ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems for Automation M

7. Processes and Threads

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

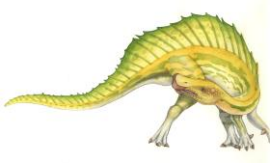
Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-Process Communication (IPC)
- Examples of IPC Systems
- Communication in Client-Server Systems

Chapter 4: Threads

- Motivation and Benefits
- Multithreading Models
- Some Issues related to Threading
- Thread Libraries (POSIX)
- Threads in Linux (NPTL)



Objectives

- To introduce the notion of a process—a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication using shared memory and message passing
- To describe communication in client-server systems
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To examine issues related to multithreaded programming
- To discuss the APIs for the Pthreads thread library
- To present how Linux supports threads

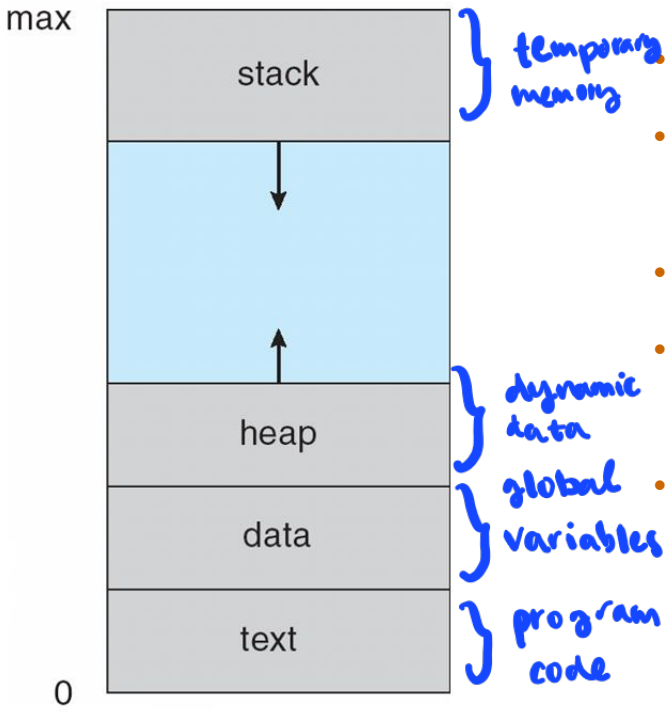
Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **tasks** (job is an instance of a task)
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion

Process in Memory 1/2

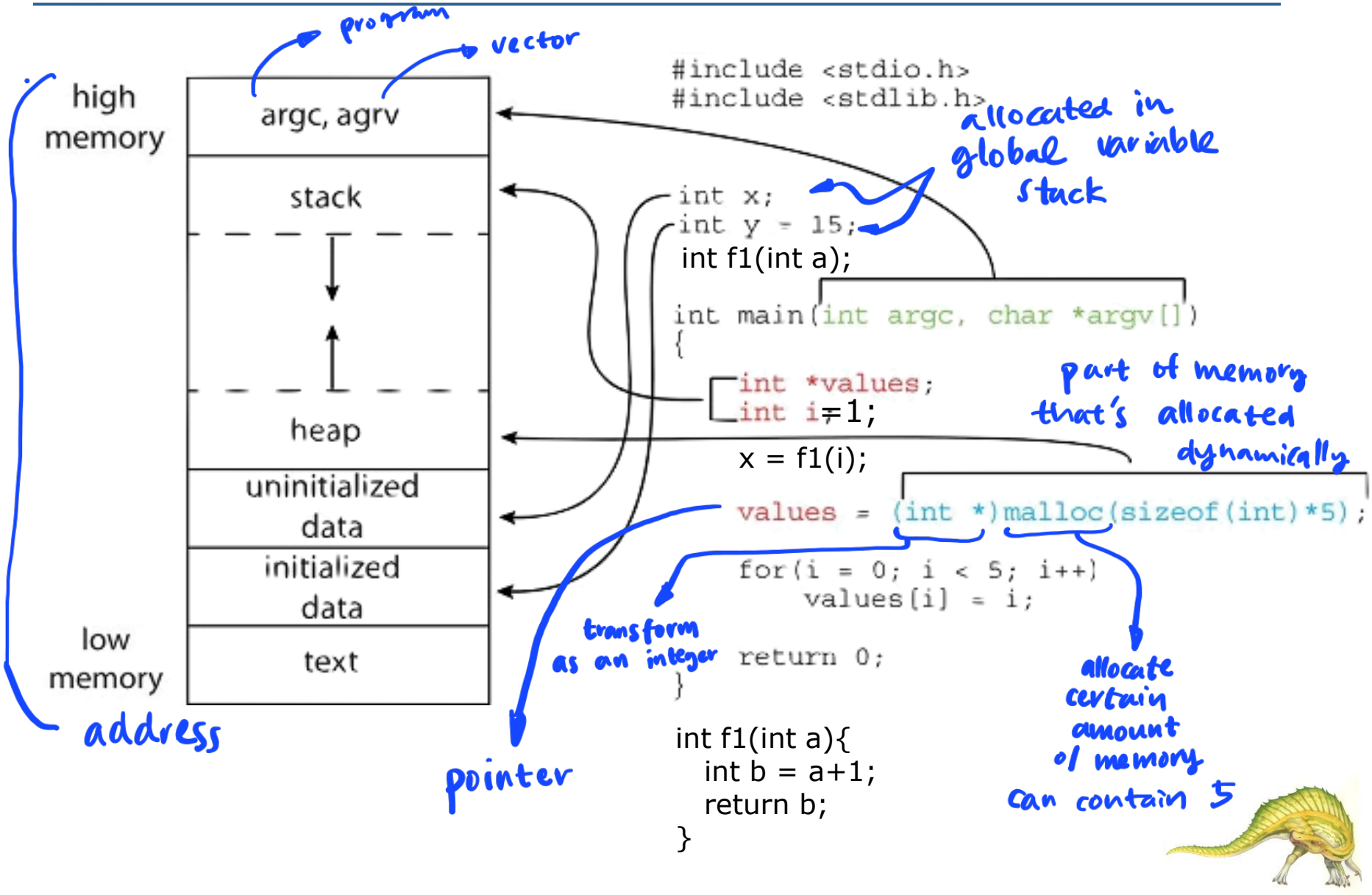
needs certain elements in memory

- Process is made of multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
    - when invoking a function, a **stack frame** is created

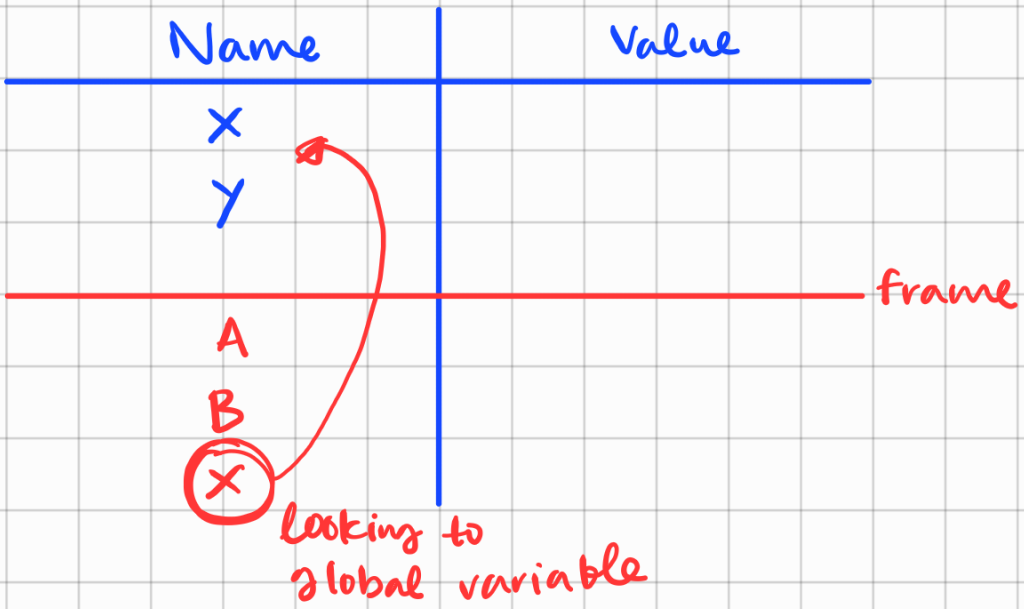


local/global are variables scopes ➔visibility inside the program

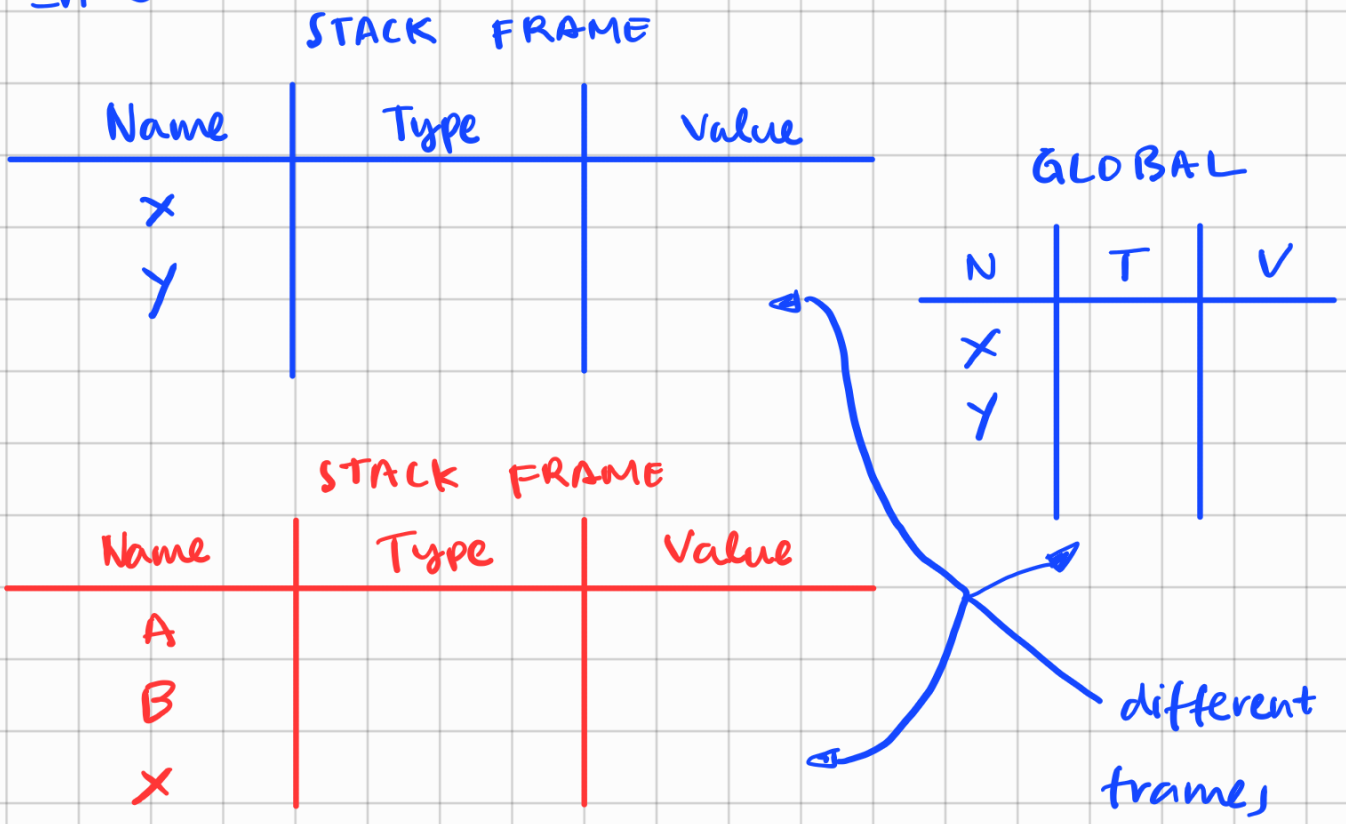
Process in Memory 2/2

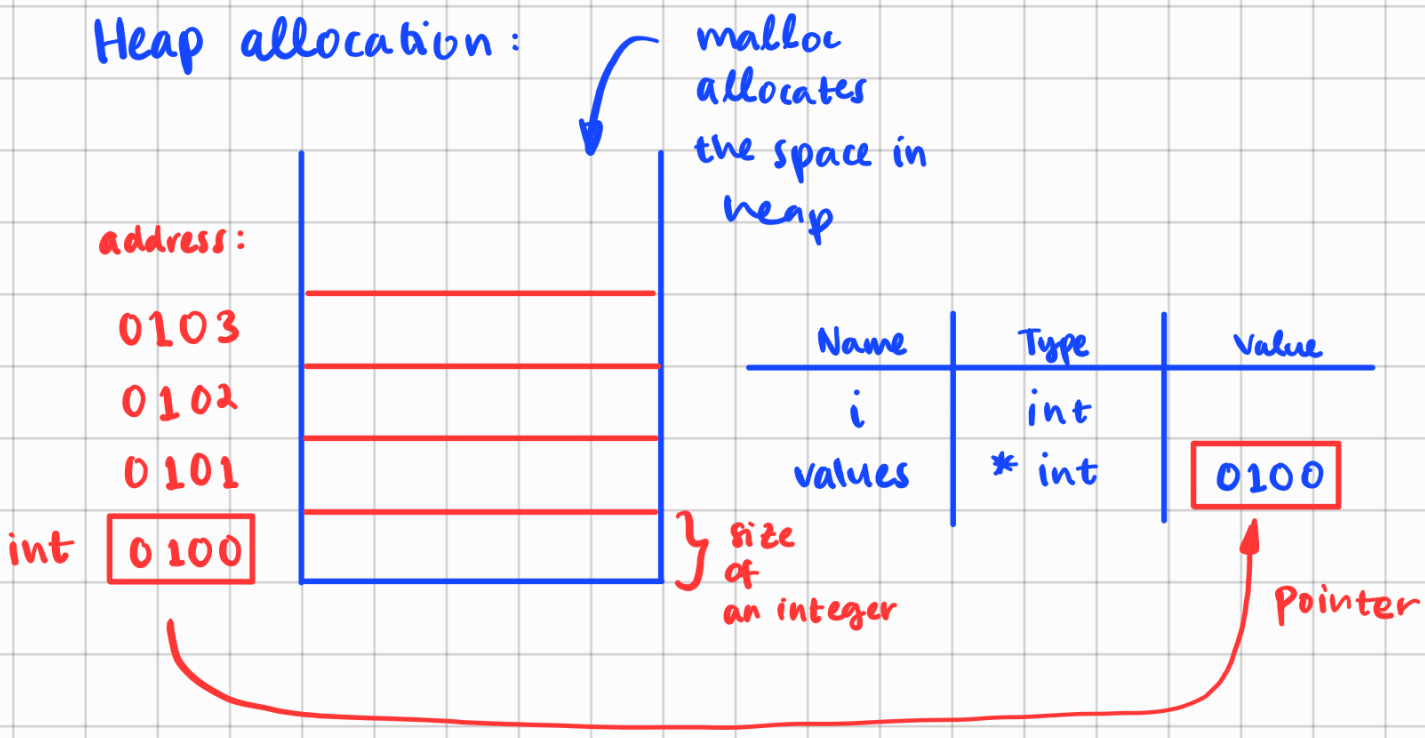


Compared to Python...



In C...

