror("execlp");
78          exit(EXIT_FAILURE);
```

```c
    } else if (pid2 < 0) { /* we have an error */
      perror("fork"); /* use perror to print the system error message */
      exit(EXIT_FAILURE);
    }

    pid3 = fork(); /* fork third process to execute command3 */
    if (pid3 == 0) { /* this is child process */
      /* close both ends of the pipefd1 */
      close(pipefd1[0]);
      close(pipefd1[1]);

      /* close write end of pipefd2 */
      close(pipefd2[1]);

      /* replace stdin with read end of pipefd2 */
      if (dup2(pipefd2[0], 0) == -1) {
    perror("dup2");
    exit(EXIT_FAILURE);
      }

      /* execute <command3> */
      execlp( file: argv[3],  arg0: argv[3], (char *)NULL);
      perror("execlp");
      exit(EXIT_FAILURE);

    } else if (pid3 < 0) { /* we have an error */
      perror("fork"); /* use perror to print the system error message */
      exit(EXIT_FAILURE);
    }

    /* close the pipes in the parent */
    close(pipefd1[0]);
    close(pipefd1[1]);
    close(pipefd2[0]);
    close(pipefd2[1]);

    /* wait for both child processes to terminate */
    waitpid(pid1, &status1, 0);
    waitpid(pid2, &status2, 0);
    waitpid(pid3, &status3, 0);

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe3.c -o pipe3

(base) mahmutunan@MacBook-Pro lecture29 % ./pipe3 ps sort wc
       5      23     161
(base) mahmutunan@MacBook-Pro lecture29 % _
```

pager.c

```c

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

int main(int argc, char **argv) {
  pid_t pid;
  int status;
  int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
  FILE *fp;
  char line[BUFSIZ];
  int n;

  if (argc != 2) {
     printf("Usage: %s <filename>\n", argv[0]);
     exit(-1);
  }

  if ( (fp = fopen( filename: argv[1], mode: "r")) == NULL) {
    printf("Error opening file %s for reading\n", argv[1]);
    exit(-1);
  }

  if (pipe(pipefd) == 0) { /* Open a pipe */
    if ((pid = fork()) == 0) { /* I am the child process */
      close(pipefd[1]);    /* close write end */
      dup2(pipefd[0], STDIN_FILENO); /* replace stdin of child */
      execlp( file: "/usr/bin/more", arg0: "more", (char *)NULL);
      perror("exec");
      exit(EXIT_FAILURE);
```

```
34          exit(EXIT_FAILURE);
35        } else if (pid > 0) { /* I am the parent process */
36          close(pipefd[0]);   /* close read end */
37          /* read lines from the file and write it to pipe */
38          while (fgets(line, BUFSIZ, fp) != NULL) {
39          n = strlen(line);
40              if (write( fd: pipefd[1], line, n) != n) {
41            printf("Error writing to pipe\n");
42            exit(-1);
43          }
44          }
45          close(pipefd[1]);   /* close write end */
46
47          wait(&status);      /* wait for child to terminate */
48          if (WIFEXITED(status))
49             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
50          else
51             printf("Child process did not terminate normally!\n");
52        } else { /* we have an error in fork */
53          perror("fork");
54          exit(EXIT_FAILURE);
55        }
56      } else {
57        perror("pipe");
58        exit(EXIT_FAILURE);
59      }
60
61      exit(EXIT_SUCCESS);
62  }
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pager smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
period that some of its noisiest authorities insisted on its
being received for good or for evil in the superlative degree
of comparison only

there were a king with a large jaw and a queen with a plain face
on the throne of england there were a king with a large jaw and
a queen with a fair face on the throne of france  in both
```

popen and pclose functions

- As we have seen in the examples, the common usage of pipes involve creating a pipe, creating a child process with fork, closing the unused ends of the pipe, execing a command in the child process, and waiting for the child process to terminate in the parent process.
- Since this is such a common usage, UNIX systems provide *popen* and *pclose* functions that perform most of these operations in a single operation.
- The C APIs for the *popen* and *pclose* functions are shown below:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- The popen function performs the following steps:
- creates a pipe
- creates a new process using fork
- perform the following steps in the child process
  - close unused ends of the pipe (based on the *type* argument)
  - execs a shell to execute the *command* provided as argument to popen (i.e., executes "sh -c command")
- perform the following steps in the parent process
  - close unused ends of the pipe (based on the *type* argument)
  - wait for the child process to terminate

- The *popen* function returns the FILE handle to the pipe created so that the calling process can read or write to the pipe using standard I/O system calls.
- If the *type* argument is specified as read-only ("r") then the calling process can read from the pipe, this results in reading from the *stdout* of the child process (see Figure 15.9).
- If the type argument is specifies as write-only ("w") then the calling process can write to the pipe, this results in writing to the *stdin* of the child process created (see Figure 15.10).

- The FILE handle returned by *popen* must be closed using *pclose* to make sure that the I/O stream opened to read or write to the pipe is closed and wait for the child process to terminate.
- The termination status of the shell started by exec will be returned when the *pclose* function returns.
-

pipe2a.c

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

int main(int argc, char **argv) {
    FILE *fp1, *fp2;
    char line[BUFSIZ];

    if (argc != 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* create a pipe, fork/exec command argv[1], in "read" mode */
    /* read mode - parent process reads stdout of child process */
    if ((fp1 = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

```

```
19
20      /* create a pipe, fork/exec command argv[2], in "write" mode */
21      /* write mode - parent process writes to stdin of child process */
22      if ((fp2 = popen(argv[2], "w")) == NULL) {
23      perror("popen");
24      exit(EXIT_FAILURE);
25      }
26
27      /* read stdout from child process 1 and write to stdin of
28         child process 2 */
29      while (fgets(line, BUFSIZ, fp1) != NULL) {
30      if (fputs(line, fp2) == EOF) {
31         printf("Error writing to pipe\n");
32         exit(EXIT_FAILURE);
33          }
34      }
35
36      /* wait for child process to terminate */
37      if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
38      perror("pclose");
39      exit(EXIT_FAILURE);
40      }
41
42      return 0;
43  }
44
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2a "ls -l" sort
-rw-r--r--@ 1 mahmutunan  staff    105 Nov  2 13:37 p1.c
-rw-r--r--@ 1 mahmutunan  staff    169 Nov  2 13:36 p2.c
-rw-r--r--@ 1 mahmutunan  staff    790 Oct 27 22:41 popen.c
-rw-r--r--@ 1 mahmutunan  staff   1694 Oct 27 22:41 pager2.c
-rw-r--r--@ 1 mahmutunan  staff   1853 Nov  4 13:07 pipe2a.c
-rw-r--r--@ 1 mahmutunan  staff   2073 Oct 27 22:41 pipe1.c
-rw-r--r--@ 1 mahmutunan  staff   2121 Oct 27 22:41 pipe0.c
-rw-r--r--@ 1 mahmutunan  staff   2284 Nov  4 12:44 pager.c
-rw-r--r--@ 1 mahmutunan  staff   2782 Nov  4 11:31 pipe2.c
-rw-r--r--@ 1 mahmutunan  staff   3858 Nov  4 11:51 pipe3.c
-rw-r--r--@ 1 mahmutunan  staff   5074 Nov  4 12:51 smalltale.txt
-rwxr-xr-x  1 mahmutunan  staff  12556 Nov  2 13:37 p1
-rwxr-xr-x  1 mahmutunan  staff  12604 Nov  2 13:37 p2
-rwxr-xr-x  1 mahmutunan  staff  12952 Nov  4 13:07 pipe2a
-rwxr-xr-x  1 mahmutunan  staff  12984 Nov  2 13:37 pipe0
-rwxr-xr-x  1 mahmutunan  staff  12996 Nov  2 13:20 pipe1
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 11:31 pipe2
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 12:41 pipe3
-rwxr-xr-x  1 mahmutunan  staff  13076 Nov  4 12:44 pager
total 352
```

- Note that since the command is executed using a shell, we can provide wildcards and other special characters that the shell can expand.
- Also note that in this version of the program the parent process is reading the *stdout* stream of the first child process and then writing to the *stdin* stream of the second child process (we did not do this in the first version).

Here is an updated version of the pager program that uses popen and pclose

pager2.c

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

int main(int argc, char **argv) {
  FILE *fpin, *fpout;
  char line[BUFSIZ];

  if (argc != 2) {
     printf("Usage: %s <filename>\n", argv[0]);
     exit(-1);
  }

  /* open file for reading */
  if ( (fpin = fopen( filename: argv[1],  mode: "r")) == NULL ) {
    printf("Error opening file %s for reading\n", argv[1]);
    exit(-1);
  }

  /* create a pipe, fork/exec process "more", in "write" mode */
  /* write mode - parent process writes, child process reads */
  if ( (fpout = popen("more", "w")) == NULL ) {
      perror("exec");
      exit(EXIT_FAILURE);
  }
```

```
    /* read lines from the file and write it fpout */
    while (fgets(line, BUFSIZ, fpin) != NULL) {
        if (fputs(line, fpout) == EOF) {
            printf("Error writing to pipe\n");
      exit(EXIT_FAILURE);
        }
    }

    /* close the pipe and wait for child process to terminate */
    if (pclose(fpout) == -1) {
        perror("pclose");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pager2.c -o page
r2
(base) mahmutunan@MacBook-Pro lecture29 % ./pager2 smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
```

popen.c

- You can also find a simpler version of the program that uses a single popen system call to create a pipe in "read" mode, execute the command specified as the command-line argument, reads the pipe and prints it to stdout

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

int main(int argc, char **argv) {
    FILE *fp;
    char line[BUFSIZ];

    if (argc != 2) {
        printf("Usage: %s <command>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    if ((fp = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

    while (fgets(line, BUFSIZ, fp) != NULL) {
      fputs(line, stdout);
    }

    if (pclose(fp) == -1) {
    perror("pclose");
    exit(EXIT_FAILURE);
    }

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall popen.c -o popen

(base) mahmutunan@MacBook-Pro lecture29 % ./popen ps
  PID TTY           TIME CMD
 1600 ttys000    0:00.46 -zsh
26658 ttys000    0:00.00 ./popen ps
(base) mahmutunan@MacBook-Pro lecture29 %
```

THREAD

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process

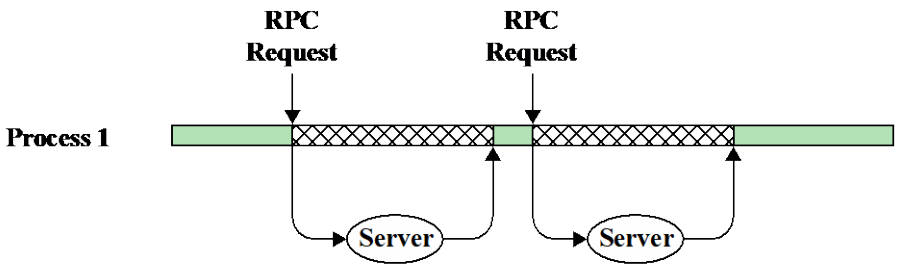
Thread Execution States

The key states for a thread are:

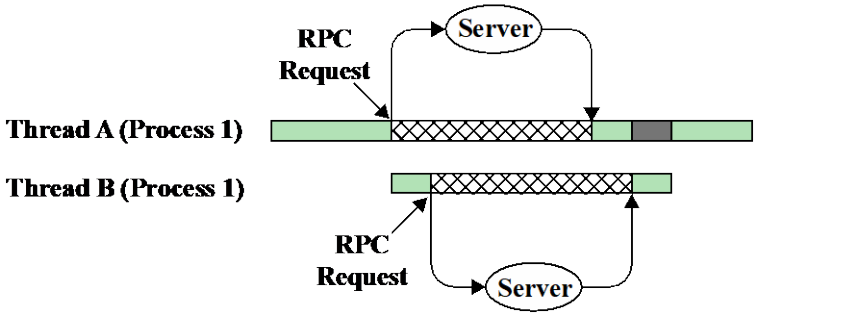
- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish



(a) RPC Using Single Thread



(b) RPC Using One Thread per Server (on a uniprocessor)

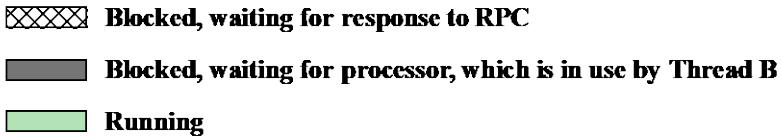


Figure 4.3 Remote Procedure Call (RPC) Using Threads

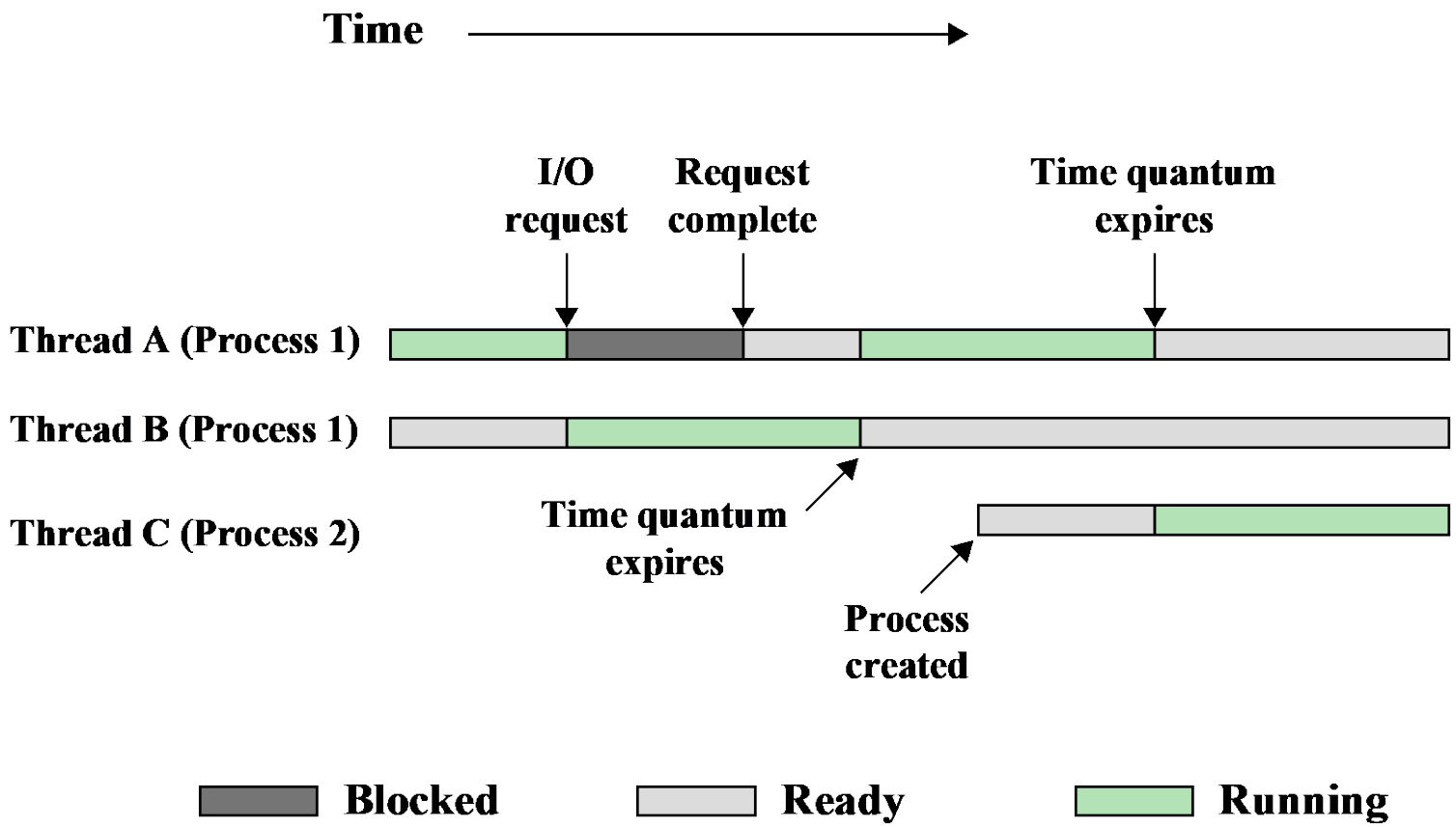
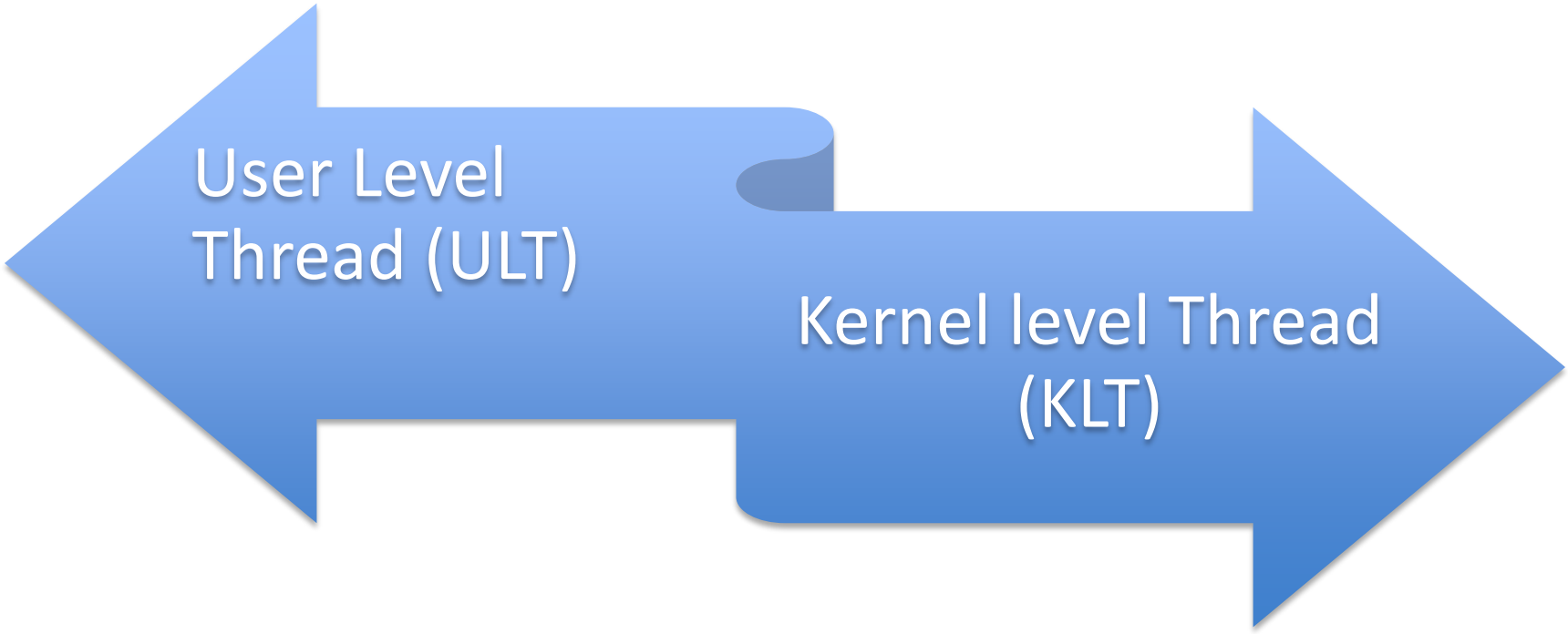


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

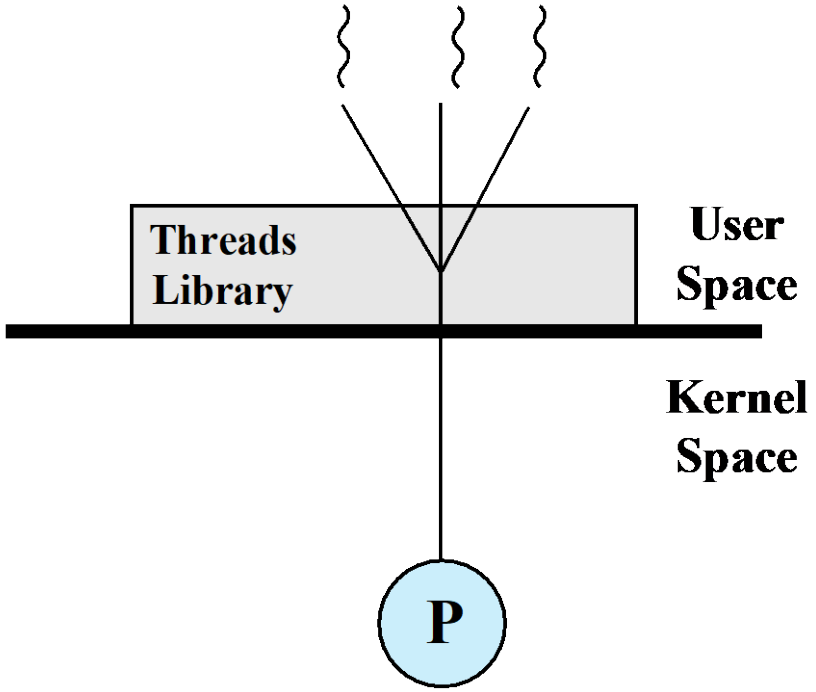
- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads