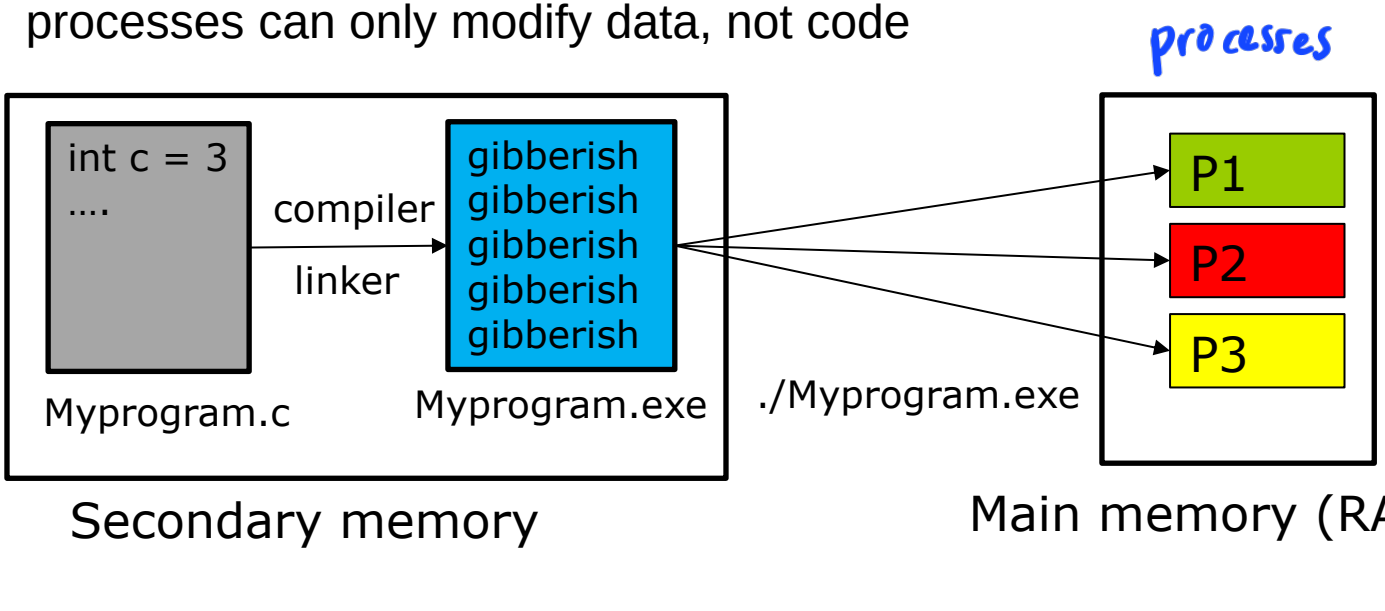


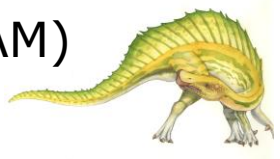

Heap allocation:
malloc allocates the space in heap
address:
0103
0102
0101
int
0100
size of an integer
Name
Type
Value
i
int
values
* int
0100
Pointer

Process Concept

- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- Program (***passive*** entity stored on disk as **executable file**) becomes process (***active*** entity) when the executable file is loaded into memory

- One program can be executed by several processes (e.g., multiple users executing the same program)
- processes can only modify data, not code





Process Execution State

life of the process

- As a process executes, it changes **state**
  - **new**: The process is being created (selected with long-term scheduling, has just PID)
  - **ready**: The process is waiting to be assigned to a processor (whole process structure in memory)
  - **running**: Instructions are being executed (selected with CPU scheduling and dispatched)
    - If multitasking (time-sharing) can go back to ready
  - **terminated**: The process has finished execution
  - **waiting**: for some event to occur (waiting for a child, I/O operation, shared memory, timer. Crucial for multiprogramming. A queue for each type of event)

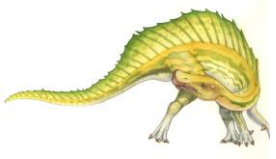
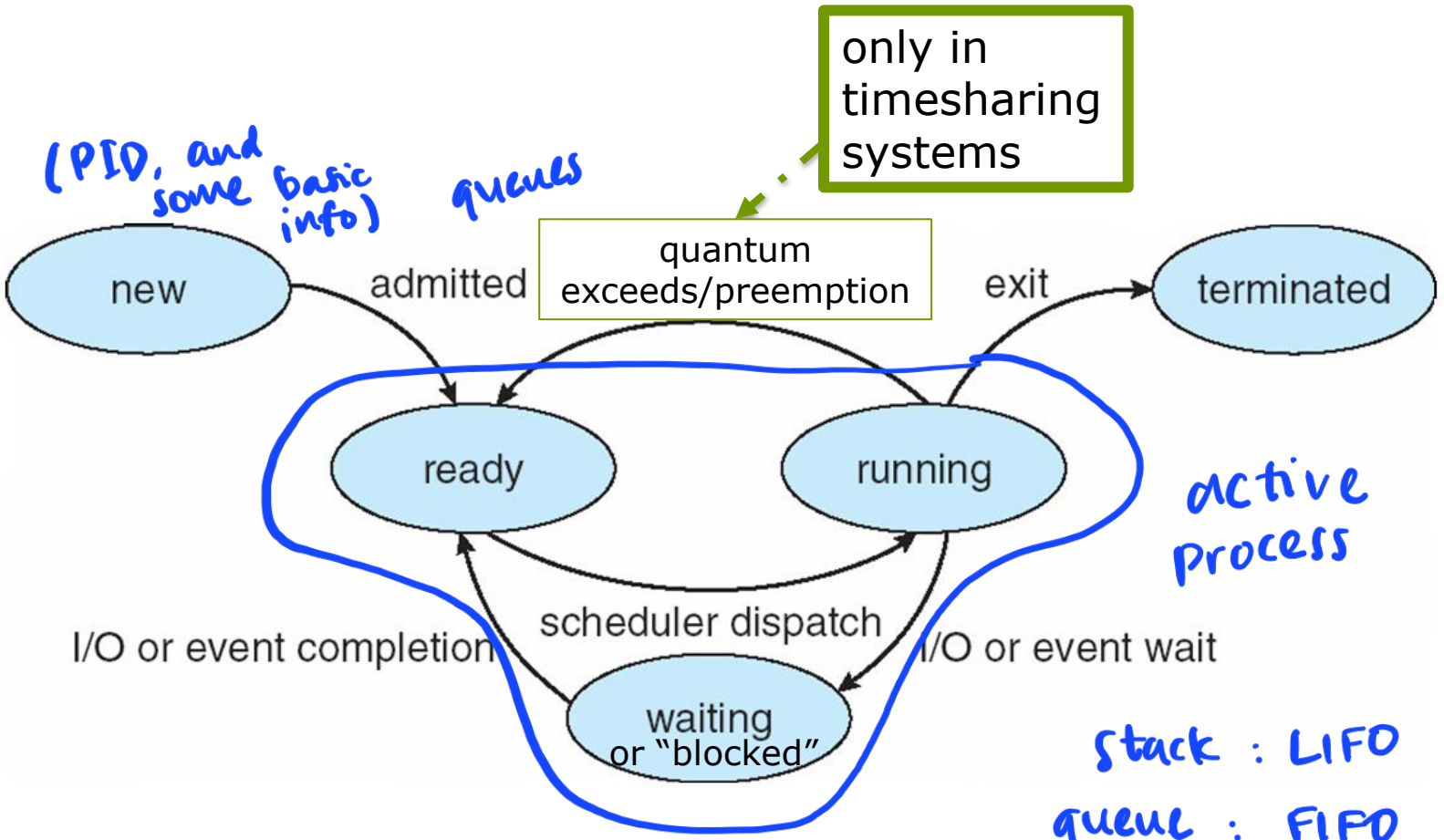


Diagram of Process Execution State



active when ready, running or waiting

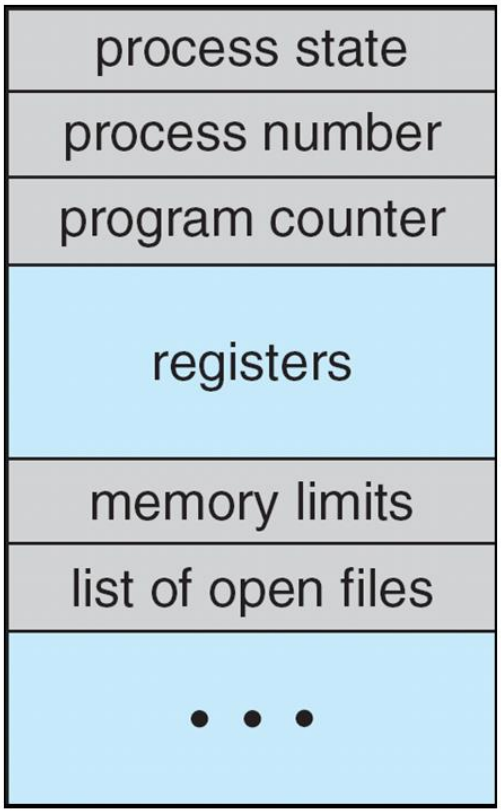
Process Control Block (PCB)

remains in memory

identity of the process

Each process is represented in the OS by a PCB (also called **task control block**)

- Process **state** (running, waiting, etc.)
- Copy of values of CPU **registers**
  - including **program counter**: location of instruction to next execute (not updated in memory during computation, just for context switches)
- CPU **scheduling** information
  - priorities, scheduling queue pointers
- **Memory**-management information (memory allocated to the process)
- **Accounting** information (CPU used, clock time elapsed since start, time limits, etc.)
- **I/O status** information (list of I/O devices allocated to process, and open files)



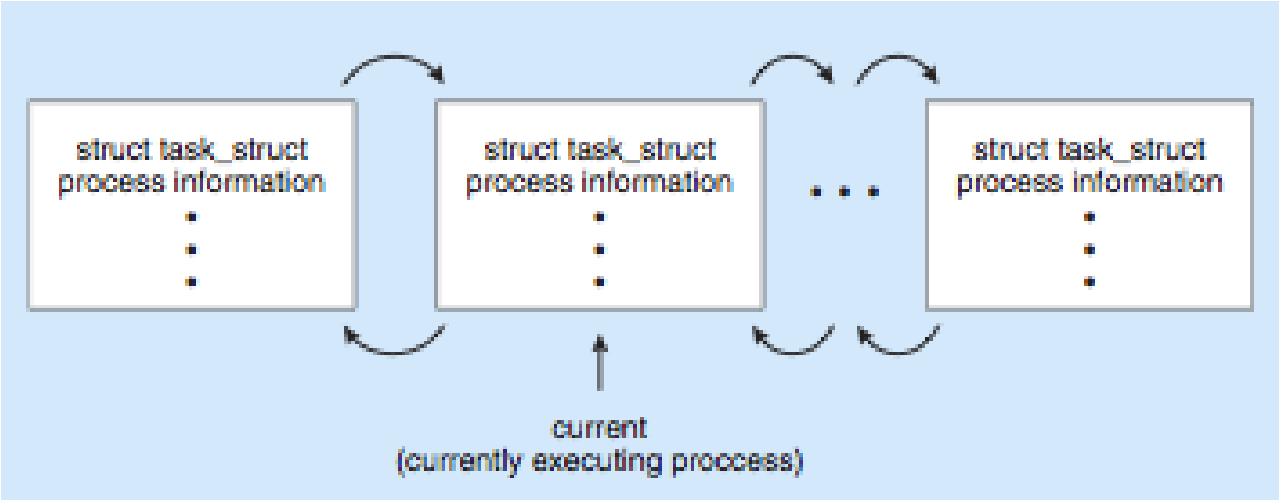
PID

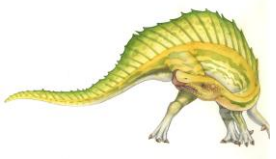
Process Representation in Linux

- Represented by the C structure `task_struct` (see linux/sched.h)

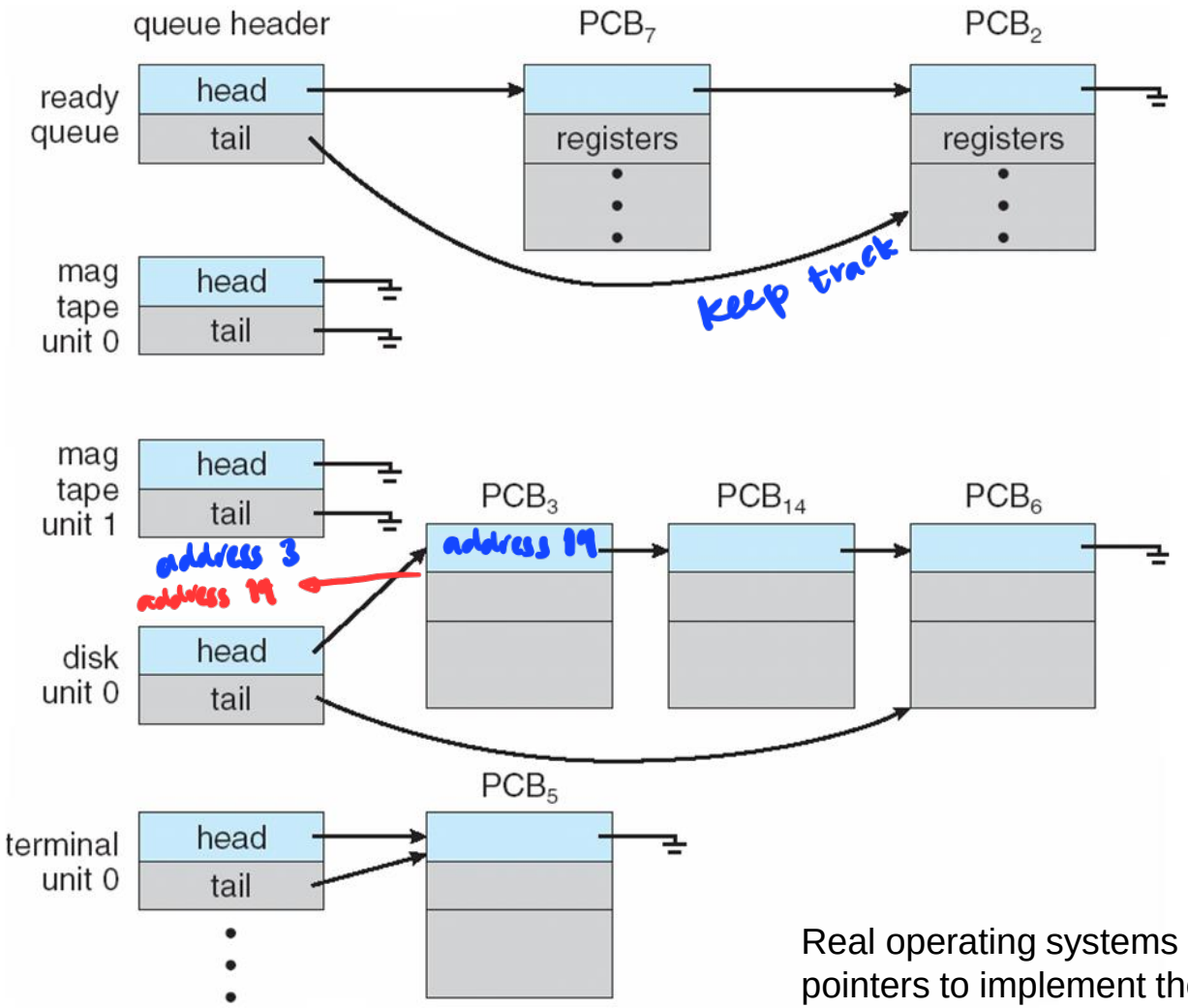
```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
...
```

At least one (often two) list pointer (struct task_struct*) is embedded in the PCB.
Linux kernel maintains active processes as a doubly linked list of `task_struct`



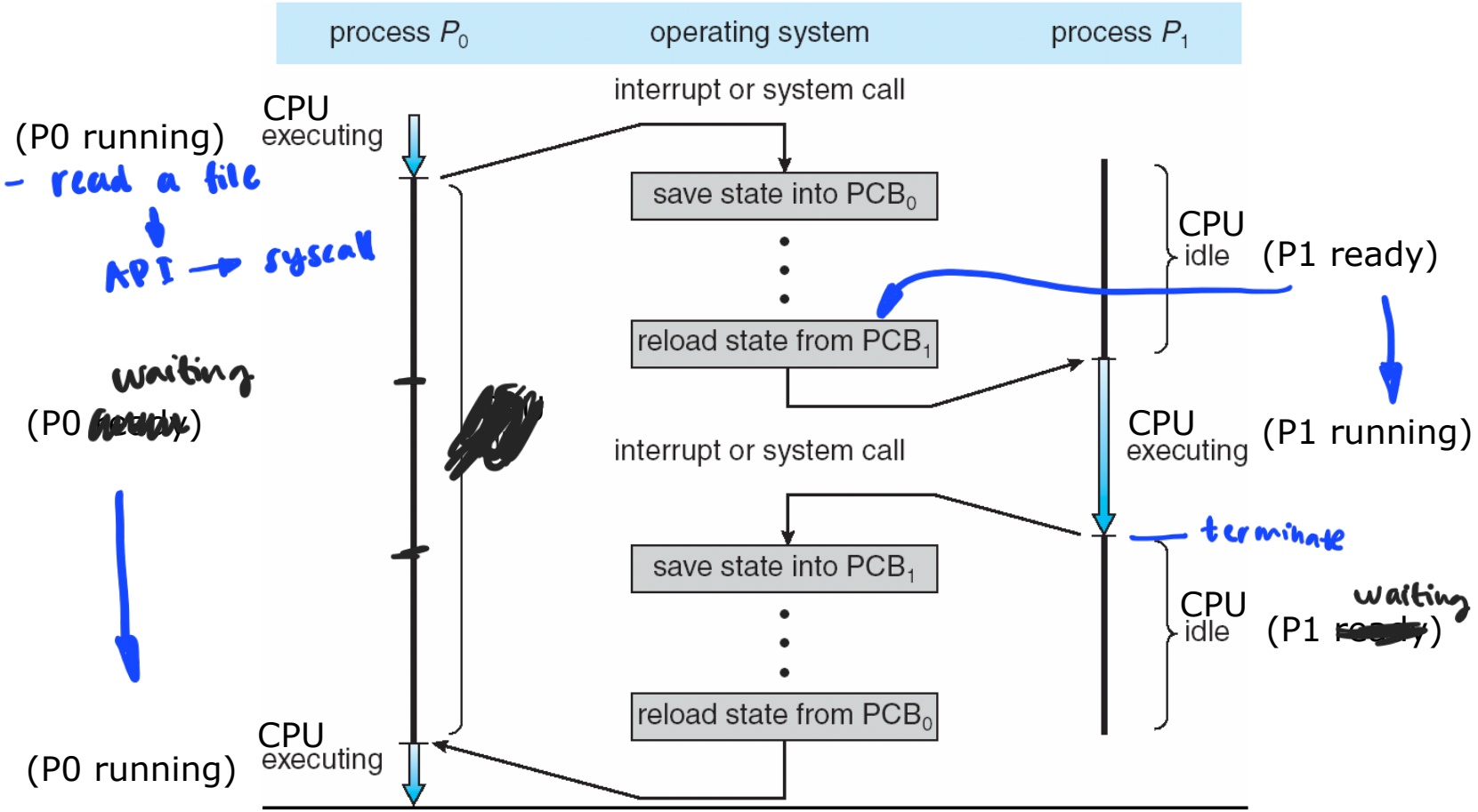


Ready Queue And Various I/O Device Queues

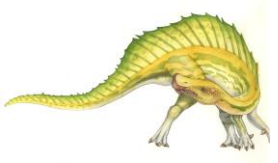


Real operating systems have lots of pointers to implement these interconnected queues.

CPU Switch From Process to Process



which PCB will be reloaded is not something under the control of the process, it is a CPU-scheduling decision!



Quizzes

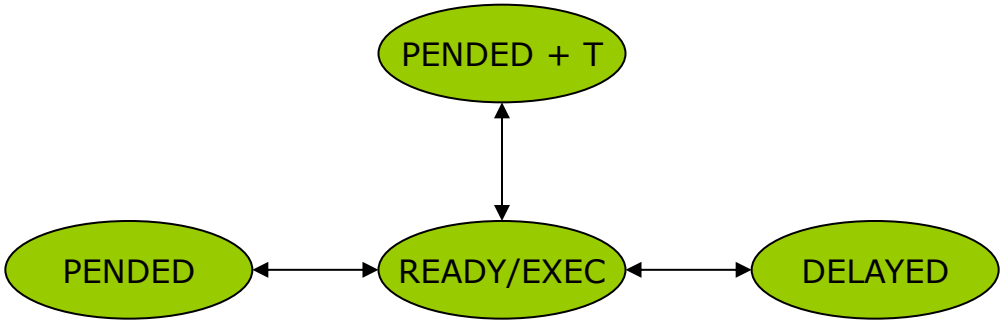
there is an information of the next queue, but it's only a pointer

- A PCB can represent only one process in the system — true
- The list of open files is part of the information contained in the PCB — true
- Pure multiprogrammed systems have no interrupt handling — false, because we still need "waiting" state

or no interactive

Processes in VxWorks

- All tasks must be in one of the 4 basic states:
  - READY (eligible to execute)
  - DELAYED (waiting for an amount of time to pass)
  - PENDED (waiting for an object event, such as a semaphore becoming available)
  - PENDED + T (waiting for an object event, but with a timeout specified)

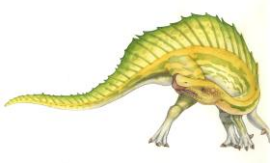


- In addition to its basic state, a task may be *tagged* as **SUSPENDED**, STOPPED, or both, thus marking the task as ineligible to execute

Source: Wind River Education Services

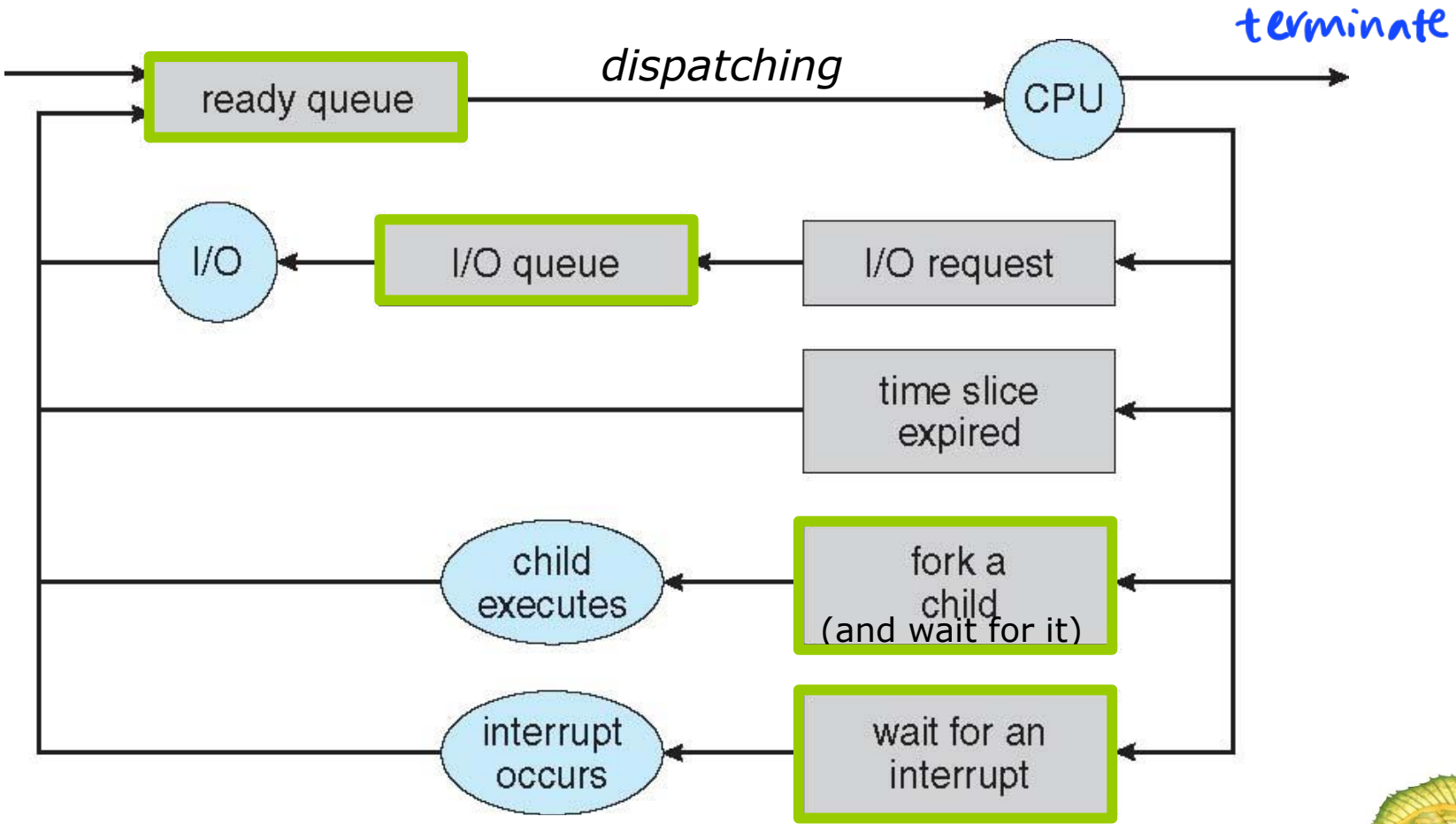
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system (regardless the state)
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



Representation of Process Scheduling

- **Queuing diagram** represents queues, resources, flows



Schedulers

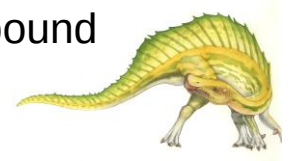
long-term
short-term
medium-term

- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue.
  - For batch systems
    - May not be present in multi-task systems (e.g., often UNIX and Windows)
  - Operates even before the processes are created
  - Invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming** (i.e., the number of processes that are concurrently active)
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)

access to memory (important for system w/ small mem.)

decides which process will be executed

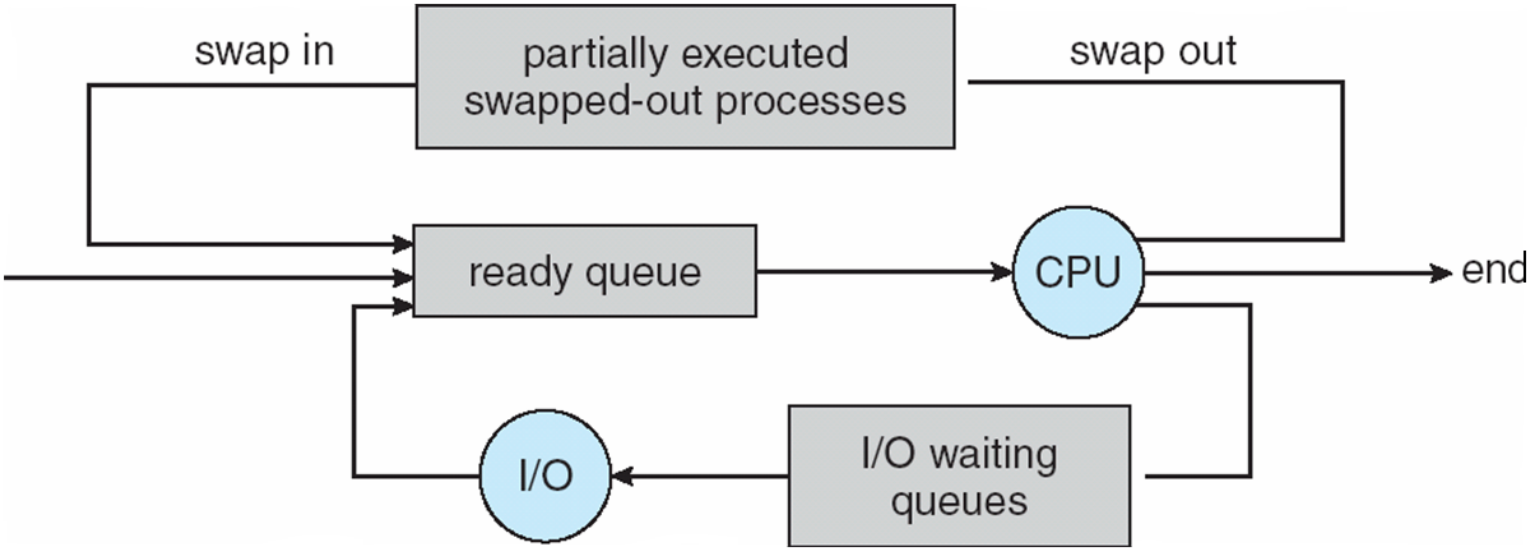
- Processes can be described as either:
  - **I/O-bound process** – more time doing I/O, many short CPU bursts
  - **CPU-bound process** – more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix*** between I/O- and CPU-bound



Addition of Medium-Term Scheduling

which process into disk or memory

- **Medium-term scheduler** can be added if degree of multiprogramming needs to decrease
  - Remove process from memory, store on disk (in the **backing store**, a.k.a **swap partition**), bring back in from disk to continue execution: **swapping**



Context Switch

unload / load the process

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB. Includes both the memory-image of the process and the CPU-image of the process

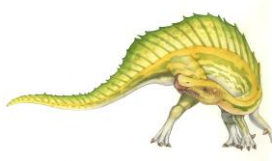
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → longer the context switch

more PCB → more processing time

- Time dependent on hardware support. E.g.,

may need a dedicated hardware to save PCB

  - Specific hardware dedicated to save registers back and forth in one single instruction
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once



Exercise

- Consider a system with 1 CPU, running 10 100% CPU-bound jobs
- The system is **not multitasking**
- Assume the following times:
  - Total CPU time needed by each job: 10s
  - CPU scheduling: 5ms
    - Needed only once, in the beginning
  - Context switch: 5ms
    - Needed every time a job becomes running (10 times)

- If jobs are executed batch (**no multitasking**), how long does it take…
  - For all jobs to complete?
  - For the first job to complete?
  - For the average job to complete?

Exercise

- Consider a system with 1 CPU, running 10 100% CPU-bound jobs.
- The system is **multitasking**
- Assume the following times:
  - Total CPU time needed by each job: 10s
  - Maximum CPU burst before context switch: 100ms
  - CPU scheduling: 5ms
    - Needed in the beginning and every time a process is suspended
  - Context switch: 5ms
    - Needed every time a job becomes running (10 times)
- If jobs are executed interactively (**multitasking**), how long does it take…
  - For all jobs to complete?
  - For the first job to complete?
  - For the average job to complete?
- What is the operating-system overhead?

Exercise

- **Not multitasking**
- 10 jobs; 10s for execution
- 5ms for scheduling
- 5ms x 10 for context switch

- All jobs done after 100s and 55ms
- First jobs done after 10s and 10ms
- On average, jobs done after about 55s

- **Multitasking**
- 10 jobs; 10s for execution
- Max burst: 100ms
  - 100 bursts to complete a job
  - 1000 bursts in total
- 5ms x 1000 for scheduling
- 5ms x 1000 for context switch

- All jobs done after 110s
- First jobs done after 109s and 10ms
- On average, jobs done after about 109s

Process Creation

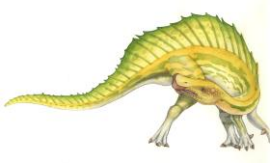
child

each process is created by parent process

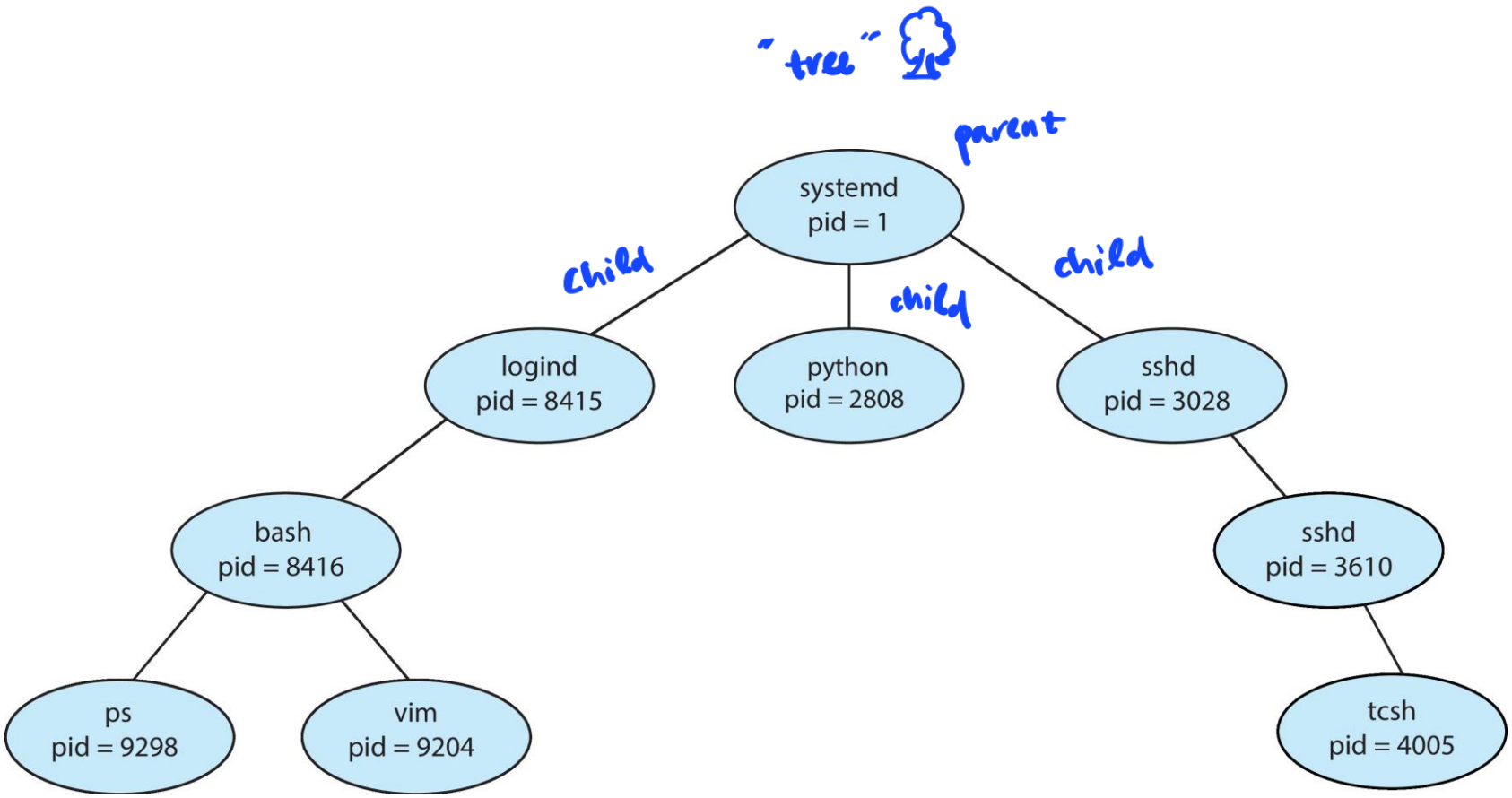
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)

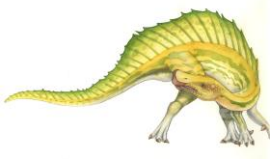
they share resources but not all

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate



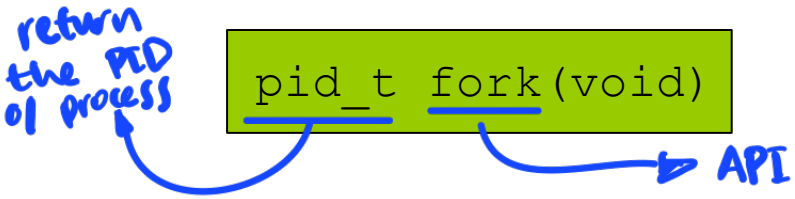
A Tree of Processes in Linux





Process Creation

- Process creation is realized through the system call:



- Creates a new process.
- Initially the child has a copy of the parent's context including code, variables and program counter
- files opened by the parent before the fork are shared with child not copied
- returns a `pid_t` (usually **int**)
  - < 0 if **error** occurred (e.g., not enough memory to create a new process)
  - = 0 for the **child**
  - > 0 for the **parent**. It is the PID of the newly created child process

Process Creation (super-basic example)

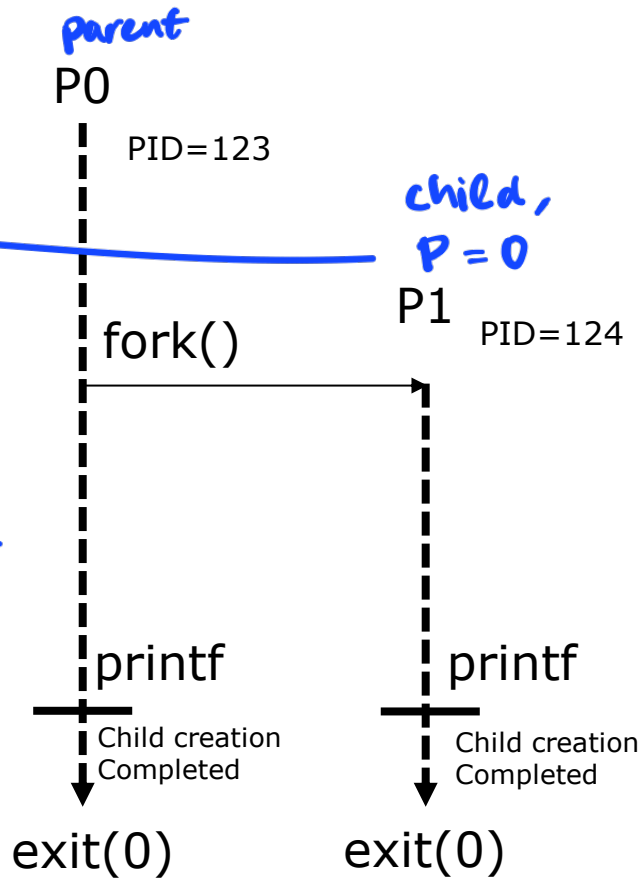
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
int p;
/* fork a child process */
p = fork();
if (p < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   exit(1);
}else {
   printf("Child creation Completed");
   exit(0);
}
```

creating new process

print an error

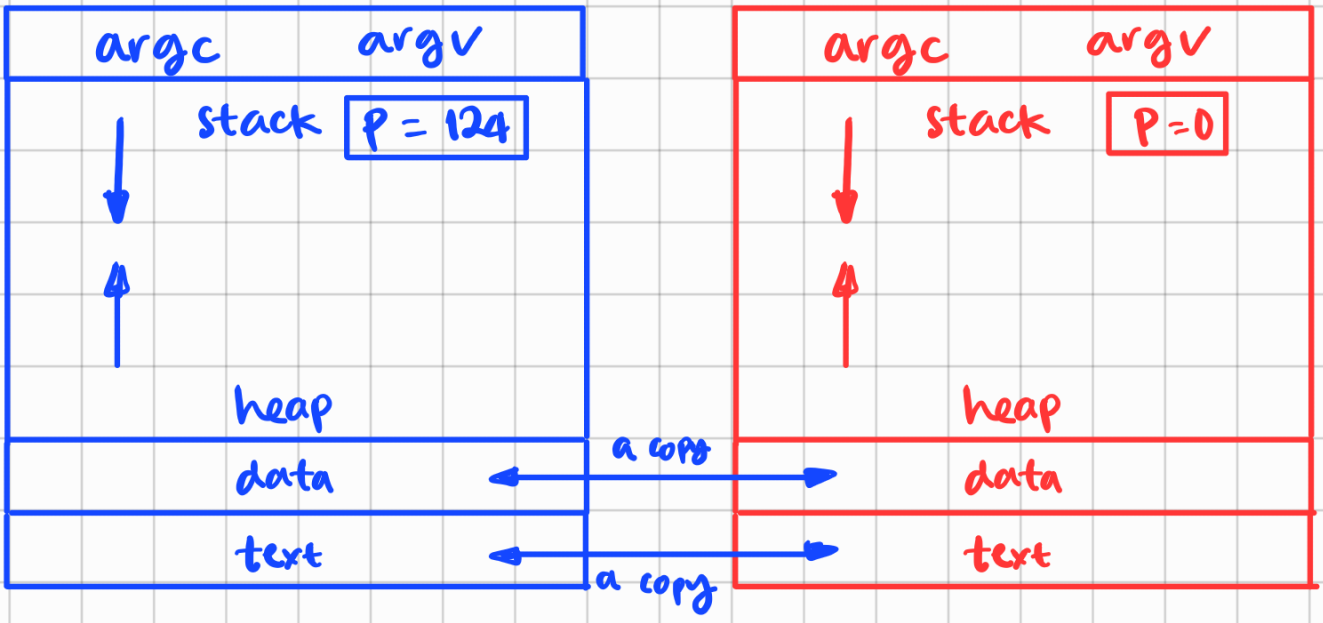
in main, >0 error

in main, 0 = okay



In parent, P0 :

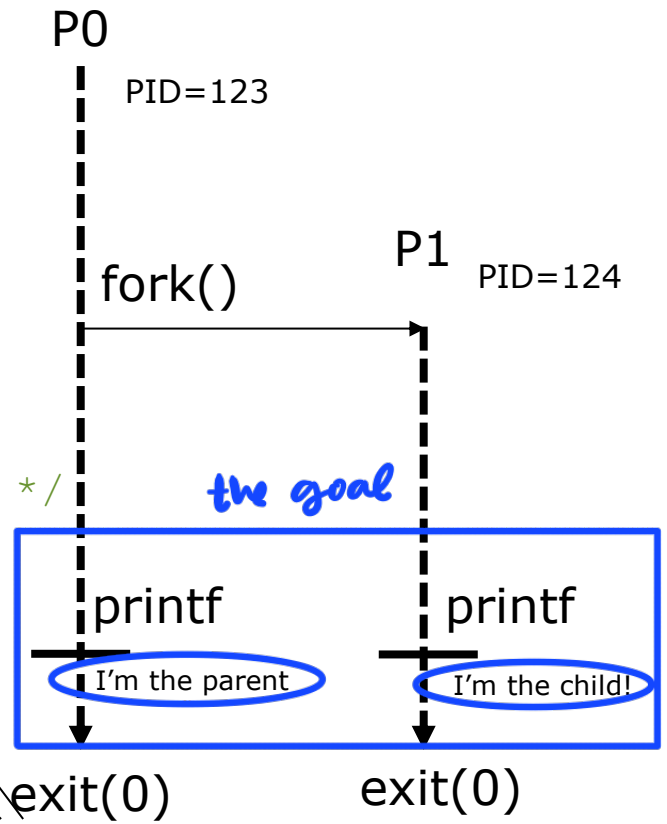
child P1 :



Process Creation (basic example)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(1);
    }else if (p == 0) { /* child process */
        printf("I'm the child!");
    }else { /* parent process */
        printf("I'm the parent!");
    }
    exit(0);
}
```

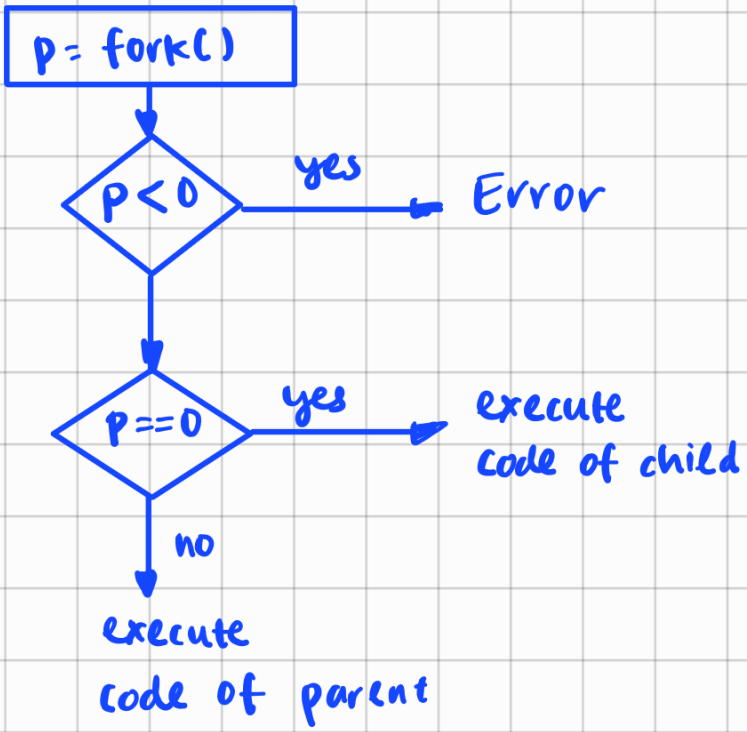
child will do this line



the order in which these will be printed is not predictable
Decision of the CPU!

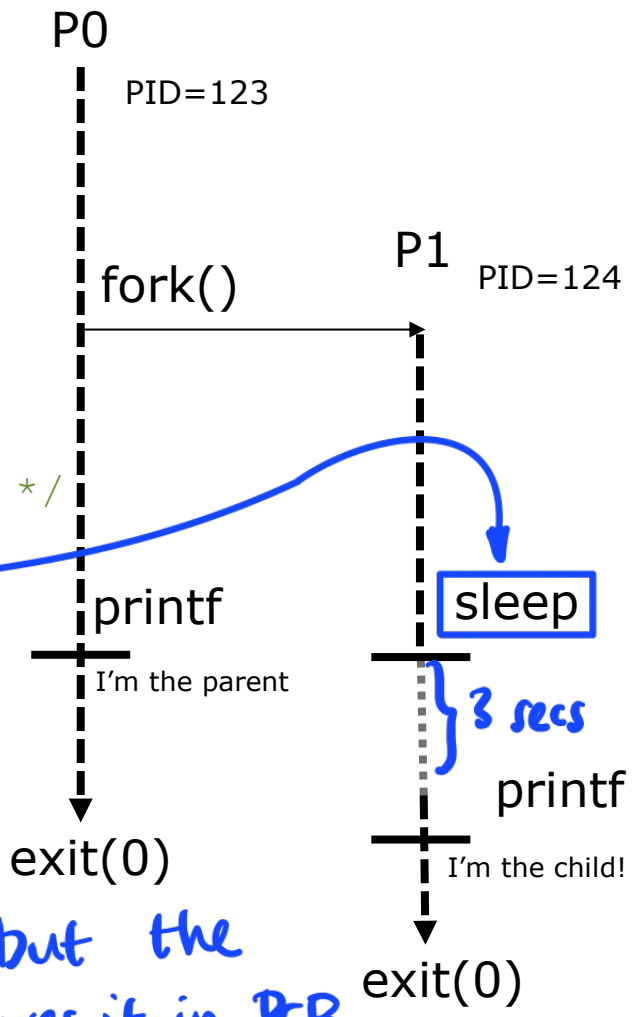
→ Could a sleep(n) help?

The algorithm of the code above:



Process Creation (less basic example)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(1);
    }else if (p == 0) { /* child process */
        printf("I'm the child!");
        sleep(3);
    }else { /* parent process */
        printf("I'm the parent!");
    }
    exit(0);
}
```



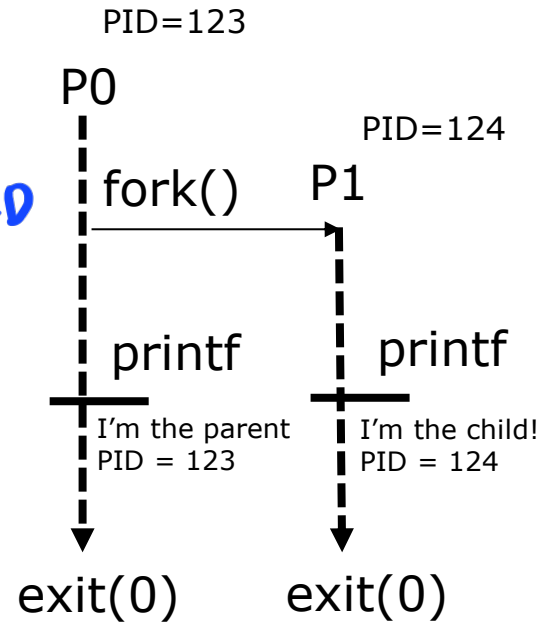
for now, the child doesn't know the parent, but the OS stores it in PCB

Get the Process Identifier

```
pid_t getpid(void)
```

API to tell our PID

- Returns the PID of the process




```
#include …
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(1);
    }else if (p == 0) { /* child process */
       printf("I'm the child! PID = %d", getpid());
    }else { /* parent process */
       printf("I'm the parent! PID = %d", getpid());
    }
    exit(0);
}
```

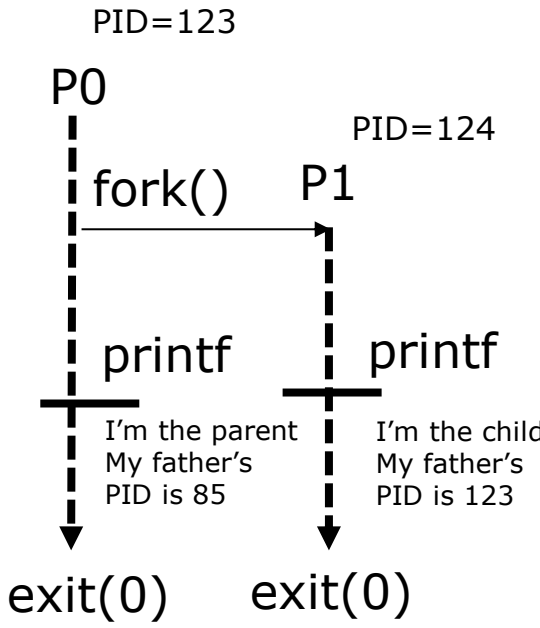
show int. marker

show int. marker

Get the parent's Process Identifier

```
pid_t getppid(void)
```

- Returns the PID of the parent process




```
#include …
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(1);
    }else if (p == 0) { /* child process */
        printf("I'm the child! My father's PID is %d", getppid());
    }else { /* parent process */
        printf("I'm the parent! My father's PID is %d", getppid());
    }
    exit(0);
}
```

Process Creation (two children example)