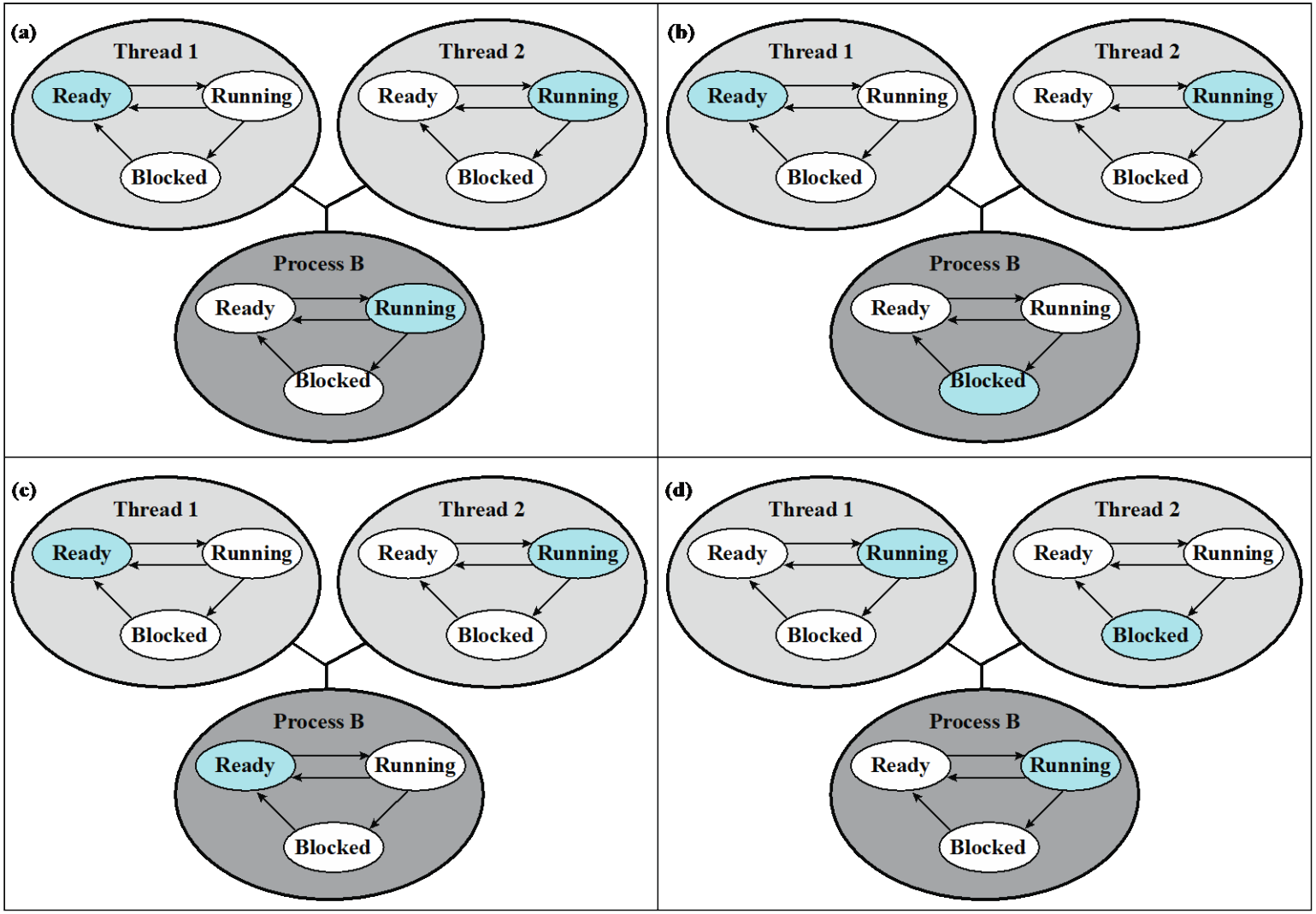


User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



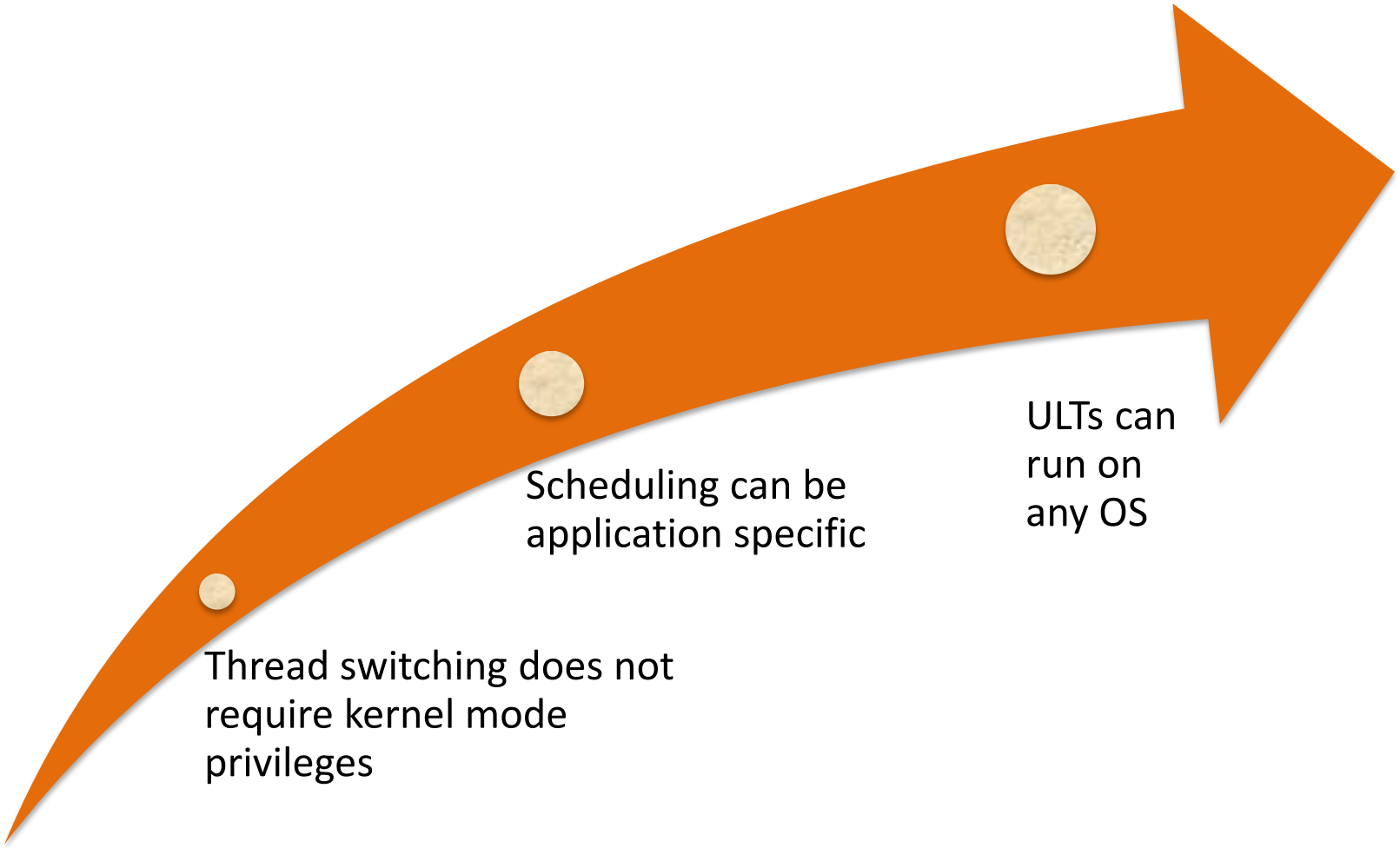
(a) Pure user-level



Colored state
is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs



Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Overcoming ULT Disadvantages

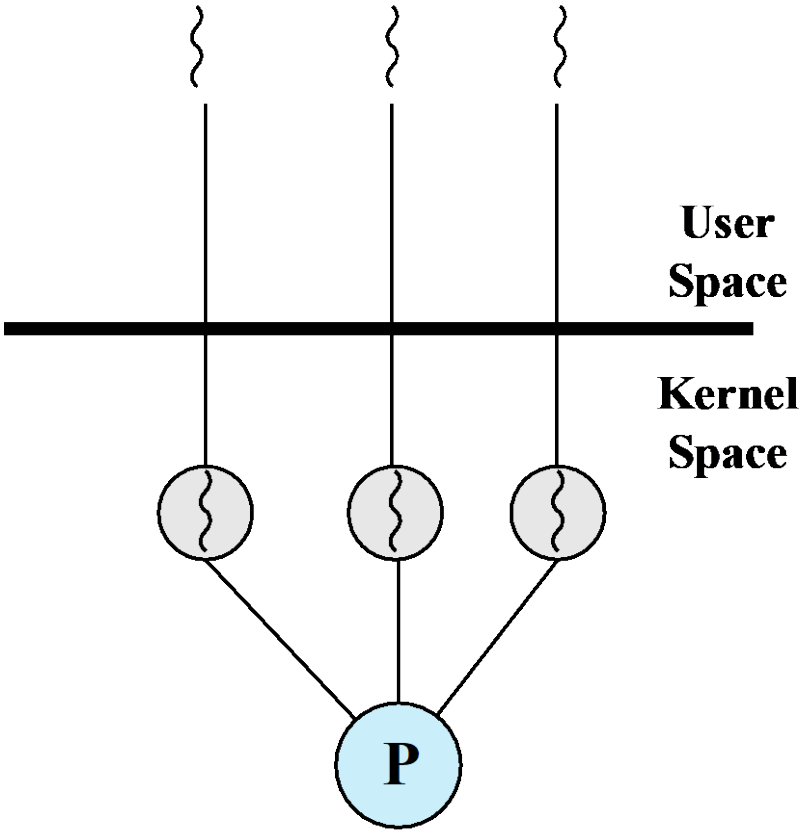
Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
    - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μs)

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

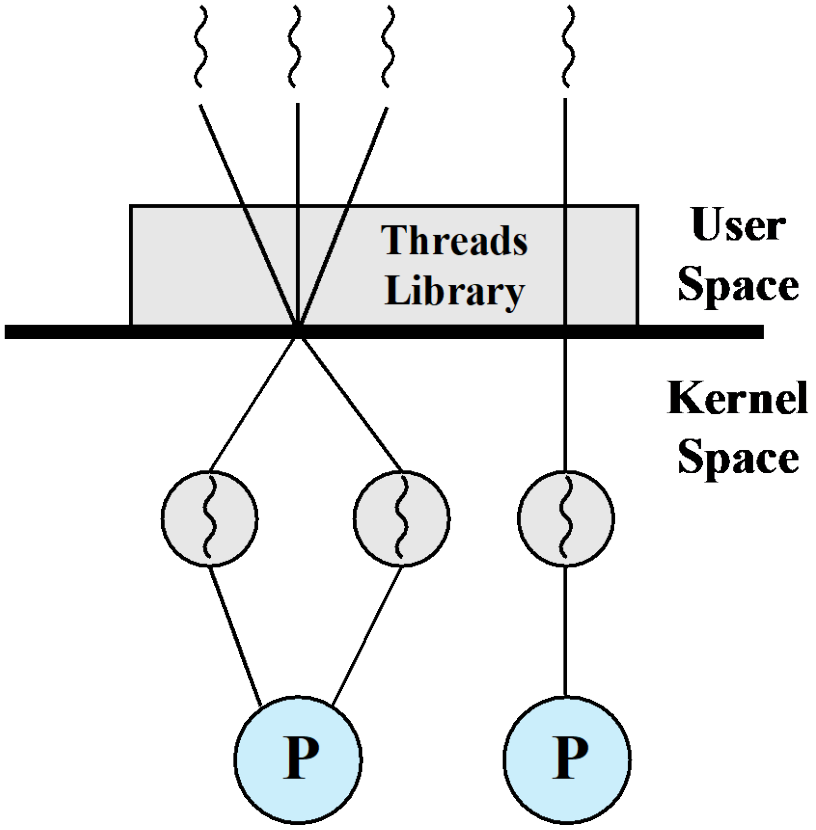
https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Multithreading Models

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



(c) Combined

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Table 4.2
Relationship between Threads and Processes

Create threads using POSIX threads library

- In the previous lectures/labs we focused on how to create processes, in this lab we will focus on creating threads and mechanisms for establishing synchronization among threads.
- First, let us understand the difference between a process and a thread.
  - A process could be considered to have two characteristics:
    - (a) resource ownership
    - (b) scheduling or execution.
- The unit of scheduling and dispatching is usually referred to as a **thread** or **lightweight process and** the ability of to support multiple, concurrent paths of execution within a single process is often referred to as *multithreading*.

- Threads offer several benefits compared to a process:
  - Threads takes less time to create a new thread than a process
  - Threads take less time to terminate a thread than a process
  - Switching between two threads (context switching) takes less time than switching between processes
  - All of the threads in a process share the state and resources of that process (since threads reside in the same address space and have access to the same data)
  - Threads enhance efficiency in communication between programs (since threads share memory and files within the same process and can communicate without invoking the kernel)

- As a result of these advantages, if we have to implement a set of functions that are closely related, implementing this functionality using multiple threads is far more efficient than using multiple processes.
- We will use the **POSIX threads library**, usually referred to as Pthreads library, that provides C APIs to create and manage threads. We have to include the file *pthread.h* and link with *-lpthread* to compile and link.
- We can create new threads using the *pthread_create()* function which has the following function definition:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- The new thread that will be created by the *pthread_create* function will invoke the function *start_routine*.
- Note that the function *start_routine* takes one argument of type *void* * and has the return type as *`void *`*.
- In other words, the function *start_routine* has the following function definition:

```
void *start_routine(void *arg)
```

- When the *pthread_create* call returns successfully, it returns the thread ID associated with the new thread created in the variable thread.
- This can be used by the main thread in subsequent pthread function calls such as *pthread_join*.
- The second argument, *attr,* provides a reference to the *pthread_attr_t* structure that describes the various attributes of the new thread to be created.
- It can be initialized using *pthread_attr_init* call or set to NULL if default attributes must be used.
- You can find out more about the different thread attributes that can be specified by looking at the man page for *pthread_attr_init*.

- The new thread created will terminate when the function *start_routine* returns or when a call to *pthread_exit* is made inside the *start_routine*.
- We can use the *pthread_join* function to wait for a thread to complete using the thread ID that was returned when *pthread_create* call was invoked.
- If a thread has already completed,
  - *pthread_join* will return immediately, otherwise, it will wait for the corresponding thread to complete.

exercise 1

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


void *someFuncToCreateThread(void *someValue)
{
    sleep(2);
    printf("I am inside the thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("I am inside the main function\n");
    pthread_create(&thread_id, NULL, someFuncToCreateThread, NULL);
    pthread_join(thread_id, NULL);
    printf("Back to the main function\n");
    exit(0);
}

```

compile & run

To compile a multithreaded program, we will be using gcc and we need to link it with the pthreads library.

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise1.c -o exercise1 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise1
I am inside the main function
I am inside the thread
Back to the main function
(base) mahmutunan@MacBook-Pro lecture31 %
```

exercise 2

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

void *function1(void *someValue)
{
    while(1==1) {
        sleep(1);
        printf("function 1 \n");
    }
}
void function2()
{
    while(1==1) {
        sleep(2);
        printf("function 2\n");
    }
}

int main()
{
    pthread_t thread_id;
    printf("I am inside the main function\n");
    pthread_create(&thread_id, NULL, function1, NULL);
    function2();
    exit(0);
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise2.c -o exercise2 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise2
I am inside the main function
function 1
function 2
function 1
function 1
function 2
function 1
function 1
function 2
function 1
function 1
function 2
function 1
function 1
```

exercise 3

```c
int globalVar = 50; //define a global variable

void *someFuncToCreateThread(void *someValue)
{
    int *threadId = (int *)someValue; // Store the value argument passed to this thread

    //define a static and a local variable
    static int staticVar = 75;
    int localVar = 10;

    // let's change the variables
    globalVar +=100;
    staticVar +=100;
    localVar +=100;
    printf("id =%d,global = %d,  local = %d, static =%d, \n", *threadId, globalVar,localVar ,staticVar);

    return NULL;
}

int main()
{
    int i;
    pthread_t thread_id;
    for (i = 0; i < 4; i++)
        pthread_create(&thread_id, NULL, someFuncToCreateThread, (void *)&thread_id);
    pthread_exit(NULL);

}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc exercise3.c -o exercise3 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise3
id =151261184,global = 150,  local = 110, static =175,
id =151261184,global = 150,  local = 110, static =175,
id =151261184,global = 250,  local = 110, static =275,
id =151261184,global = 250,  local = 110, static =275,
(base) mahmutunan@MacBook-Pro lecture31 % _
```

Remember, global and static variables are stored in data segment.
All threads share data segment, so they are shared by all threads.

pthread1.c

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

int nthreads;

void *compute(void *arg) {
  long tid = (long)arg;

  printf("Hello, I am thread %ld of %d\n", tid, nthreads);

  return (NULL);
}

int main(int argc, char **argv) {
  long i;
  pthread_t *tid;

  if (argc != 2) {
    printf("Usage: %s <# of threads>\n",argv[0]);
    exit(-1);
  }

  nthreads = atoi(argv[1]); // no. of threads
```

```
25
26      // allocate vector and initialize
27      tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
28
29      // create threads
30      for ( i = 0; i < nthreads; i++)
31        pthread_create(&tid[i], NULL, compute, (void *)i);
32
33      // wait for them to complete
34      for ( i = 0; i < nthreads; i++)
35        pthread_join(tid[i], NULL);
36
37      printf("Exiting main program\n");
38
39      return 0;
40    }
41
```

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc pthread1.c -o exercise4 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise4 4
Hello, I am thread 0 of 4
Hello, I am thread 1 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
```

pthread2.c

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

int nthreads;

void *compute(void *arg) {
  long tid = (long)arg;
  pthread_t pthread_id = pthread_self();

  printf("Hello, I am thread %ld of %d, pthread_self() = %lu (0x%lx)\n",
         tid, nthreads, (unsigned long)pthread_id, (unsigned long)pthread_id);

  return (NULL);
}

int main(int argc, char **argv) {
  long i;
  pthread_t *tid;
  pthread_t pthread_id = pthread_self();
```

```
21
22    if (argc != 2) {
23      printf("Usage: %s <# of threads>\n",argv[0]);
24      exit(-1);
25    }
26
27    nthreads = atoi(argv[1]); // no. of threads
28
29    // allocate vector and initialize
30    tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
31
32    // create threads
33    for ( i = 0; i < nthreads; i++)
34      pthread_create(&tid[i], NULL, compute, (void *)i);
35
36    for ( i = 0; i < nthreads; i++)
37      printf("tid[%ld] = %lu (0x%lx)\n", i, tid[i], tid[i]);
38
39    printf("Hello, I am main thread. pthread_self() = %lu (0x%lx)\n",
40           (unsigned long)pthread_id, (unsigned long)pthread_id);
41
42    // wait for them to complete
43    for ( i = 0; i < nthreads; i++)
44      pthread_join(tid[i], NULL);
45
46    printf("Exiting main program\n");
47
48    return 0;
49  }
```

```
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise5 4
tid[0] = 123145541038080 (0x70000e3ab000)
tid[1] = 123145541574656 (0x70000e42e000)
tid[2] = 123145542111232 (0x70000e4b1000)
tid[3] = 123145542647808 (0x70000e534000)
Hello, I am main thread. pthread_self() = 4365594048 (0x10435adc0)
Hello, I am thread 1 of 4, pthread_self() = 123145541574656 (0x70000e42e000)
Hello, I am thread 2 of 4, pthread_self() = 123145542111232 (0x70000e4b1000)
Hello, I am thread 0 of 4, pthread_self() = 123145541038080 (0x70000e3ab000)
Hello, I am thread 3 of 4, pthread_self() = 123145542647808 (0x70000e534000)
Exiting main program
```

pthread3.c

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

typedef struct foo {
    pthread_t ptid;  /* thread id returned by pthread_create */
    int tid;         /* user managed thread id (0 through nthreads-1) */
    int nthreads;    /* total no. of threads created */
} FOO;

void *compute(void *args) {
   FOO *info = (FOO *)args;
   printf("Hello, I am thread %d of %d\n", info->tid, info->nthreads);

   return (NULL);
}

int main(int argc, char **argv) {
   int i, nthreads;
   FOO *info;

   if (argc != 2) {
     printf("Usage: %s <# of threads>\n",argv[0]);
     exit(-1);
   }
```

```
27      nthreads = atoi(argv[1]); // no. of threads
28
29      // allocate structure
30      info = (FOO *)malloc(sizeof(FOO)*nthreads);
31
32      // create threads
33      for ( i = 0; i < nthreads; i++) {
34        info[i].tid = i;
35        info[i].nthreads = nthreads;
36        pthread_create(&info[i].ptid, NULL, compute, (void *)&info[i]);
37      }
38
39      // wait for them to complete
40      for ( i = 0; i < nthreads; i++)
41        pthread_join(info[i].ptid, NULL);
42
43      free(info);
44      printf("Exiting main program\n");
45
46      return 0;
47  }
```

```
(base) mahmutunan@MacBook-Pro lecture31 % gcc pthread3.c -o exercise6 -lpthread
(base) mahmutunan@MacBook-Pro lecture31 % ./exercise6 4
Hello, I am thread 1 of 4
Hello, I am thread 0 of 4
Hello, I am thread 2 of 4
Hello, I am thread 3 of 4
Exiting main program
```

Thread Synchronization using Mutexes

- We can use the mutexes provided by the Pthreads library to control access to critical sections of the program and provide synchronization across the threads.
- We will use the example of computing the sum of the elements in a vector to illustrate the use of mutexes.
- We have a vector of N elements, we would like to assign each thread to compute the partial sum of N/P elements (where P is the number of threads), then we will update the shared global variable *sum* with the partial sums using mutex locks.

- The API for the mutex lock and unlock functions are shown below:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t
*mutex);
```

pthread_sum.c

- The *mutex* variable is of type *pthread_mutex_t* and can be initially statically by assigning the value *PTHREAD_MUTEX_INITIALIZER*.
- Note that the mutex variable must be declared in global scope since it will be shared among multiple threads.
- A mutex can also be initialized dynamically using the function *pthread_mutex_init*.
- The *pthread_mutex_destroy* function can be used to destroy the mutex that was initialized using *pthread_mutex_init*.

pthread_sum.c

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

double *a=NULL, sum=0.0;
int    N, size;

void *compute(void *arg) {
    int myStart, myEnd, myN, i;
    long tid = (long)arg;

    // determine start and end of computation for the current thread
    myN = N/size;
    myStart = tid*myN;
    myEnd = myStart + myN;
    if (tid == (size-1)) myEnd = N;

    // compute partial sum
    double mysum = 0.0;
    for (i=myStart; i<myEnd; i++)
      mysum += a[i];