```
#include …
int main() {
    int p1, p2; char ch;
    p1 = fork();
    if (p1 < 0) {
       fprintf(stderr, "First fork Failed");
       exit(1);
    }else if (p1 == 0) {
       printf("I'm the FIRST child!");}
    else {
       printf("I'm the parent, First child created!");
       p2 = fork();
       if (p2 < 0) {
          fprintf(stderr, "Second fork Failed");
          exit(1);
       }else if (p2 == 0) {
          printf("I'm the SECOND child!");}
       else {
          printf("I'm the parent, Second child created!");
       }
       exit(0);
}
```

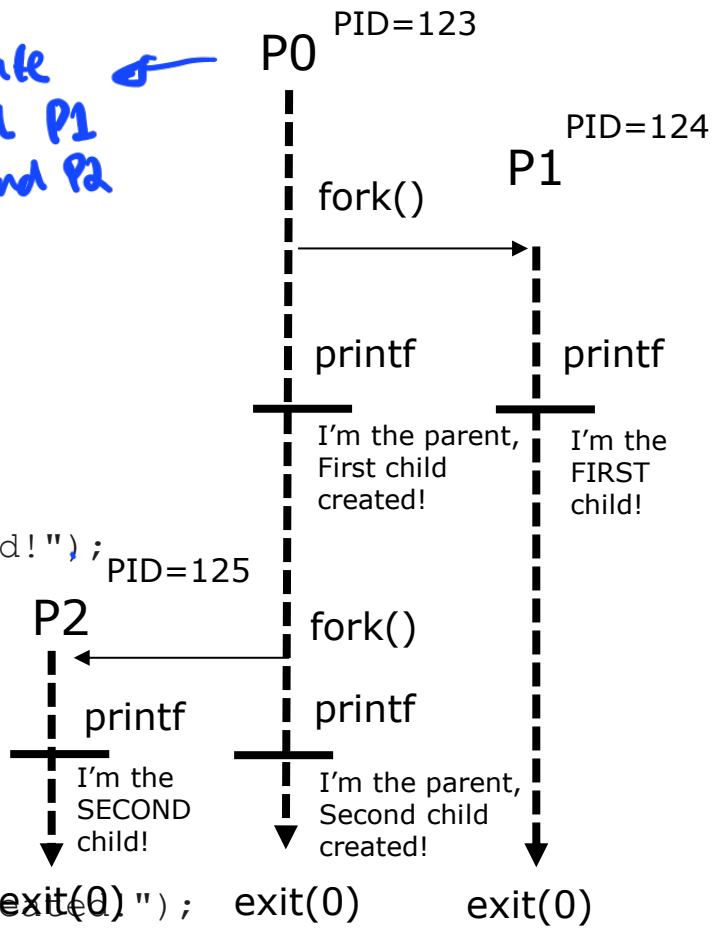
p1 code

parent code

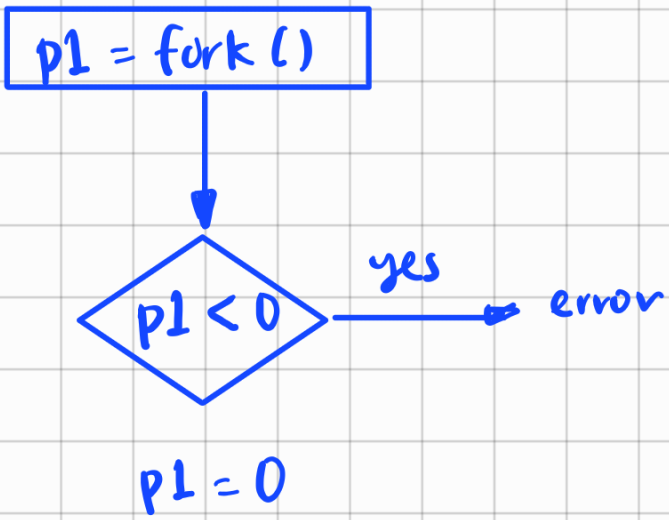
p2 code

parent code

create child P1 and P2



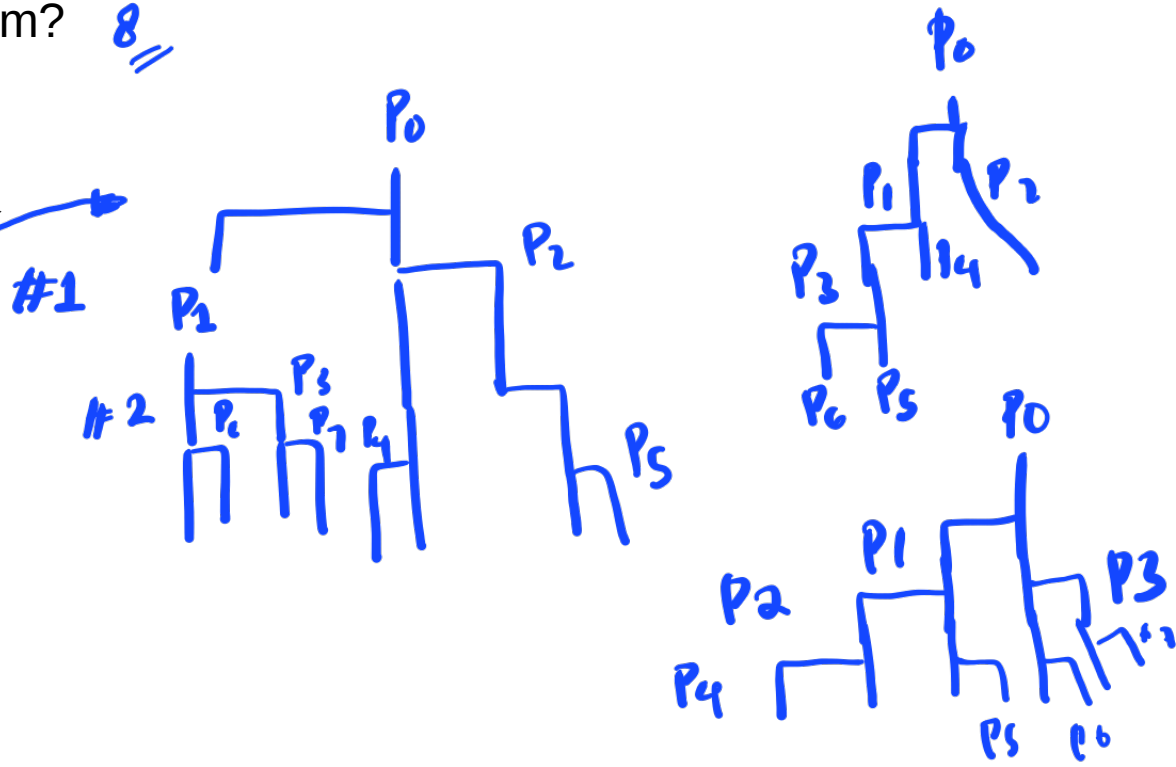
The algorithm above:



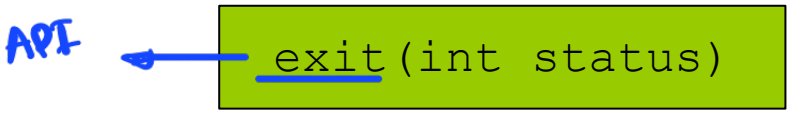
Quizzes

- [T] [F] A child process, right after creation, shares variables with its parent
  - it's a COPY, NOT sharing variables
- [T] [F] A child process, right after creation, has the same list of open files as its parent
- Including the initial parent process, how many processes are created by the following program? 8

```
#include <stdio.h>
#include <unistd.h>
int main() {
   fork();
   fork();
   fork();
      exit(0);
}
```



Process Termination



- Causes the process to terminate
- Asks the OS to remove the process structure from memory
- Asks the OS to close any file not shared with others
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- For convention status is:
  - 0 for no error
  - 1 to 255 for error condition
- Indirectly called by a `return` statement in the `main` function

Process Creation with super-basic synchronization

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(1);
    }else if (p == 0) { /* child process */
        printf("I'm the child!");
    }else { /* parent process */
        wait(NULL);
        printf("I'm the parent!");
    }
    exit(0);
}
```

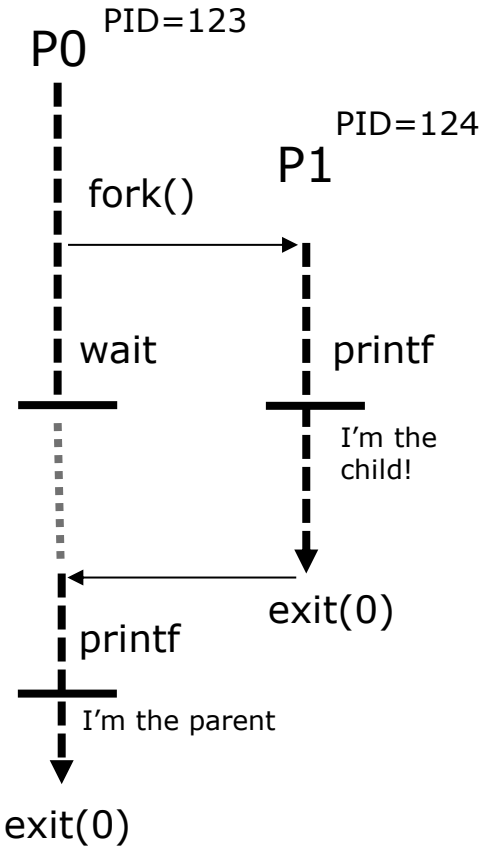
error lines

child lines

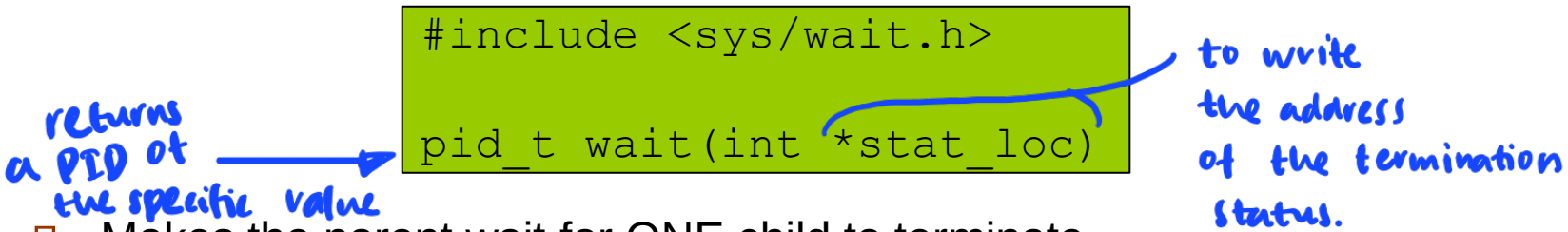
parent lines

wait for something happens

parent waits for child termination



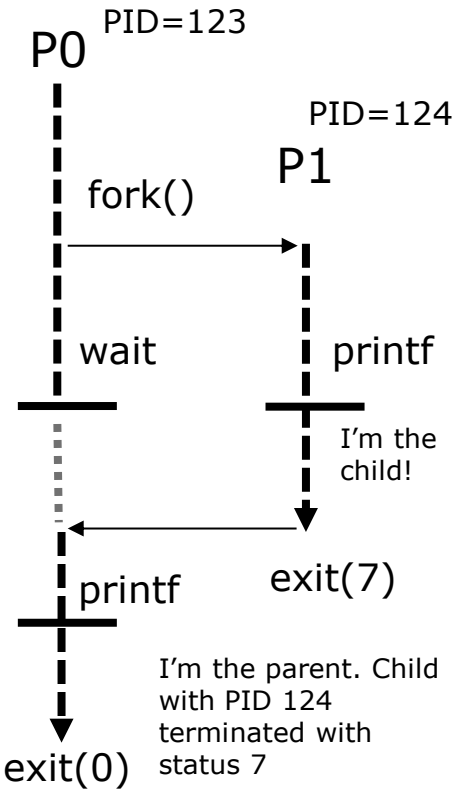
Parent-child synchronization



- Makes the parent wait for ONE child to terminate.
  - e.g., parent with two children must call `wait` twice
  - no control on which child will terminate first
- returns the PID of the terminated child
- `int *stat_loc` allows the parent to get the child's termination status, i.e., the parameter of the `exit` syscall executed by the child.
- `stat_loc` parameter is passed by reference

Detect child termination status

```
#include …
#include <sys/wait.h>
int main() {
    int p;
    /* fork a child process */
    p = fork();
    if (p < 0) { /* error occurred */
       …
    }else if (p == 0) { /* child process */
       printf("I'm the child!");
       exit(7);
    }else { /* parent process */
       int child_pid, status;
       child_pid = wait(&status);
       printf("I'm the parent. Child with PID=%d terminated with
    status %d", child_pid, WEXITSTATUS(status));
    }
    exit(0);
}
```



exec() family of syscalls

- After fork() the child receives a copy of the parent code and data
- The syscalls of the exec() family allow loading a different code
  - execl()
  - execlp()
  - execle()
  - execv()
  - execvp()
  - execvpe()

They do not create any new process!

execl()

```
int execl(char *pathname, char *arg0, …char* argN, NULL)
```

- `pathname` is the path to the executable file to be loaded
- `arg0` is the name of the program (how would you call it from the shell CLI)
- `arg1…argN` are the program parameters
  - As many as needed
- `NULL` to say that the input parameter list is finished

Using execl()

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    printf("Process started\n");
    printf("Launching the ls command with -l option:\n");
    execl("/bin/ls", "ls", "-l", NULL);
    fprintf(stderr, "Error occurred while calling execl \n");
    exit(1);
}
```

Key points:
1. No new process, just code substitution
2. If everything goes well, there is no return

execlp()

```
int execlp(char *filename, char *arg0, …char* argN, NULL)
```

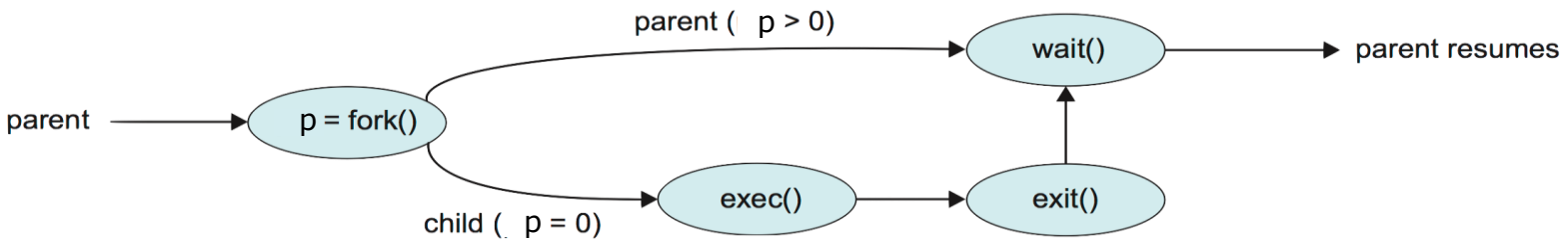
- `filename` is the name of the executable file to be loaded. It will be searched into the `PATH` variable.
- `arg0` is the name of the program (how would you call it from the shell CLI)
- `arg1…argN` are the program parameters
- `NULL` to say that the input parameter list is finished

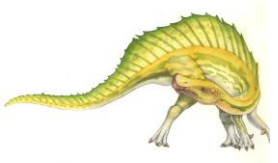
Using execlp()

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   printf("Process started\n");
   printf("Launching the ls command with -l option:\n");
   execlp("ls", "ls", "-l", NULL);
   fprintf(stderr, "Error occurred while calling execl \n");
   exit(1);
}
```

Putting all together

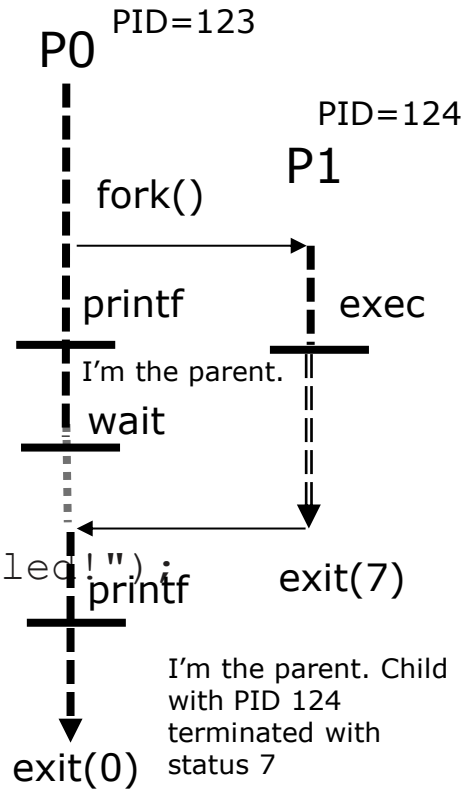
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- Usually the programmer needs:
  - **`fork()`** system call creates new process. Initially it has a copy of the parent's context
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - **`wait()`** to realize a foreground execution. No wait for background.





fork()... exec()... wait()

```
#include …
int main() {
    int p;
    p = fork();
    if (p < 0) {
        fprintf(stderr, "Fork Failed");
        exit(1);
    }else if (p == 0) { /* child process */
        execlp("/bin/pwd","pwd",NULL);
        fprintf(stderr, "I'm the child. execlp failed!");
        exit(2);
    }else { /* parent process */
        printf("I'm the parent!");
        int child_pid, status;
        child_pid = wait(&status);
        printf("I'm the parent. Child with PID=%d terminated with
    status %d", child_pid, WEXITSTATUS(status));
    }
    exit(0);
}
```



Quizzes

- [T] [F] The instructions after `exec*(…)` are never executed
- [T] [F] The wait syscall allows collecting the termination state of a child
- [T] [F] In case of multiple children a single call to the wait syscall allows collecting all the states of all children at once
- [T] [F] `execlp("ls","ls",NULL)` generates a new process dedicated to execute the code of `"ls"`
- Given the following code, in which order will the lines be printed