    // grab the lock, update global sum, and release lock
    pthread_mutex_lock(&mutex);
    sum += mysum;
    pthread_mutex_unlock(&mutex);

    return (NULL);
}
```

pthread_sum.c

```
34  int main(int argc, char **argv) {
35      long i;
36      pthread_t *tid;
37
38      if (argc != 3) {
39         printf("Usage: %s <# of elements> <# of threads>\n",argv[0]);
40         exit(-1);
41      }
42
43      N = atoi(argv[1]); // no. of elements
44      size = atoi(argv[2]); // no. of threads
45
46      // allocate vector and initialize
47      tid = (pthread_t *)malloc(sizeof(pthread_t)*size);
48      a = (double *)malloc(sizeof(double)*N);
49      for (i=0; i<N; i++)
50        a[i] = (double)(i + 1);
51
52      // create threads
53      for ( i = 0; i < size; i++)
54        pthread_create(&tid[i], NULL, compute, (void *)i);
55
56      // wait for them to complete
57      for ( i = 0; i < size; i++)
58        pthread_join(tid[i], NULL);
59
60      printf("The total is %g, it should be equal to %g\n",
61             sum, ((double)N*(N+1))/2);
62
63      return 0;
64  }
```

pthread_sum.c

- compile & run

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum.c -o exercise7 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 200 4
The total is 20100, it should be equal to 20100
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 % _
```

pthread_sum2.c

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

double *a=NULL, *partialsum;
int    N, nthreads;

void *compute(void *arg) {
    int myStart, myEnd, myN, i;
    long tid = (long)arg;

    // determine start and end of computation for the current thread
    myN = N/nthreads;
    myStart = tid*myN;
    myEnd = myStart + myN;
    if (tid == (nthreads-1)) myEnd = N;

    // compute partial sum
    double mysum = 0.0;
    for (i=myStart; i<myEnd; i++)
      mysum += a[i];

    partialsum[tid] = mysum;
    return (NULL);
}

```

pthread_sum2.c

```
27
28  int main(int argc, char **argv) {
29      long i;
30      pthread_t *tid;
31      double sum = 0.0;
32
33      if (argc != 3) {
34         printf("Usage: %s <# of elements> <# of threads>\n",argv[0]);
35         exit(-1);
36      }
37
38      N = atoi(argv[1]); // no. of elements
39      nthreads = atoi(argv[2]); // no. of threads
40
41      // allocate vector and initialize
42      tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
43      a = (double *)malloc(sizeof(double)*N);
44      partialsum = (double *)malloc(sizeof(double)*nthreads);
45      for (i=0; i<N; i++)
46        a[i] = (double)(i + 1);
47
48      // create threads
49      for ( i = 0; i < nthreads; i++)
50        pthread_create(&tid[i], NULL, compute, (void *)i);
51
```

pthread_sum2.c

```
51
52      // wait for them to complete
53      for ( i = 0; i < nthreads; i++)
54        pthread_join(tid[i], NULL);
55
56      for ( i = 0; i < nthreads; i++)
57        sum += partialsum[i];
58
59      printf("The total is %g, it should be equal to %g\n",
60             sum, ((double)N*(N+1))/2);
61
62      free(tid);
63      free(a);
64      free(partialsum);
65
66      return 0;
67  }
68
```

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum2.c -o exercise8 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise8 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise8 200 4
The total is 20100, it should be equal to 20100
```

Extra exercises

```c
pthread_t tid[2];
int counter;

void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise2.c -o exercise2 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise2

 Job 1 has started

 Job 2 has started

 Job 2 has finished

 Job 2 has finished
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>


pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```c
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise3.c -o exercise3 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise3

 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

Thread Synchronization using Mutexes

- We can use the mutexes provided by the Pthreads library to control access to critical sections of the program and provide synchronization across the threads.
- We will use the example of computing the sum of the elements in a vector to illustrate the use of mutexes.
- We have a vector of N elements, we would like to assign each thread to compute the partial sum of N/P elements (where P is the number of threads), then we will update the shared global variable *sum* with the partial sums using mutex locks.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

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

- The API for the mutex lock and unlock functions are shown below:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t
*mutex);
```

pthread_sum.c

- The *mutex* variable is of type *pthread_mutex_t* and can be initially statically by assigning the value *PTHREAD_MUTEX_INITIALIZER*.
- Note that the mutex variable must be declared in global scope since it will be shared among multiple threads.
- A mutex can also be initialized dynamically using the function *pthread_mutex_init*.
- The *pthread_mutex_destroy* function can be used to destroy the mutex that was initialized using *pthread_mutex_init*.

pthread_sum.c

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

double *a=NULL, sum=0.0;
int    N, size;

void *compute(void *arg) {
    int myStart, myEnd, myN, i;
    long tid = (long)arg;

    // determine start and end of computation for the current thread
    myN = N/size;
    myStart = tid*myN;
    myEnd = myStart + myN;
    if (tid == (size-1)) myEnd = N;

    // compute partial sum
    double mysum = 0.0;
    for (i=myStart; i<myEnd; i++)
      mysum += a[i];

    // grab the lock, update global sum, and release lock
    pthread_mutex_lock(&mutex);
    sum += mysum;
    pthread_mutex_unlock(&mutex);

    return (NULL);
}
```

pthread_sum.c

```
34  int main(int argc, char **argv) {
35      long i;
36      pthread_t *tid;
37
38      if (argc != 3) {
39         printf("Usage: %s <# of elements> <# of threads>\n",argv[0]);
40         exit(-1);
41      }
42
43      N = atoi(argv[1]); // no. of elements
44      size = atoi(argv[2]); // no. of threads
45
46      // allocate vector and initialize
47      tid = (pthread_t *)malloc(sizeof(pthread_t)*size);
48      a = (double *)malloc(sizeof(double)*N);
49      for (i=0; i<N; i++)
50        a[i] = (double)(i + 1);
51
52      // create threads
53      for ( i = 0; i < size; i++)
54        pthread_create(&tid[i], NULL, compute, (void *)i);
55
56      // wait for them to complete
57      for ( i = 0; i < size; i++)
58        pthread_join(tid[i], NULL);
59
60      printf("The total is %g, it should be equal to %g\n",
61             sum, ((double)N*(N+1))/2);
62
63      return 0;
64  }
```

pthread_sum.c

- compile & run

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum.c -o exercise7 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 200 4
The total is 20100, it should be equal to 20100
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise7 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 % _
```

pthread_sum2.c

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

double *a=NULL, *partialsum;
int    N, nthreads;

void *compute(void *arg) {
    int myStart, myEnd, myN, i;
    long tid = (long)arg;

    // determine start and end of computation for the current thread
    myN = N/nthreads;
    myStart = tid*myN;
    myEnd = myStart + myN;
    if (tid == (nthreads-1)) myEnd = N;

    // compute partial sum
    double mysum = 0.0;
    for (i=myStart; i<myEnd; i++)
      mysum += a[i];

    partialsum[tid] = mysum;
    return (NULL);
}

```

pthread_sum2.c

```
27
28  int main(int argc, char **argv) {
29      long i;
30      pthread_t *tid;
31      double sum = 0.0;
32
33      if (argc != 3) {
34         printf("Usage: %s <# of elements> <# of threads>\n",argv[0]);
35         exit(-1);
36      }
37
38      N = atoi(argv[1]); // no. of elements
39      nthreads = atoi(argv[2]); // no. of threads
40
41      // allocate vector and initialize
42      tid = (pthread_t *)malloc(sizeof(pthread_t)*nthreads);
43      a = (double *)malloc(sizeof(double)*N);
44      partialsum = (double *)malloc(sizeof(double)*nthreads);
45      for (i=0; i<N; i++)
46        a[i] = (double)(i + 1);
47
48      // create threads
49      for ( i = 0; i < nthreads; i++)
50        pthread_create(&tid[i], NULL, compute, (void *)i);
51
```

pthread_sum2.c

```
51
52      // wait for them to complete
53      for ( i = 0; i < nthreads; i++)
54        pthread_join(tid[i], NULL);
55
56      for ( i = 0; i < nthreads; i++)
57        sum += partialsum[i];
58
59      printf("The total is %g, it should be equal to %g\n",
60             sum, ((double)N*(N+1))/2);
61
62      free(tid);
63      free(a);
64      free(partialsum);
65
66      return 0;
67  }
68
```

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc pthread_sum2.c -o exercise8 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise8 1000 4
The total is 500500, it should be equal to 500500
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise8 200 4
The total is 20100, it should be equal to 20100
```

Extra exercises

```c
pthread_t tid[2];
int counter;

void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise2.c -o exercise2 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise2

 Job 1 has started

 Job 2 has started

 Job 2 has finished

 Job 2 has finished
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>


pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```c
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

```
(base) mahmutunan@MacBook-Pro lecture32 % gcc exercise3.c -o exercise3 -lpthread
(base) mahmutunan@MacBook-Pro lecture32 % ./exercise3

 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished
```


https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/?ref=rp

Extra Exercise

- https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_mutex.c

Semaphore

- The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.
- Any complex coordination requirement can be satisfied by the appropriate structure of signals.
- For signaling, special variables called semaphores are used.
- To transmit a signal via semaphore s , a process executes the primitive semSignal(s) .
- To receive a signal via semaphore s , a process executes the primitive semWait(s) ; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

Semaphores

- While mutexes are one solution to ensure synchronization, semaphores provide an alternative and more generalized approach to establish synchronization among multiple threads.
- While mutexes provide a locking mechanism (both lock and unlock are performed by a single thread - *cooperative locks*), semaphores provide a signaling mechanism (two different threads cooperate with the wait/signal calls) to synchronize access to shared resources.
- Note that it is possible to implement a mutex using a binary semaphore.

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) A semaphore may be initialized to a nonnegative integer value
2) The semWait operation decrements the semaphore value
3) The semSignal operation increments the semaphore value

- The Pthread library provides the wait and signal calls that are described in the text book. However, since Linux uses the word signal for software interrupts, the wait and signal calls are called *sem_wait* and *sem_post*.
- The C APIs for *sem_wait* and *sem_post* are:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- We have to include the header file *semaphore.h* to compile and link with the pthread library using -*lpthread*.
- The *sem_wait* call decrements the semaphore variable *sem*.
- The *sem_wait* call returns immediately if the value of *sem* is greater than zero after decrementing the semaphore.
- If the current value of the semaphore is zero, then *sem_wait* call will block until the semaphore values goes above zero or it is interrupted by a signal handler.
- Similarly, the *sem_post* call increments the semaphore variable *sem* and if the value of *sem* goes above zero, it will then wake up the corresponding thread or process that was blocked in a *sem_wait* call.

- The *sem_init* function must be used to initialize a semaphore and the *sem_destroy* method to destroy a semaphore.
- The C API for the *sem_init* and *sem_destroy* are given below:

```
#include <semaphore.h>

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

int sem_destroy(sem_t *sem);
```

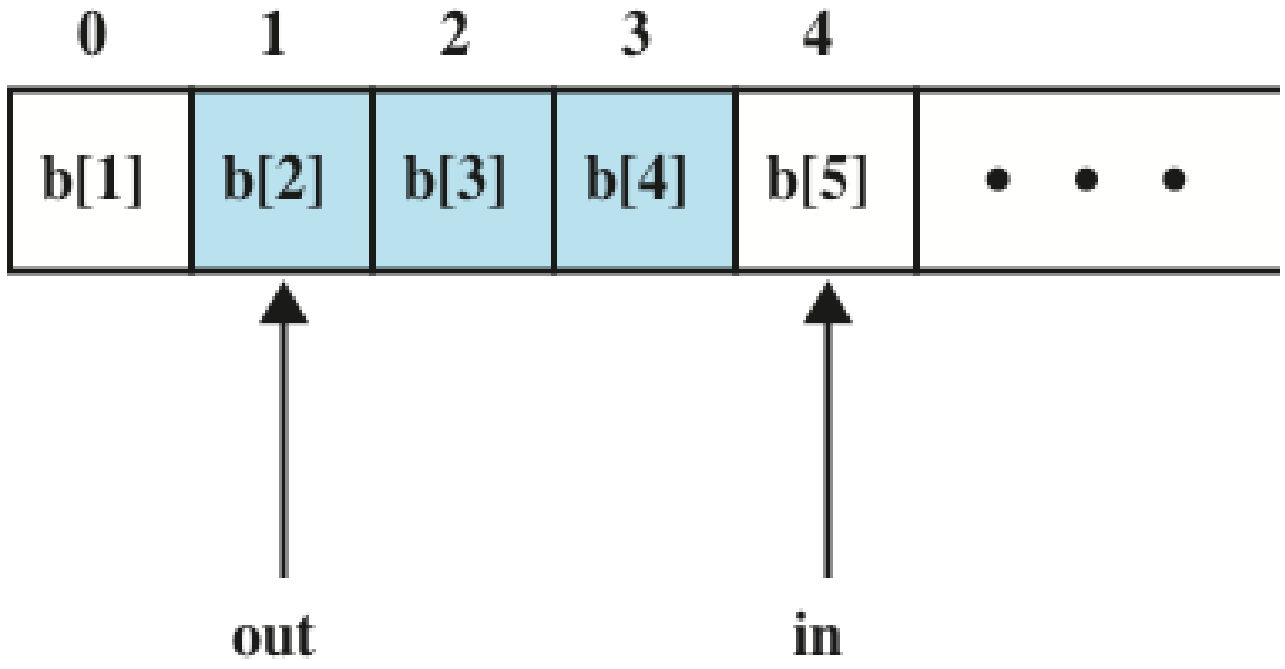
- The *sem_init* call will assign a semaphore *sem* with the initial value provided and the second parameter *pshared* specifies whether the semaphore is shared between threads in with a process or it is shared between processes.

- If the value of *pshared* is zero, then the semaphore is shared between threads and the semaphore must be declared in a shared address space so that all threads can access it.

- If the value of *pshared* is nonzero, then the semaphore will be shared between processes and the semaphore must be declared in a shared memory region

Producer/Consumer Problem

General Statement:	One or more producers are generating data and placing these in a buffer
	A single consumer is taking items out of the buffer one at a time
	Only one producer or consumer may access the buffer at any one time
The Problem:	Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem_wait* and *sem_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

prodcons1.c

```c
/* Solution to the single Producer/Consumer problem using semaphores.
   This example uses a circular buffer to put and get the data
   (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
   To run: ./<filename> <#items>

   To enable printing add -DDEBUG to compile:
   gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

```

```
18
19  #define  NBUFF          10
20
21  int      nitems;                                  /* read-only */
22
23  struct {                    /* data shared by producer and consumer */
24    int       buff[NBUFF];
25    sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
26  } shared;
27
28  void       *producer(void *), *consumer(void *);
29
30  /* end globals */
31
32  /* main program */
33  int main(int argc, char **argv)
34  {
35    pthread_t       tid_producer, tid_consumer;
36
37    if (argc != 2) {
38      printf("Usage: %s <#items>\n", argv[0]);
39      exit(-1);
40    }
41
```

```
41
42      nitems = atoi(argv[1]);
43
44      /* initialize three semaphores */
45      sem_init(&shared.mutex, 0, 1);
46      sem_init(&shared.nempty, 0, NBUFF);
47      sem_init(&shared.nstored, 0, 0);
48
49      /* create one producer thread and one consumer thread */
50      pthread_create(&tid_producer, NULL, producer, NULL);
51      pthread_create(&tid_consumer, NULL, consumer, NULL);
52
53      /* wait for producer and consumer threads */
54      pthread_join(tid_producer, NULL);
55      pthread_join(tid_consumer, NULL);
56
57      /* remove the semaphores */
58      sem_destroy(&shared.mutex);
59      sem_destroy(&shared.nempty);
60      sem_destroy(&shared.nstored);
61
62      return 0;
63  }
64  /* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
  }
  return (NULL);
}
/* end producer */
```

```c
/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
            shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);        /* 1 more empty slot */
  }
  return (NULL);
}
/* end consumer */
```

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**
  $ **./a.out 20**
  wrote 0 to buffer at location 0
  wrote 1 to buffer at location 1
  wrote 2 to buffer at location 2
  wrote 3 to buffer at location 3
  wrote 4 to buffer at location 4
  wrote 5 to buffer at location 5
  wrote 6 to buffer at location 6
  wrote 7 to buffer at location 7
  wrote 8 to buffer at location 8
  read 0 from buffer at location 0
  read 1 from buffer at location 1
  read 2 from buffer at location 2
  read 3 from buffer at location 3
  read 4 from buffer at location 4
  read 5 from buffer at location 5
  read 6 from buffer at location 6
  read 7 from buffer at location 7
  read 8 from buffer at location 8
  wrote 9 to buffer at location 9
  wrote 10 to buffer at location 0
  wrote 11 to buffer at location 1
  wrote 12 to buffer at location 2
  wrote 13 to buffer at location 3
  wrote 14 to buffer at location 4
  wrote 15 to buffer at location 5
  wrote 16 to buffer at location 6
  wrote 17 to buffer at location 7
  wrote 18 to buffer at location 8
  read 9 from buffer at location 9
  read 10 from buffer at location 0
  read 11 from buffer at location 1
  read 12 from buffer at location 2
  read 13 from buffer at location 3
  read 14 from buffer at location 4
  read 15 from buffer at location 5
  read 16 from buffer at location 6
  read 17 from buffer at location 7
  read 18 from buffer at location 8
  wrote 19 to buffer at location 9
  read 19 from buffer at location 9

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```
int      nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
   int       buff[NBUFF];
    sem_t     mutex, nempty, nstored;
      /* semaphores */
} shared;
```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem_init* as per the pseudocode from the textbook.

```
nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);
```

Semaphores

- Generalization of the `semWait` and `semSignal` primitives
  - No other process may access the semaphore until all operations have completed

Consists of:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Semaphores

Solaris provides classic counting semaphores with the following primitives:

- sema_p() Decrements the semaphore, potentially blocking the thread
- sema_v() Increments the semaphore, potentially unblocking a waiting thread
- sema_tryp() Decrements the semaphore if blocking is not required

Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4

– Internally:
  - Implemented as functions within the kernel and are more efficient than user-visable semaphores

– Three types of kernel semaphores:
  – Binary semaphores
  – Counting semaphores
  – Reader-writer semaphores

Traditional Semaphores	
void sema init(struct semaphore *sem, int count)	Initializes the dynamically created semaphore to the given count
void init MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init MUTEX LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down interruptible(struct semaphore *sem)	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received
int down trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
Reader-Writer Semaphores	
void init rwsem(struct rw_semaphore, *rwsem)	Initializes the dynamically created semaphore with a count of 1
void down read(struct rw semaphore, *rwsem)	Down operation for readers
void up read(struct rw semaphore, *rwsem)	Up operation for readers
void down write(struct rw_semaphore, *rwsem)	Down operation for writers
void up write(struct rw semaphore, *rwsem)	Up operation for writers

Table 6.5

Linux Semaphores

(Table can be found on page 293 in textbook)

sem_init

- **#include <semaphore.h>**
- **int sem_init(sem_t **sem*, int *pshared*, unsigned int *value*);**
- Link with *-pthread*.

- **sem_init**() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore. The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

https://www.man7.org/linux/man-pages/man3/sem_init.3.html

Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b - 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem_wait* and *sem_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

prodcons1.c

```
1  /* Solution to the single Producer/Consumer problem using semaphores.
2     This example uses a circular buffer to put and get the data
3     (a bounded-buffer).
4     Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6     To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7     To run: ./<filename> <#items>
8
9     To enable printing add -DDEBUG to compile:
10    gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
11 */
12
13 /* include globals */
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <pthread.h>     /* for POSIX threads */
17 #include <semaphore.h>   /* for POSIX semaphores */
18
```

```
18
19  #define  NBUFF          10
20
21  int      nitems;                                  /* read-only */
22
23  struct {                  /* data shared by producer and consumer */
24    int       buff[NBUFF];
25    sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
26  } shared;
27
28  void       *producer(void *), *consumer(void *);
29
30  /* end globals */
31
32  /* main program */
33  int main(int argc, char **argv)
34  {
35    pthread_t       tid_producer, tid_consumer;
36
37    if (argc != 2) {
38      printf("Usage: %s <#items>\n", argv[0]);
39      exit(-1);
40    }
41
```

```
41
42      nitems = atoi(argv[1]);
43
44      /* initialize three semaphores */
45      sem_init(&shared.mutex, 0, 1);
46      sem_init(&shared.nempty, 0, NBUFF);
47      sem_init(&shared.nstored, 0, 0);
48
49      /* create one producer thread and one consumer thread */
50      pthread_create(&tid_producer, NULL, producer, NULL);
51      pthread_create(&tid_consumer, NULL, consumer, NULL);
52
53      /* wait for producer and consumer threads */
54      pthread_join(tid_producer, NULL);
55      pthread_join(tid_consumer, NULL);
56
57      /* remove the semaphores */
58      sem_destroy(&shared.mutex);
59      sem_destroy(&shared.nempty);
60      sem_destroy(&shared.nstored);
61
62      return 0;
63  }
64  /* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
  }
  return (NULL);
}
/* end producer */
```

```
87  /* consumer function */
88  void *consumer(void *arg)
89  {
90    int i;
91
92    for (i = 0; i < nitems; i++) {
93      sem_wait(&shared.nstored);        /* wait for at least 1 stored item */
94      sem_wait(&shared.mutex);
95
96      if (shared.buff[i % NBUFF] != i)
97        printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
98  #ifdef DEBUG
99      printf("read %d from buffer at location %d\n",
100             shared.buff[i % NBUFF], i % NBUFF);
101 #endif
102
103     sem_post(&shared.mutex);
104     sem_post(&shared.nempty);         /* 1 more empty slot */
105   }
106   return (NULL);
107 }
108 /* end consumer */
```

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```
int      nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
   int       buff[NBUFF];
    sem_t     mutex, nempty, nstored;
      /* semaphores */
} shared;
```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem_init* as per the pseudocode from the textbook.

```
nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);
```

- We create two separate threads, one for the producer and one for the consumer, and wait for the two threads to complete.