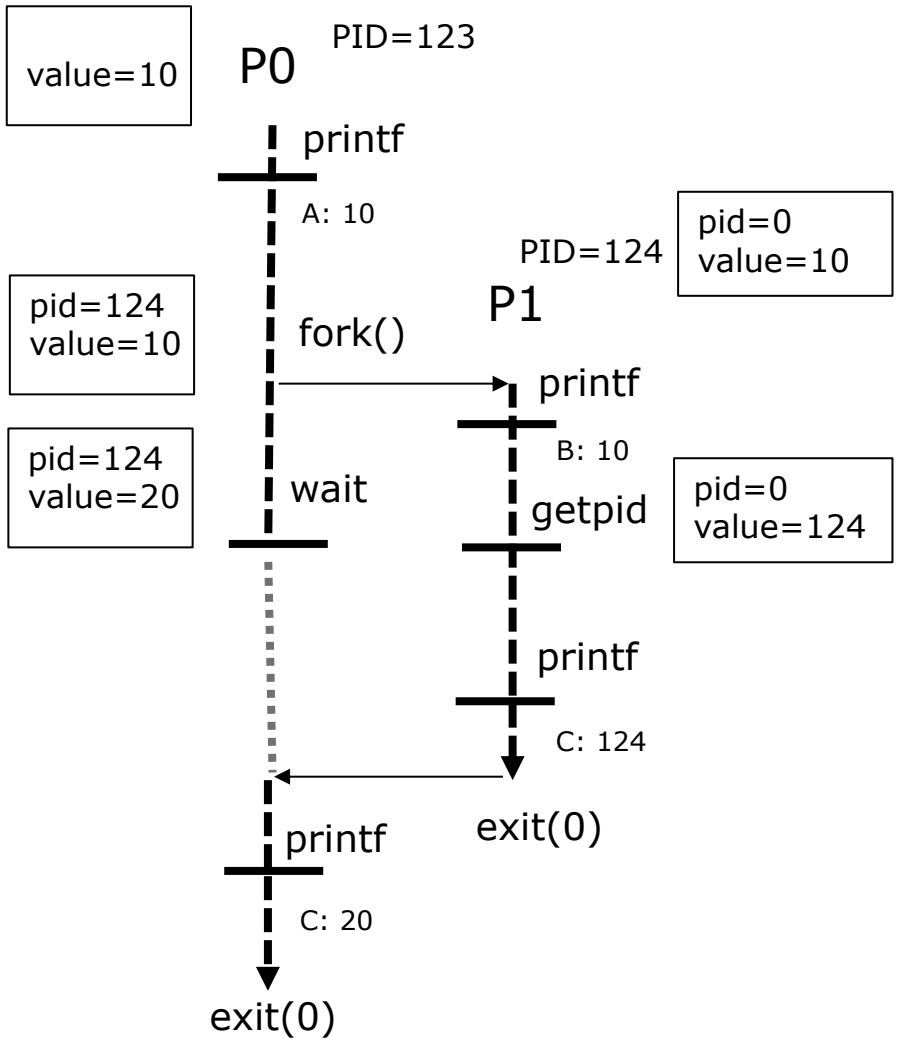
```
int main() {
    int p = fork();
    if (p == 0) {
       sleep(5); /*to make the process sleep for 5 seconds*/
       printf("I'm the child!");}
    else {
       printf("I'm the parent!");
    }
    exit(0);
}
```

fork()/wait() exercise: what output?

```
int value = 10;
int main(){
    pid_t pid;
    int status;
    printf("A: value = %d\n", value);
    pid = fork();
    if( pid==0 ) {
        printf("B: value = %d\n", value);
        value = getpid();
    }else if( pid>0 ) {
        value = 20;
        wait(&status);
    }
    printf("C: value = %d\n", value);
    return 0;
}
```

fork()/wait() exercise: what output?

```
int value = 10;
int main(){
    pid_t pid, status;
    printf("A: value = %d\n", value);
    pid = fork();
    if( pid==0 ) {
        printf("B: value = %d\n", value);
        value = getpid();
    }else if( pid>0 ) {
        value = 20;
        wait(&status);
    }
    printf("C: value = %d\n", value);
    return 0;
}
```

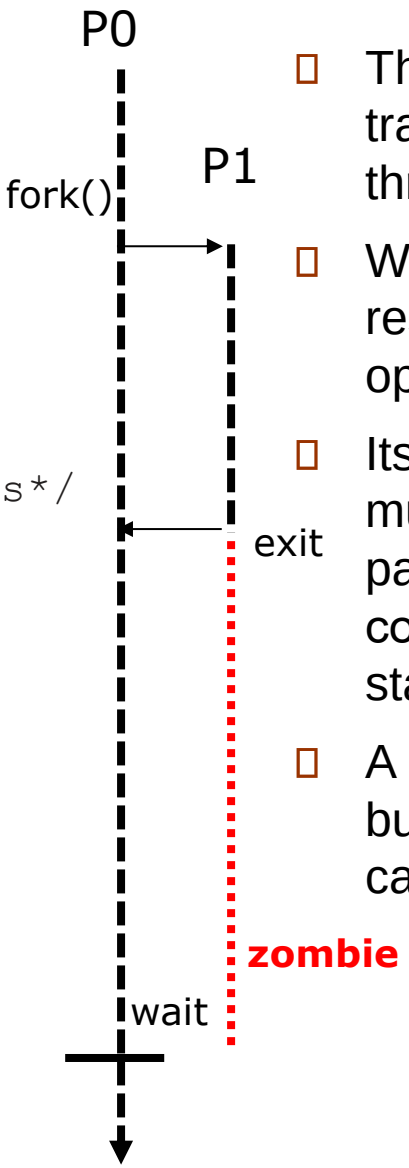


VxWorks example: taskSpawn()

```
#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "sysLib.h"
#include "subsys/timer/vxbTimerLib.h"

TASK_ID WorkerId;

void AuxClkISR(void) {
        static int count=0;
        logMsg("Tick: %d\n", ++count, 0, 0, 0, 0, 0);
        taskResume(WorkerId);
}

void Worker(void) {
        int JobId=0;
        FOREVER {
                taskSuspend(0);
                logMsg("job %d started\n", JobId, 0, 0, 0, 0, 0);
                /* do some computation... */
                logMsg("job %d finished\n", JobId, 0, 0, 0, 0, 0);
                JobId++;
        }
}

void start(void) {
        sysAuxClkRateSet(sysClkRateGet());
        WorkerId=taskSpawn("Worker", 100, 0, 4000, (FUNCPTR) Worker, 0, 0, 0, 0, 0, 0, 0, 0, 0

        sysAuxClkEnable();
        sysAuxClkConnect((FUNCPTR) AuxClkISR, (_Vx_usr_arg_t) 0);
        taskDelay(10*sysClkRateGet());
        sysAuxClkDisable();
}
```

Process table & Zombie state

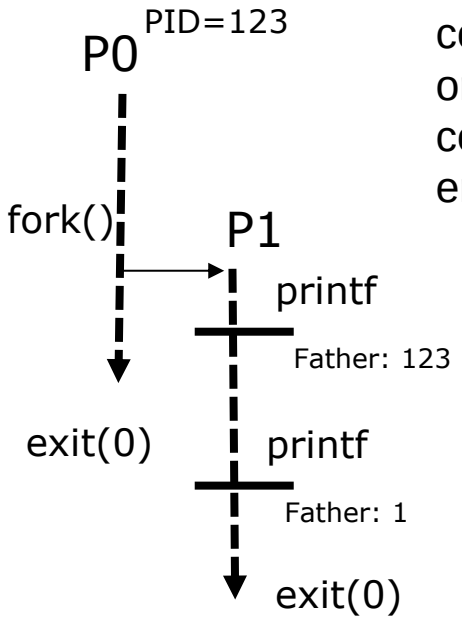
```
#include …
int main() {
    int p;
    p = fork();
    if (p == 0) {
        <instruction p1.1>;
        <instruction p1.2>;
        <instruction p1.3>;
        exit(0);
    }else { /*parent process*/
        <instruction p0.1>;
        <instruction p0.2>;
        <instruction p0.3>;
        wait(NULL);
    }
    exit(0);
}
```



- The operating system keeps track of all the processes through a **process table**
- When a process terminates, its resources are deallocated by the operating system.
- Its entry in the process table must remain there until the parent calls wait(), because it contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait(), is in **zombie** state

Orphan child

```
#include …
int main() {
    int p;
    p = fork();
    if (p == 0) {
        <instruction p1.1>;
        <instruction p1.2>;
        printf("My father is %d", getppid());
        exit(0);
    }else { /* parent process */
        <instruction p0.1>;
        <instruction p0.2>;
        <instruction p0.3>;
        wait(NULL);
    }
    exit(0);
}
```



- Parent can terminate before child (if parent doesn't wait).
- **Orphan** child processes are "adopted" by the init/systemd process.
- Therefore, after parent's termination, a call to **getppid**() by the child will return 1 (PID of init)
- Init periodically invokes wait(), collecting the status of any orphan and releasing the corresponding process table entry

pause() and sleep()

- system calls to put the process into waiting state

```
unsigned int sleep(unsigned int seconds);
```

- makes the process wait for an event from the timer to arrive.
- The event is programmed to arrive after n seconds.
- Return the number of seconds remaining to sleep

```
pause();
```

- makes the process wait indefintely
- untill a **signal** arrives

Asynchronous Notifications: Signals

- Used to alert a process about some event occurring
- Linux defines a fixed set of numbered signals (1-31)
- Some examples:
  - SIGINT (2): terminal interrupt (CTR+C, maskable)
  - SIGKILL (9): kill (unmaskable)
  - SIGUSR1 (10), SIGUSR2 (12): user defined
  - SIGPIPE (13): write on a pipe with no reader
  - SIGALRM (14): alarm clock
  - SIGTERM (15): termination
  - SIGCHLD (17): child process has stopped or exited
  - SIGSTOP (19): stop executing (unmaskable)
- Signals can either be raised (error condition) or sent by another process
- From shell: **`kill -signo pid`**
  - **`kill -10 123`**

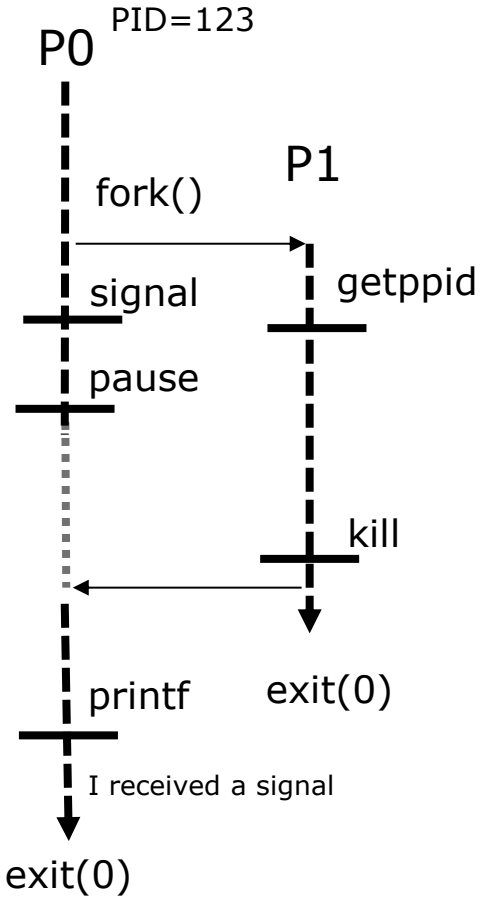
Signal Handling in Linux

- Most signals are *maskable*: can be **ignored** or **caught** and **handled** by processes
- System calls:
  - **`signal(sig, handler)`** associates handler to signal
  - **`kill(pid, sig)`** sends signal to process (also shell command)
- Default action (`SIG_DFL`) is usually termination

```
#include <signal.h>
void my_handler(int sig) {
   printf("I caught signal %d\n", sig);
   signal(SIGINT, SIG_DFL);
}
int main() {
   signal(SIGINT, my_handler);
   ...
}
```

signal()... kill()…

```
#include …
void my_handler(int signo){
   printf("I received a signal!");
}

int main() {
    int p;
    p = fork();
    if (p < 0) { /* error */
        ...
    }else if (p == 0) { /* child */
        fathers_pid = getppid();
        kill(fathers_pid, SIGUSR1);
        exit(0);
    }else { /* parent */
        signal(SIGURS1, my_handler);
        pause(); //wait indefinitely for signal
        exit(0);
    }
}
```



alarm()

```
unsigned int alarm(unsigned int seconds);
```

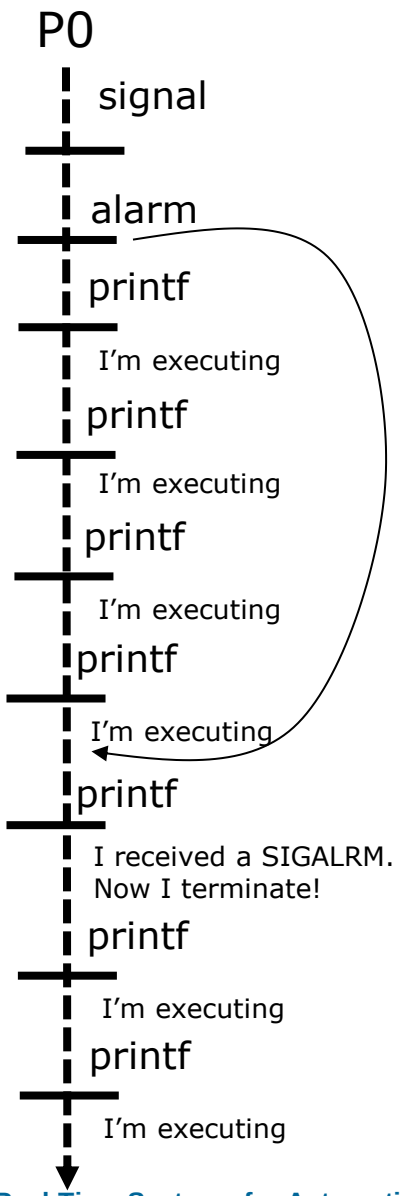
- schedules the sending of a **`SIGALRM`** after n seconds
- The signal will be sent by the OS to the same process that executed the system call
- It does not interrupt the execution. The process will go on executing the next operation without waiting.
- Default action for **`SIGALRM`** is termination
- **`signal(SIGALRM, my_handler)`** to specify another action

signal()… alarm()… Example 1

```
void f(int s) {
        puts("Got it!");
}

main() {
        signal(SIGALARM, f);
        alarm(1);
        pause();
        puts("So long");
}
```

signal()... alarm()… Example 2

```
#include …
void my_handler(int signo){
   printf("I received a SIGALRM. Now I
terminate!");
}

int main() {
    int p;
    signal(SIGALRM, my_handler);
    alarm(3);
    while(1){
       printf("I'm executing…");
    }
    exit(0);
}
```



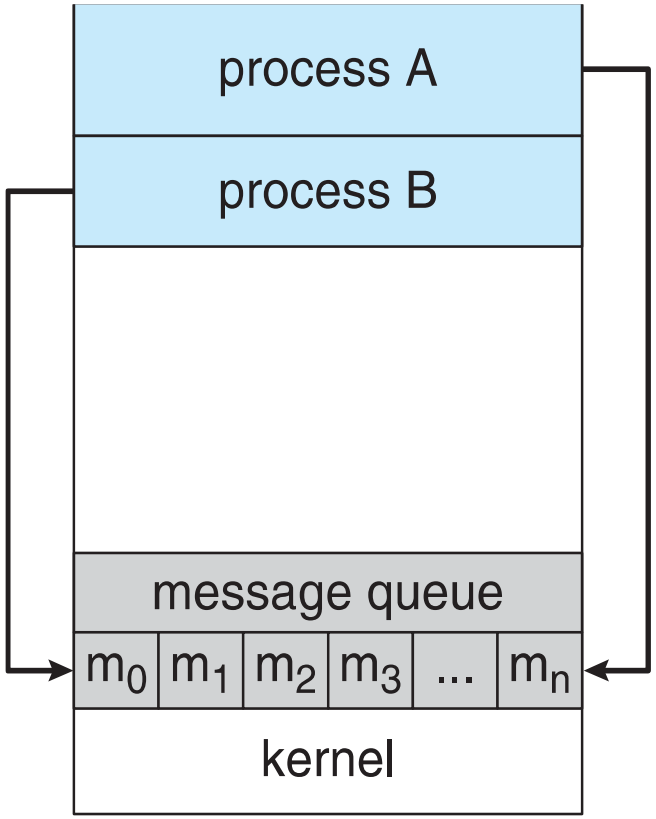
Quizzes

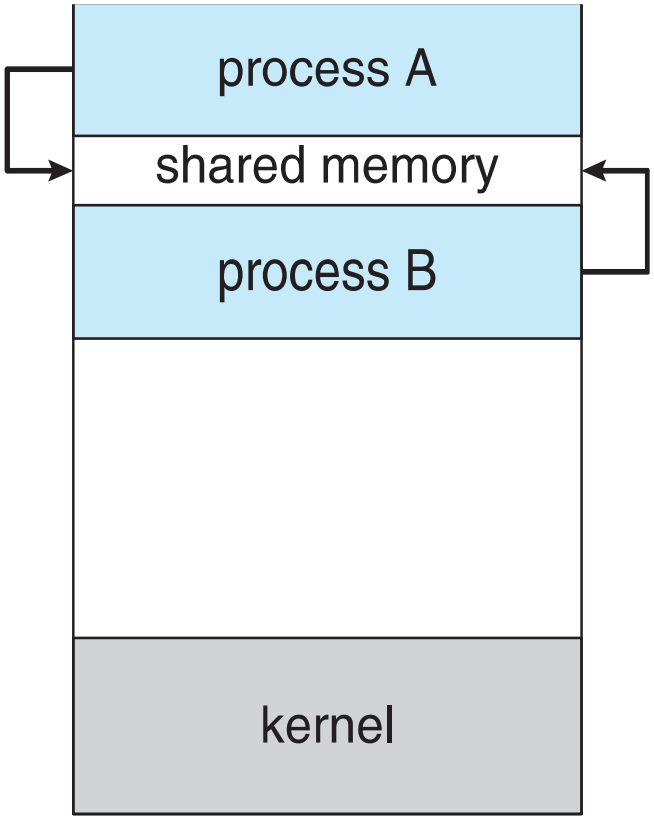
- a maskable signal is a signal that, whenever received, triggers the execution of a user-defined handler
- an unmaskable signal cannot be handled with a user-defined handler
- sleep(n) syscall makes the process wait for n seconds
- alarm(n) syscall makes the process wait for an alarm to arrive after n seconds
- signals are a communication tool

Interprocess Communication

- Processes within a system may be **cooperating (exchanging data)** or **independent (not exchanging)**
- Cooperating process can affect or be affected by other processes, including **sharing data**
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity & convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models





message passing
- create/destroy
- send/receive

SO takes care of synchronization

shared memory
- OS provides create/destroy methods (to extend the process space with a set of addresses that can be used also by others)