```
/* create one producer thread and one
consumer thread */
pthread_create(&tid_producer, NULL,
producer, NULL);
pthread_create(&tid_consumer, NULL,
consumer, NULL);

/* wait for producer and consumer threads */
pthread_join(tid_producer, NULL);
pthread_join(tid_consumer, NULL);
```

- Now let us look at the producer thread.
- It executes a loop equal to the number of items specified (*nitems*) and during each iteration of the loop, waits on the semaphore *nempty*.
- Initially *nempty* is set to *NBUFF*, so *sem_wait* returns immediately and waits on the semaphore *mutex*. Since *mutex* is initially set to 1, the producer thread enters the critical section and assigns the value *i* to the buffer location *i % NBUFF* and then release the *mutex* semaphore (calls *sem_post* on the semaphore *mutex*).
- Now that there is at least one element in the buffer, it also posts *sem_post* on the semaphore *nstored* to indicate to the consumer that there is an element in the buffer and continues with the loop.
- The producer thread terminates when the loop completes (i.e., after *nitems* iterations).
-

```
/* producer function */
void *producer(void *arg) {
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least
1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into
circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i %
NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored
item */
  }

  return (NULL);
}
/* end producer */
```

- Meanwhile, the consumer thread will enter the loop and wait on the semaphore *nstored*.
- Since initially *nstored* is set to 0, this call will block and the consumer will wait until the producer posts on the semaphore *nstored*.
- When the producer posts on the semaphore *nstored*, the consumer will return from *sem_wait* on *nstored* and invoke the *sem_wait* on the semaphore *mutex*.
- If the producer is not in the critical section, the consumer will obtain the *mutex* semaphore, consume the buffer (we simply check if the value in the buffer match the corresponding *(loop index mod NBUFF)* and print an error message in case they don't match), and release the mutex by calling *sem_post* on the *mutex* semaphore.
- Then the consumer thread will post the *sem_post* on the semaphore *nempty* to indicate to the producer that now there is an empty slot. The consumer thread terminates when the loop completes (i.e., after *nitems* iterations).

```
/* consumer function */
void *consumer(void *arg) {
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);        /* wait for at
least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i
% NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
            shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);         /* 1 more empty
slot */
  }

  return (NULL);
}
/* end consumer *
```

- You can compile the program with the DEBUG variable defined using -DDEBUG during compilation and see how the two threads progress. Here is a sample output when we execute the program with 20 items. You will notice that the output would be different every time you execute the program even with the same number of elements.

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**
  $ **./a.out 20**
  wrote 0 to buffer at location 0
  wrote 1 to buffer at location 1
  wrote 2 to buffer at location 2
  wrote 3 to buffer at location 3
  wrote 4 to buffer at location 4
  wrote 5 to buffer at location 5
  wrote 6 to buffer at location 6
  wrote 7 to buffer at location 7
  wrote 8 to buffer at location 8
  read 0 from buffer at location 0
  read 1 from buffer at location 1
  read 2 from buffer at location 2
  read 3 from buffer at location 3
  read 4 from buffer at location 4
  read 5 from buffer at location 5
  read 6 from buffer at location 6
  read 7 from buffer at location 7
  read 8 from buffer at location 8
  wrote 9 to buffer at location 9
  wrote 10 to buffer at location 0
  wrote 11 to buffer at location 1
  wrote 12 to buffer at location 2
  wrote 13 to buffer at location 3
  wrote 14 to buffer at location 4
  wrote 15 to buffer at location 5
  wrote 16 to buffer at location 6
  wrote 17 to buffer at location 7
  wrote 18 to buffer at location 8
  read 9 from buffer at location 9
  read 10 from buffer at location 0
  read 11 from buffer at location 1
  read 12 from buffer at location 2
  read 13 from buffer at location 3
  read 14 from buffer at location 4
  read 15 from buffer at location 5
  read 16 from buffer at location 6
  read 17 from buffer at location 7
  read 18 from buffer at location 8
  wrote 19 to buffer at location 9
  read 19 from buffer at location 9

prodcons2.c

- Solution to multiple producer and single consumer problem:

```c
/* Solution to the Multiple Producer/Single Consumer problem using
   semaphores. This example uses a circular buffer to put and get the
   data (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

#define  min(a, b)  (((a) < (b)) ? (a) : (b))

#define  NBUFF           10
#define  MAXNTHREADS     100
```

```
19
20  int      nitems, nproducers;                    /* read-only */
21
22  struct {                  /* data shared by producers and consumers */
23    int       buff[NBUFF];
24    int       nput;                    /* item number: 0, 1, 2, ... */
25    int       nputval;                 /* value to store in buff[] */
26    sem_t     mutex, nempty, nstored;  /* semaphores, not pointers */
27  } shared;
28
29  void       *producer(void *), *consumer(void *);
30
31  /* end globals */
32
33  /* main program */
34  int main(int argc, char **argv)
35  {
36    int             i, prodcount[MAXNTHREADS];
37    pthread_t       tid_producer[MAXNTHREADS], tid_consumer;
38
39    if (argc != 3) {
40      printf("Usage: %s <#items> <#producers>\n", argv[0]);
41      exit(-1);
42    }
43
```

```
44    nitems = atoi(argv[1]);
45    nproducers = min(atoi(argv[2]), MAXNTHREADS);
46
47    /* initialize three semaphores */
48    sem_init(&shared.mutex, 0, 1);
49    sem_init(&shared.nempty, 0, NBUFF);
50    sem_init(&shared.nstored, 0, 0);
51
52    /* create all producers and one consumer */
53    for (i = 0; i < nproducers; i++) {
54      prodcount[i] = 0;
55      pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
56    }
57    pthread_create(&tid_consumer, NULL, consumer, NULL);
58
59    /* wait for all producers and the consumer */
60    for (i = 0; i < nproducers; i++) {
61      pthread_join(tid_producer[i], NULL);
62      printf("producer count[%d] = %d\n", i, prodcount[i]);
63    }
64    pthread_join(tid_consumer, NULL);
65
66    sem_destroy(&shared.mutex);
67    sem_destroy(&shared.nempty);
68    sem_destroy(&shared.nstored);
69
70    return 0;
71  }
72  /* end main */
```

```
74  /* producer function */
75  void *producer(void *arg)
76  {
77    for ( ; ; ) {
78      sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
79      sem_wait(&shared.mutex);
80
81      if (shared.nput >= nitems) {
82        sem_post(&shared.nempty);
83        sem_post(&shared.mutex);
84        return(NULL);                 /* all done */
85      }
86
87      shared.buff[shared.nput % NBUFF] = shared.nputval;
88  #ifdef DEBUG
89      printf("wrote %d to buffer at location %d\n",
90             shared.nputval, shared.nput % NBUFF);
91  #endif
92      shared.nput++;
93      shared.nputval++;
94
95      sem_post(&shared.mutex);
96      sem_post(&shared.nstored);      /* 1 more stored item */
97      *((int *) arg) += 1;
98    }
99  }
100 /* end producer */
```

```
101
102 /* consumer function */
103 void *consumer(void *arg)
104 {
105   int i;
106
107   for (i = 0; i < nitems; i++) {
108     sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
109     sem_wait(&shared.mutex);
110
111     if (shared.buff[i % NBUFF] != i)
112       printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
113 #ifdef DEBUG
114     printf("read %d from buffer at location %d\n",
115            shared.buff[i % NBUFF], i % NBUFF);
116 #endif
117
118     sem_post(&shared.mutex);
119     sem_post(&shared.nempty);        /* 1 more empty slot */
120   }
121
122   return (NULL);
123 }
124 /* end consumer */
```

prodcons3.c

- Solution to multiple producer and multiple consumer problem:

```
1   /* Solution to the Multiple Producer/Multiple Consumer problem using
2      semaphores. This example uses a circular buffer to put and get the
3      data (a bounded-buffer).
4      Source: UNIX Network Programming, Volume 2 by W. Richard Stevens
5
6      To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
7   */
8
9   /* include globals */
10  #include <stdio.h>
11  #include <stdlib.h>
12  #include <pthread.h>     /* for POSIX threads */
13  #include <semaphore.h>   /* for POSIX semaphores */
14
15  #define  min(a, b)  (((a) < (b)) ? (a) : (b))
```

```
16
17  #define  NBUFF           10
18  #define  MAXNTHREADS     100
19
20  int      nitems, nproducers, nconsumers;              /* read-only */
21
22  struct {                 /* data shared by producers and consumers */
23    int       buff[NBUFF];
24    int       nput;                     /* item number: 0, 1, 2, ... */
25    int       nputval;                  /* value to store in buff[] */
26    int       nget;                     /* item number: 0, 1, 2, ... */
27    int       ngetval;                  /* value fetched from buff[] */
28    sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
29  } shared;
30
31  void       *producer(void *), *consumer(void *);
32
33  /* end globals */
34
35  /* main program */
36  int main(int argc, char **argv)
37  {
38    int             i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
39    pthread_t       tid_producer[MAXNTHREADS], tid_consumer[MAXNTHREADS];
40
41    if (argc != 4) {
42      printf("Usage: %s <#items> <#producers> <#consumers>\n", argv[0]);
43      exit(-1);
44    }
```

```
45
46      nitems = atoi(argv[1]);
47      nproducers = min(atoi(argv[2]), MAXNTHREADS);
48      nconsumers = min(atoi(argv[3]), MAXNTHREADS);
49
50      /* initialize three semaphores */
51      sem_init(&shared.mutex, 0, 1);
52      sem_init(&shared.nempty, 0, NBUFF);
53      sem_init(&shared.nstored, 0, 0);
54
55      /* create all producers and all consumers */
56      for (i = 0; i < nproducers; i++) {
57        prodcount[i] = 0;
58        pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
59      }
60      for (i = 0; i < nconsumers; i++) {
61        conscount[i] = 0;
62        pthread_create(&tid_consumer[i], NULL, consumer, &conscount[i]);
63      }
64
65      /* wait for all producers and all consumers */
66      for (i = 0; i < nproducers; i++) {
67        pthread_join(tid_producer[i], NULL);
68        printf("producer count[%d] = %d\n", i, prodcount[i]);
69      }
70      for (i = 0; i < nconsumers; i++) {
71        pthread_join(tid_consumer[i], NULL);
72        printf("consumer count[%d] = %d\n", i, conscount[i]);
73      }
```

```
74
75      sem_destroy(&shared.mutex);
76      sem_destroy(&shared.nempty);
77      sem_destroy(&shared.nstored);
78
79      return 0;
80    }
81    /* end main */
82
83    /* producer function */
84    void *producer(void *arg)
85    {
86      for ( ; ; ) {
87        sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
88        sem_wait(&shared.mutex);
89
90        if (shared.nput >= nitems) {
91          sem_post(&shared.nstored);    /* let consumers terminate */
92          sem_post(&shared.nempty);
93          sem_post(&shared.mutex);
94          return(NULL);                 /* all done */
95        }
96
97        shared.buff[shared.nput % NBUFF] = shared.nputval;
98        shared.nput++;
99        shared.nputval++;
100
101       sem_post(&shared.mutex);
102       sem_post(&shared.nstored);      /* 1 more stored item */
103       *((int *) arg) += 1;
104     }
105   }
106   /* end producer */
```