The program must take care of synchronization

POSIX Shared Memory

- POSIX Shared Memory (access by name):
  - **Create** a new shared memory object (or open an existing object and share it) using a name (in the namespace of the filesystem, specifying access rights)

ugo

```
shm_fd = shm_open("object_1", O_CREAT | O_RDRW, 0666);
```

int

modality rights

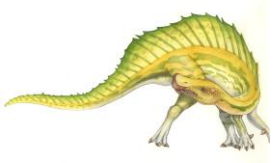
  - **Memory map** the shared memory object, to extend the address space with this shared memory object

```
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

void*

  - **Use** shared memory

```
char msg_0="Writing to shared memory";
sprintf(ptr, msg_0);
ptr += strlen(msg_0);
```

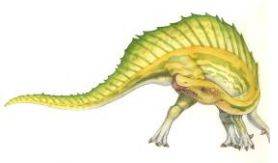


POSIX Producer

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

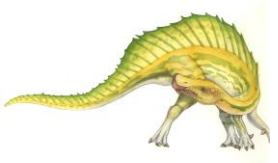
Table of open files

int	Pointers to file in memory



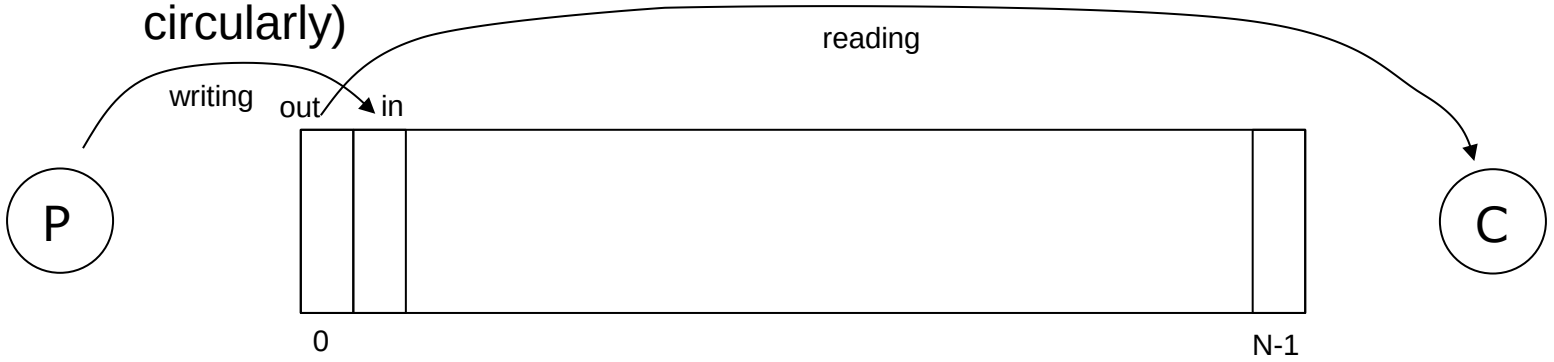
POSIX Consumer

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **Unbounded buffer** places no practical limit on the size of the buffer
  - **Bounded buffer** assumes that there is a fixed buffer size (managed circularly)



If faster than C, must not overwrite

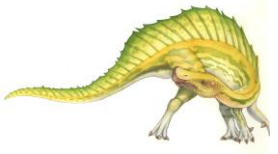
If faster than P, must not consume “rubbish” data

They must wait each other!

If buffer is full, must wait

If buffer is empty, must wait

Both keep an index to know where they read/write (in/out)



Bounded Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

used to understand if the buffer is full or empty

- Notice a limitation: can only use BUFFER_SIZE-1 elements
  - During increment if `in` or `out` are greater than `N` is necessary to "reset" them: `in=(in+1)%N`
- Buffer empty if `out==in`
- Buffer full if `out ==(in+1)%N`

Bounded Buffer – Example

Producer

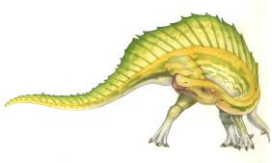
```
item next_produced;
while (true) {
        /* produce an item in next_produced */
        while (((in + 1) % BUFFER SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER SIZE;
}
```

Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;
        /* consume the item in next_consumed */
}
```

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- If *P* and *Q* wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via IPC:
    - **`send`**(*message*) – message size fixed or variable
    - **`receive`**(*message*)
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)



Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must **name** each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q

- Properties of communication link
  - Links are established automatically (must know identity of other)
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as **ports**)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

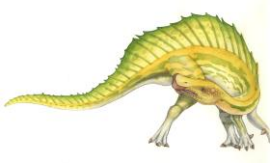
Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A



Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation (mutual exclusion)
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization in message passing

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization in message passing

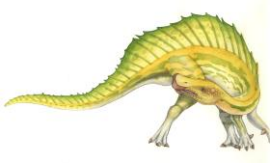
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**
- Producer-consumer becomes trivial:
  - the buffer will never be full or empty cause P and C wait each other

```
message next_produced;

while (true) {
    /* produce an item in next produced */

   send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits (receiver may still have to wait)

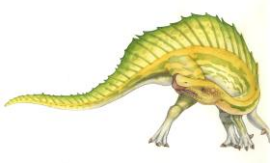


Communications in Client-Server Systems

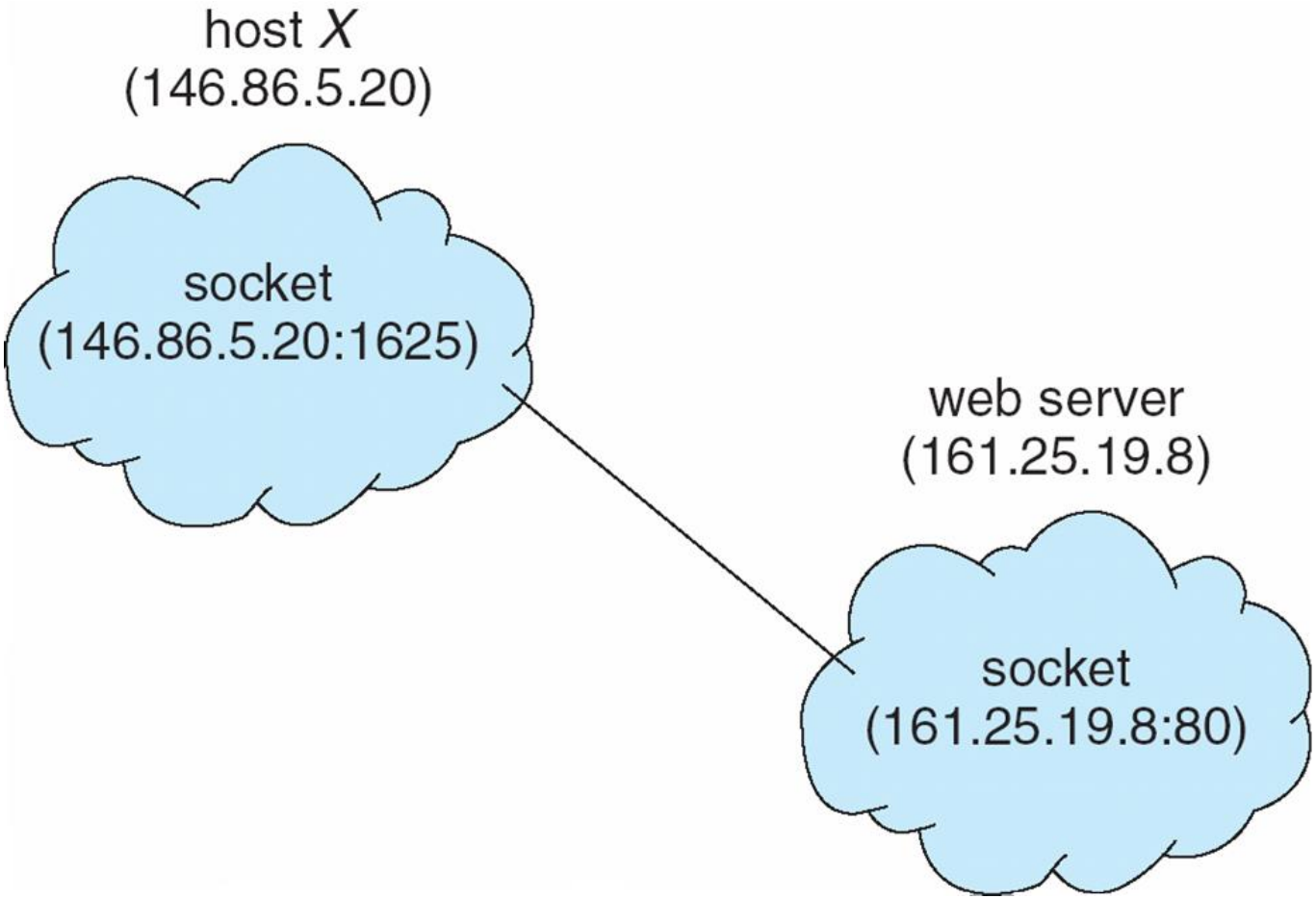
- A process making (consuming) requests, the other answering (producing)
  - Sockets (are the processes on the same machine or not?)
  - Remote Procedure Calls (the interaction is similar to the execution of a function?)
  - Pipes (if processes are on the same machine, do they need to be in some parent/child relation?)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

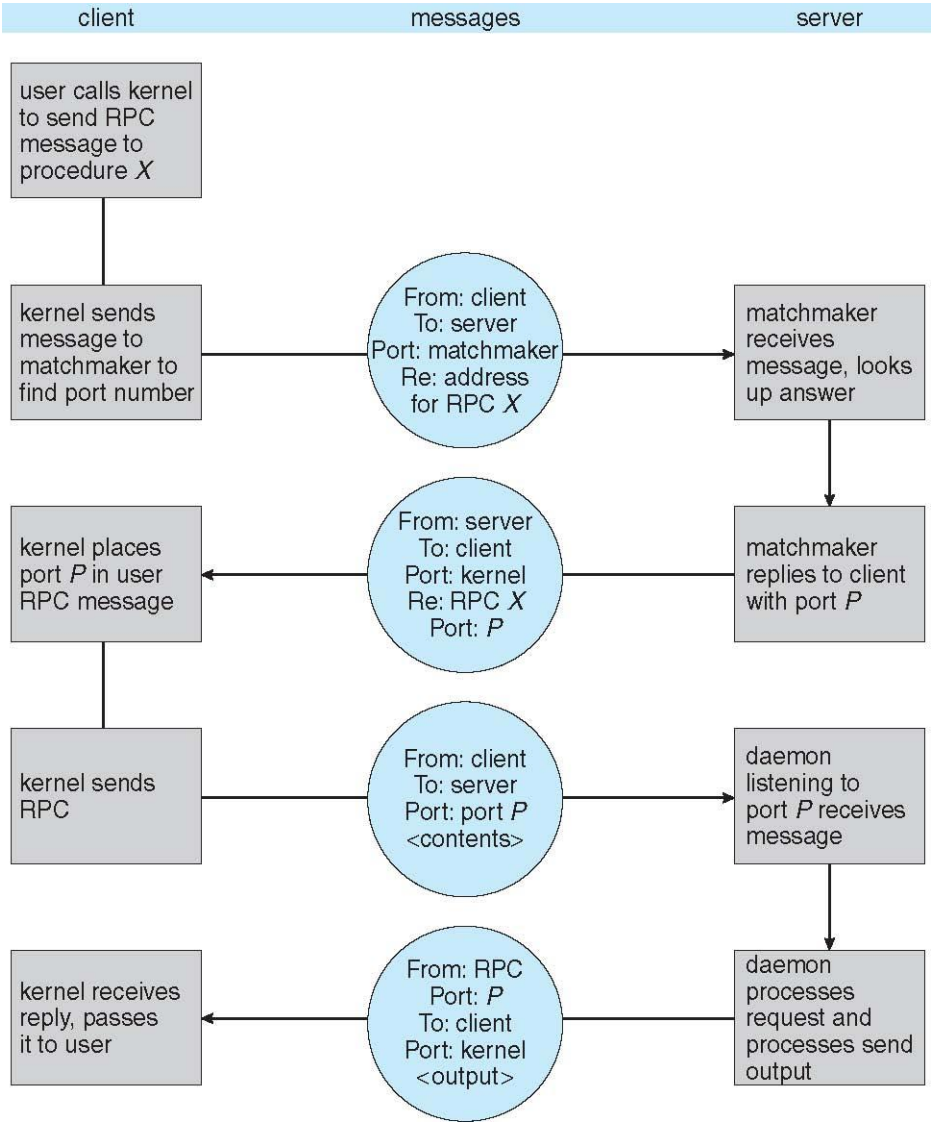


Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems

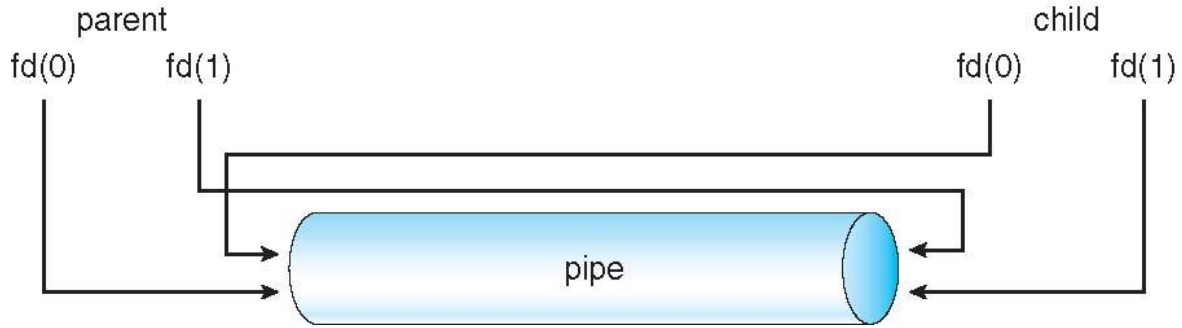


Pipes

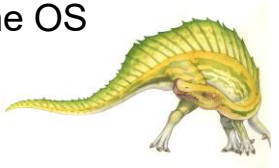
- Acts as a conduit allowing two processes to communicate
- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
    - How many ways can be used at the same time?
  - Must there exist a relationship (i.e. ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
  - See pipes in **shell**
- Require parent-child relationship between communicating processes

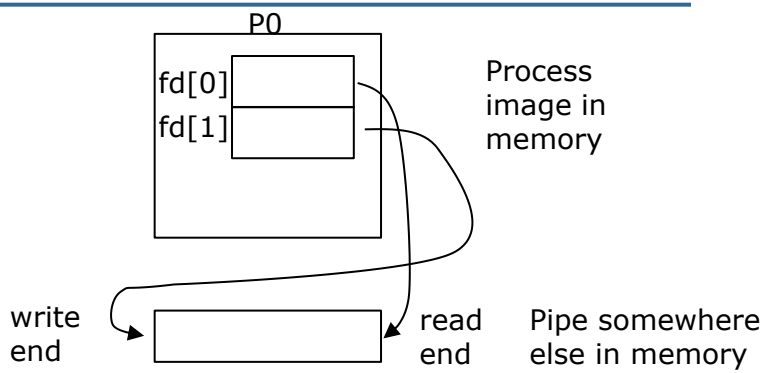


- Remember to close unused end of pipe from start, and other end once done
- It is a shared space in memory BUT it's not managed by the programmers but by the OS
  - OS takes care of size of buffer, point where to read/write, synchronization, etc



pipe() system call

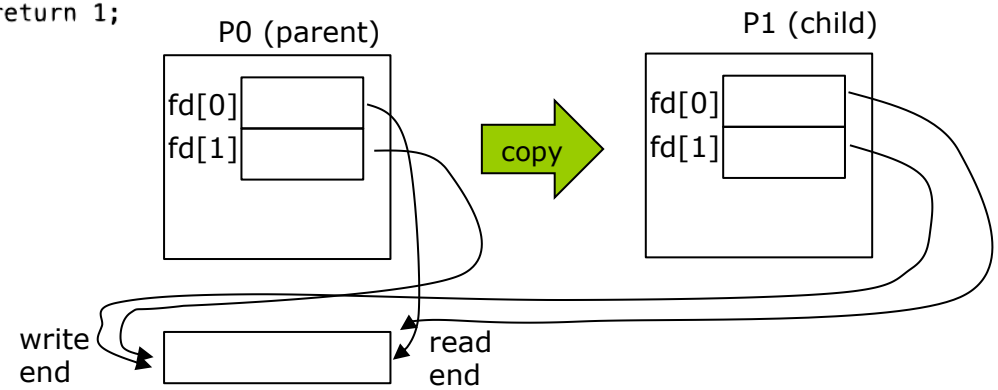
```
int pipe(int fd[2]);
```

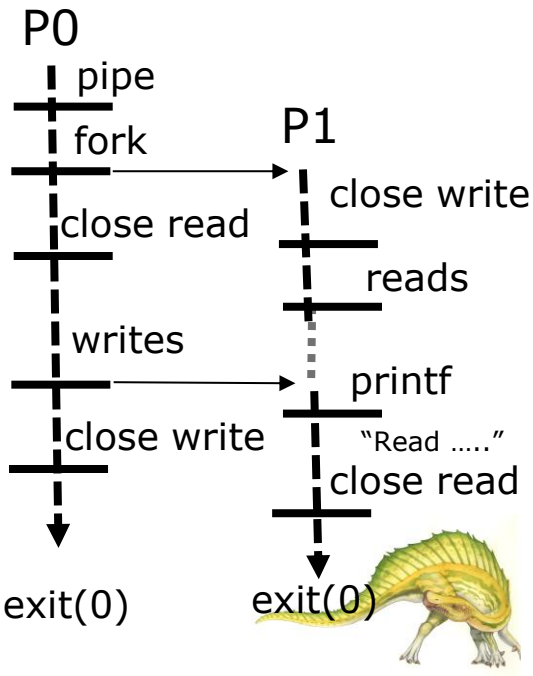


- Creates an ordinary pipe
- input parameter: takes a pointer to an array of two integers.
  - It is actually an output parameter: the OS writes it!
- After the syscall is performed
  - fd[0] will be the file descriptor of the read-end of the pipe
  - fd[1] will be the file descriptor of the write-end of the pipe
- returns
  - 0 if everything worked
  - -1 if error

Pipes Example

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void)

{
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];

  int fd[2];
  pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
     fprintf(stderr,"Pipe failed");
     return 1;
   }

   /* fork a child process */
   pid = fork();
   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }

   if (pid > 0) { /* parent process */
     /* close the unused end of the pipe */
     close(fd[READ_END]);
     /* write to the pipe */
     write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
     /* close the write end of the pipe */
     close(fd[WRITE_END]);
   }

   else { /* child process */
     /* close the unused end of the pipe */
     close(fd[WRITE_END]);
     /* read from the pipe */
     read(fd[READ_END], read_msg, BUFFER_SIZE);
     printf("read %s",read_msg);
     /* close the read end of the pipe */
     close(fd[READ_END]);
   }

   return 0;
}
```