```
107
108 /* consumer function */
109 void *consumer(void *arg)
110 {
111   int i;
112
113   for ( ; ; ) {
114     sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
115     sem_wait(&shared.mutex);
116
117     if (shared.nget >= nitems) {
118       sem_post(&shared.nstored);
119       sem_post(&shared.mutex);
120       return(NULL);                  /* all done */
121     }
122
123     i = shared.nget % NBUFF;
124     if (shared.buff[i] != shared.ngetval)
125       printf("error: buff[%d] = %d\n", i, shared.buff[i]);
126     shared.nget++;
127     shared.ngetval++;
128
129     sem_post(&shared.mutex);
130     sem_post(&shared.nempty);        /* 1 more empty slot */
131     *((int *) arg) += 1;
132   }
133 }
134 /* end consumer */
135
```

Semaphore for Resource Allocation

- Pool of N problems
- Resource sharing among multiple processes / uninterrupted period
- Limit the highest number of resources in use at any time
- ……

What we have learned so far…

- ls
- cd
- rm
- ps
- cat
- touch
- …….
- ……..

Copying a file or directory

- `cp [options] <source> <destination>`

```
unan@unan-VirtualBox:~/lecture36$ ls
dir1  dir2  somefile.txt
unan@unan-VirtualBox:~/lecture36$ cp somefile.txt dir1
unan@unan-VirtualBox:~/lecture36$ ls
dir1  dir2  somefile.txt
unan@unan-VirtualBox:~/lecture36$ cd dir1
unan@unan-VirtualBox:~/lecture36/dir1$ ls
file1.txt  somefile.txt
unan@unan-VirtualBox:~/lecture36/dir1$
```

Moving a File or Directory

- `mv [options] <source> <destination>`

```
unan@unan-VirtualBox:~/lecture36$ mkdir dir3
unan@unan-VirtualBox:~/lecture36$ mv dir1 dir3
unan@unan-VirtualBox:~/lecture36$ ls
dir2  dir3  somefile.txt
unan@unan-VirtualBox:~/lecture36$
```

```
unan@unan-VirtualBox:~/lecture36$ mv ./dir2 ./dir2new
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile.txt
unan@unan-VirtualBox:~/lecture36$
```

nl - line numbers

- `nl [-options] [path]`

```
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
unan@unan-VirtualBox:~/lecture36$ nl somefile.txt
     1  Mahmut Unan 55
     2  John Smith 45
     3  John doe 67
     4  Julie Doe 99
     5  Zack Zetta 88
```

cut

- `cut [-options] [path]`

```
unan@unan-VirtualBox:~/lecture36$ cut -f 1 -d ' ' somefile.txt
Mahmut
John
John
Julie
Zack
unan@unan-VirtualBox:~/lecture36$
```

other useful data ops

- sed
  - sed stands for **Stream Editor** and it effectively allows us to do a search and replace on our data. It is quite a powerful command but we will use it here in it's basic format.
  - **sed <expression> [path]**
- uniq
  - uniq stands for **unique** and it's job is to remove duplicate lines from the data. One limitation however is that those lines must be adjacent (ie, one after the other
  - **uniq [options] [path]**

https://ryanstutorials.net/linuxtutorial/filters.php

diff

```
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
unan@unan-VirtualBox:~/lecture36$ cat somefile2.txt
Mahmut Unan 55
John Smith 45
John doe 67
Julie Doe 99
Zack Zetta 88
some extra line
unan@unan-VirtualBox:~/lecture36$ diff somefile.txt somefile2.txt
5a6
> some extra line
unan@unan-VirtualBox:~/lecture36$
```

grep / egrep

- `grep [options] <pattern> [files]`
- `egrep [options] <pattern> [files]`

```
unan@unan-VirtualBox:~/lecture36$ grep 'o' somefile.txt
John Smith 45
John doe 67
Julie Doe 99
unan@unan-VirtualBox:~/lecture36$ grep -c 'o' somefile.txt
3
```

https://ryanstutorials.net/linuxtutorial/cheatsheetgrep.php

Wildcards

- ***** - zero or more characters
- **?** - a single character
- **[ ]** - a range of characters

Networking / Communication

- SSH
  - SSH username@ip-address or hostname

- Ping
  - ping hostname
  - ping ipaddress

```
unan@unan-VirtualBox:~/lecture36$ ping www.google.com
PING www.google.com (64.233.185.99) 56(84) bytes of data.
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=1 ttl=63 time=23.2 m
s
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=2 ttl=63 time=24.8 m
s
64 bytes from yb-in-f99.1e100.net (64.233.185.99): icmp_seq=3 ttl=63 time=22.6 m
```

- ftp hostname

```
unan@unan-VirtualBox:~/lecture36$ ftp ftp.javatutorialhub.com
Connected to javatutorialhub.com.
220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
220-You are user number 1 of 50 allowed.
220-Local time is now 15:39. Server port: 21.
220-This is a private system - No anonymous login
220-IPv6 connections are also welcome on this server.
220 You will be disconnected after 15 minutes of inactivity.
Name (ftp.javatutorialhub.com:unan):
```

- telnet
  - telnet hostname

Linux Administrator

- Add new user
  - sudo adduser mynewuser
- Disable an account
  - sudo passwd -l mynewuser
- Delete an account
  - sudo userdel -r mynewuser
- Add user to usergroup
  - sudo usermod -a -G home mynewuser

finger

- This command is used to **procure information of the users on a Linux machine**.
- You can use it on both local & remote machines
- The syntax 'finger' gives data on all the logged users on the remote and local machine.
- The syntax 'finger username' specifies the information of the user.

https://www.guru99.com/linux-admin.html

date /cal / time

```
unan@unan-VirtualBox:~/lecture36$ date
Fri 20 Nov 2020 01:54:12 PM CST
unan@unan-VirtualBox:~/lecture36$ date  +'%a %d-%m-%y'
Fri 20-11-20
unan@unan-VirtualBox:~/lecture36$ time ls
dir2new  dir3  somefile2.txt  somefile.txt

real    0m0.002s
user    0m0.001s
sys     0m0.000s
unan@unan-VirtualBox:~/lecture36$ cal
    November 2020
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

unan@unan-VirtualBox:~/lecture36$
```

wget / curl

- wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.17.2.tar.xz

- curl https://www.example.com/

https://curl.se/docs/manual.html

https://linuxize.com/post/wget-command-examples/

gzip / tar/ bzip2

- gzip filename
- gzip -k filename
- gunzip

- tar [options] [archive-file] [file or directory to be archived]

- bzip2 somefilename
- bunzip2 somefilename

su / sudo

- su **<username>**
- su root
- sudo apt install curl

mount

- mount -t type **<device> <directory>**

```
unan@unan-VirtualBox:~/lecture36$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,noexec,relatime,size=1987276k,nr_in
6819,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,
e=000)
tmpfs on /run type tmpfs (rw,nosuid,nodev,noexec,relatime,size=403088k,mo
/dev/sda5 on / type ext4 (rw,relatime,errors=remount-ro)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexe
ime)
```

shred


unan@unan-VirtualBox:~/lecture36$ ls
dir2new dir3 somefile2.txt somefile.txt
unan@unan-VirtualBox:~/lecture36$ shred somefile.txt
unan@unan-VirtualBox:~/lecture36$ ls
dir2new dir3 somefile2.txt somefile.txt
unan@unan-VirtualBox:~/lecture36$ cat somefile.txt

verbose info / delete

- shred -u -v **<fileName>**

```
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile2.txt  somefile.txt
unan@unan-VirtualBox:~/lecture36$ shred -u -v somefile.txt
shred: somefile.txt: pass 1/3 (random)...
shred: somefile.txt: pass 2/3 (random)...
shred: somefile.txt: pass 3/3 (random)...
shred: somefile.txt: removing
shred: somefile.txt: renamed to 000000000000
shred: 000000000000: renamed to 00000000000
shred: 00000000000: renamed to 0000000000
shred: 0000000000: renamed to 000000000
shred: 000000000: renamed to 00000000
shred: 00000000: renamed to 0000000
shred: 0000000: renamed to 000000
shred: 000000: renamed to 00000
shred: 00000: renamed to 0000
shred: 0000: renamed to 000
shred: 000: renamed to 00
shred: 00: renamed to 0
shred: somefile.txt: removed
unan@unan-VirtualBox:~/lecture36$ ls
dir2new  dir3  somefile2.txt
unan@unan-VirtualBox:~/lecture36$
```

last

```
unan@unan-VirtualBox:~/lecture36$ last
unan     :0           :0                Thu Nov 19 22:47    gone - no logout
reboot   system boot  5.4.0-42-generic Thu Nov 19 22:47   still running
unan     :0           :0                Thu Nov 19 20:13 - crash  (02:34)
reboot   system boot  5.4.0-42-generic Thu Nov 19 20:12   still running
unan     :0           :0                Mon Oct 12 00:19 - crash (38+20:53)
reboot   system boot  5.4.0-42-generic Mon Oct 12 00:18   still running
unan     :0           :0                Mon Sep 14 16:52 - crash (27+07:26)
reboot   system boot  5.4.0-42-generic Mon Sep 14 16:51   still running
unan     :0           :0                Tue Jul 28 11:49 - crash (48+05:02)
reboot   system boot  5.4.0-42-generic Tue Jul 28 11:49   still running

wtmp begins Tue Jul 28 11:49:20 2020
```

ifconfig

- The command ifconfig stands for interface configurator. This command enables us to initialize an interface, assign IP address, enable or disable an interface. It display route and network interface.
- You can view IP address, MAC address and MTU (Maximum Transmission Unit) with ifconfig command.
- A newer version of ifconfig is ip command. ifconfig command works for all the versions.

Syntax:

ifconfig

https://www.javatpoint.com/linux-ifconfig

ip

```
unan@unan-VirtualBox:~/lecture36$ ip
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | address | addrlabel | route | rule | neigh | ntable |
                   tunnel | tuntap | maddress | mroute | mrule | monitor | xfrm
|
                   netns | l2tp | fou | macsec | tcp_metrics | token | netconf |
 ila |
                   vrf | sr | nexthop }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -h[uman-readable] | -iec | -j[son] | -p[retty] |
                    -f[amily] { inet | inet6 | mpls | bridge | link } |
                    -4 | -6 | -I | -D | -M | -B | -0 |
                    -l[oops] { maximum-addr-flush-attempts } | -br[ief] |
                    -o[neline] | -t[imestamp] | -ts[hort] | -b[atch] [filename]
|
                    -rc[vbuf] [size] | -n[etns] name | -N[umeric] | -a[ll] |
                    -c[olor]}
```

Linux Certifications

- Most of the hiring manager are looking to recruit Linux professionals.
- The emergence of open cloud platforms is creating increasing demand for Linux professionals who have the right expertise
- Linux-certified professionals always be a better position in the job market
- Employers are looking for more Linux talent.
- Better salary increments for Linux certified professionals
- Some famous certificates
  - Red Hat Linux SuSE Linux
  - Linux Professional Institute (LPIC)
  - CompTIA
  - Linux Foundation
  - Oracle

https://www.guru99.com/best-linux-certifications.html

Useful resources

- https://www.guru99.com/best-linux-books-beginners.html
- https://ubuntu.com/tutorials?page=2
- https://ubuntu.com/tutorials/create-your-first-snap#1-overview