Reading from and writing to pipes

- read/write syscall have the same interface for both files and pipes.
- The OS implementation will differ
  - read/write on pipe returns the number of actually read/written bytes (as for files)
  - read on empty pipe is blocking ➔ process in waiting state
  - read from a pipe with closed writing ends returns 0 (immediately, no waiting)
  - write on full pipe may block or fail (configurable behaviour)
  - writing to a pipe with closed reading ends ➔ error condition. OS notifies the writing process of the error with a signal

Named Pipes

- Named Pipes (FIFO in Unix) are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes (but must be on same machine)
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

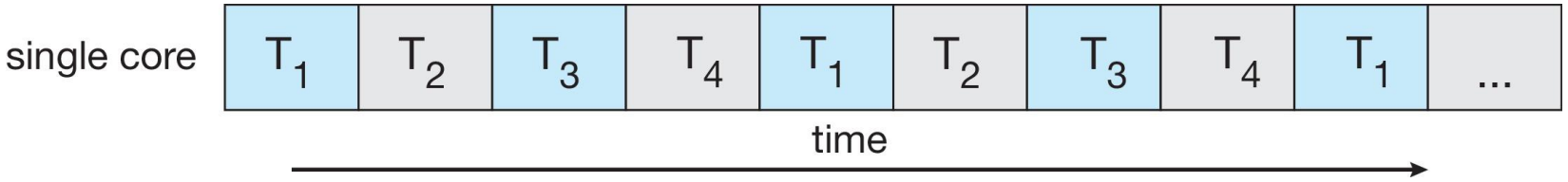
…more details about this on third lab

Quizzes

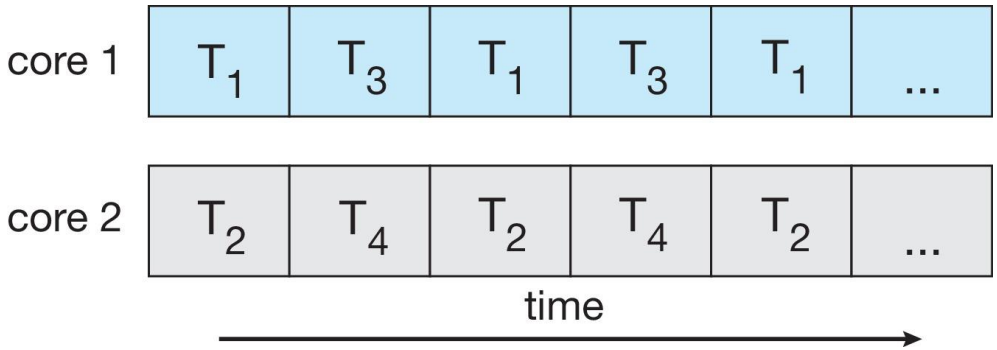
- In direct communication processes must know each other's "name"
- buffers are employed only for shared memory communication
- pipes are direct, blocking communication tools

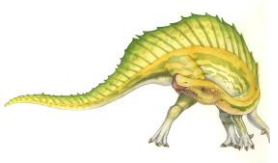
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

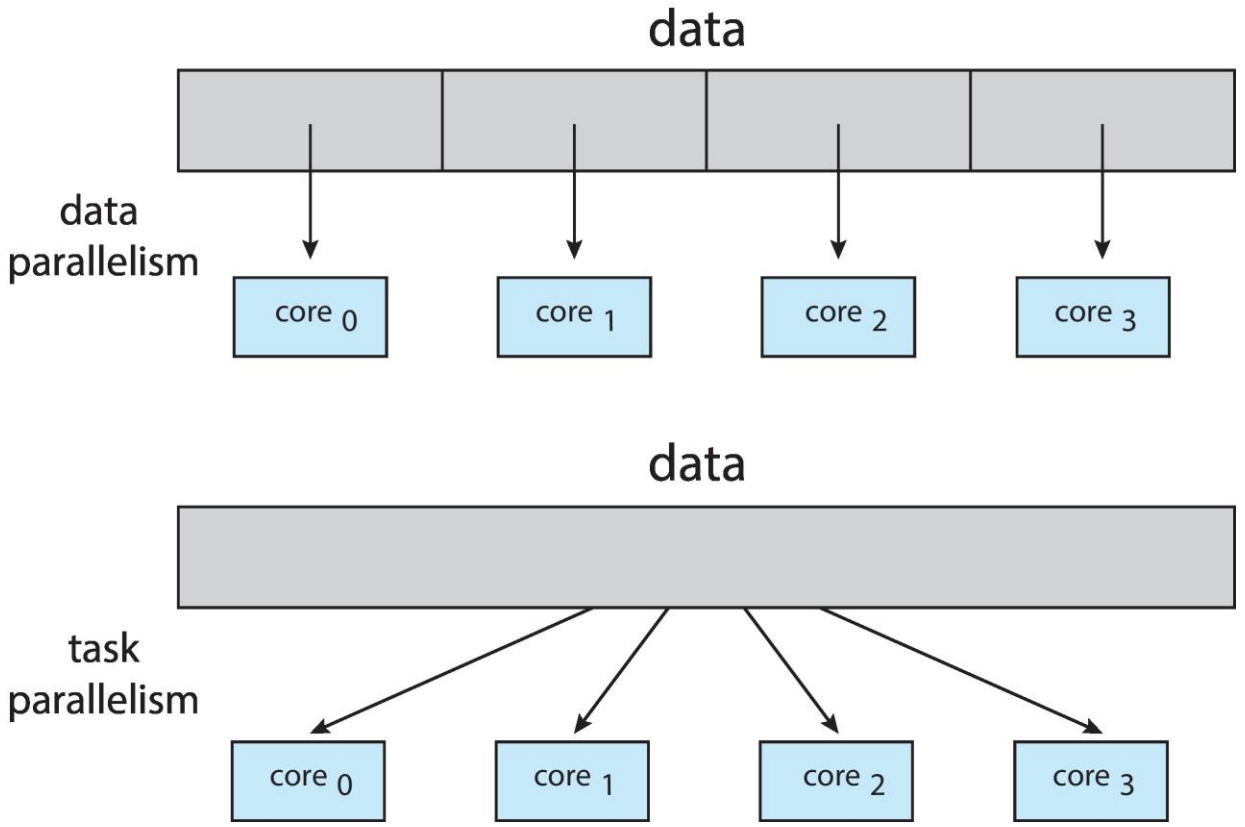


- **Parallelism on a multi-core system:**



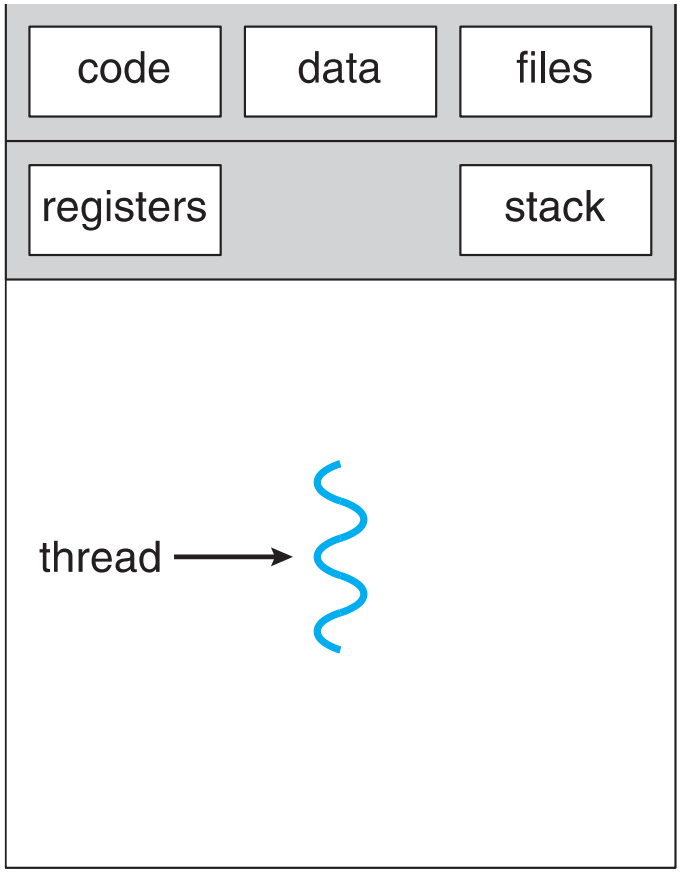


Data and Task Parallelism

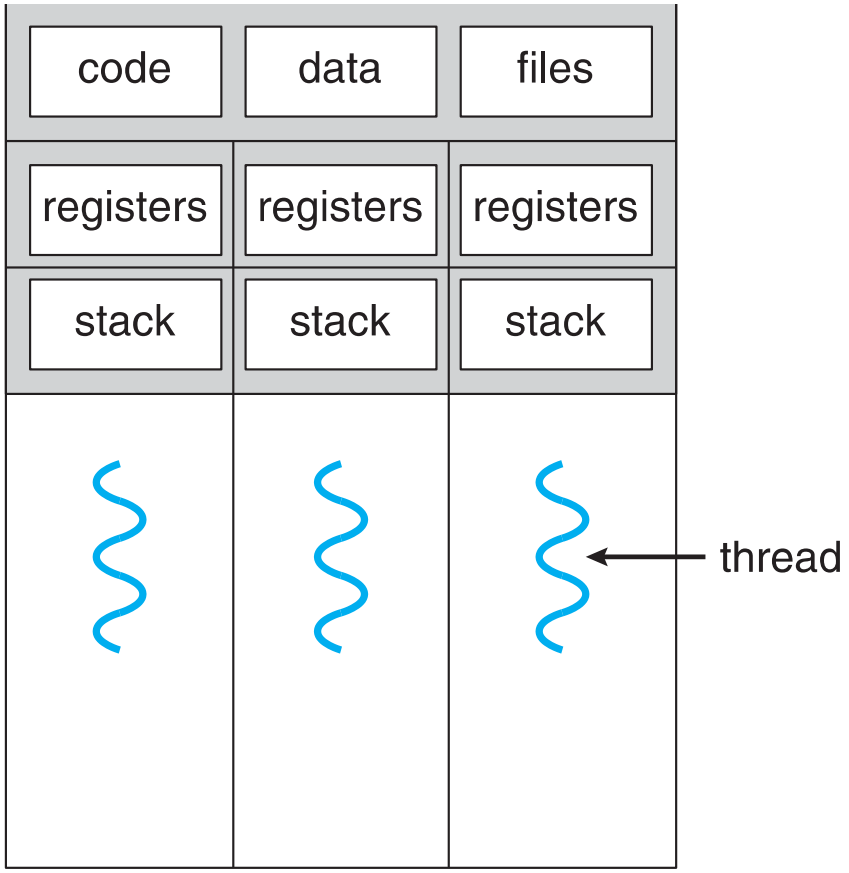


in practice, no application is strictly data or task parallel

Multithreaded Processes



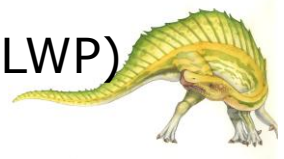
single-threaded process



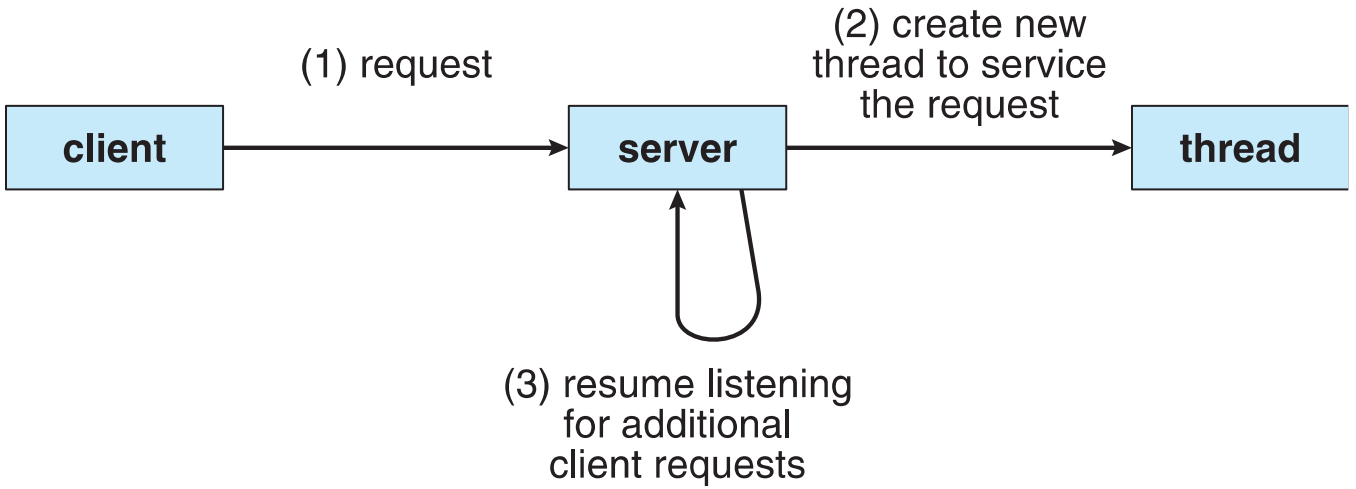
multithreaded process

heavy-weight processes

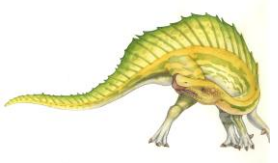
light-weight processes(LWP)



Multithreaded Server Architecture



- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

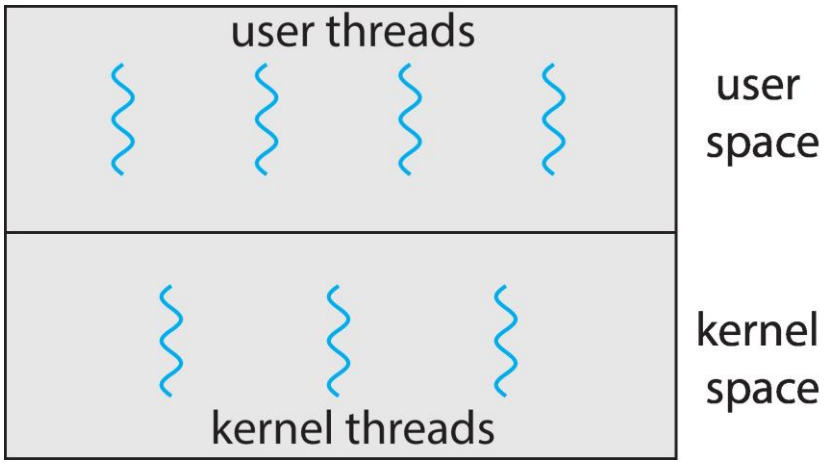


Benefits of threads

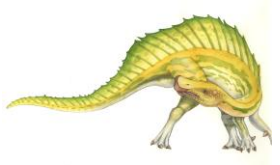
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

User Threads and Kernel Threads

- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

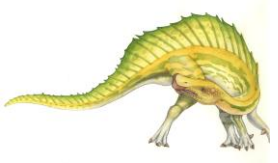


- Support for threads can be either provided at user level or at kernel level
  - **User threads** - management done by user-level threads library
    - user-level scheduler of the library decides which thread to execute. Puts instructions from different threads into a sequence. Kernel is not even aware of the presence of threads.
  - **Kernel threads** - Supported by the Kernel with syscall
    - the creation of a thread (through a thread library) generates a thread in the kernel. No user-level scheduler is needed.
    - Examples: Windows, **Linux,** Mac OS X, iOS, Android
- Transparent to programmer



Multithreading Models

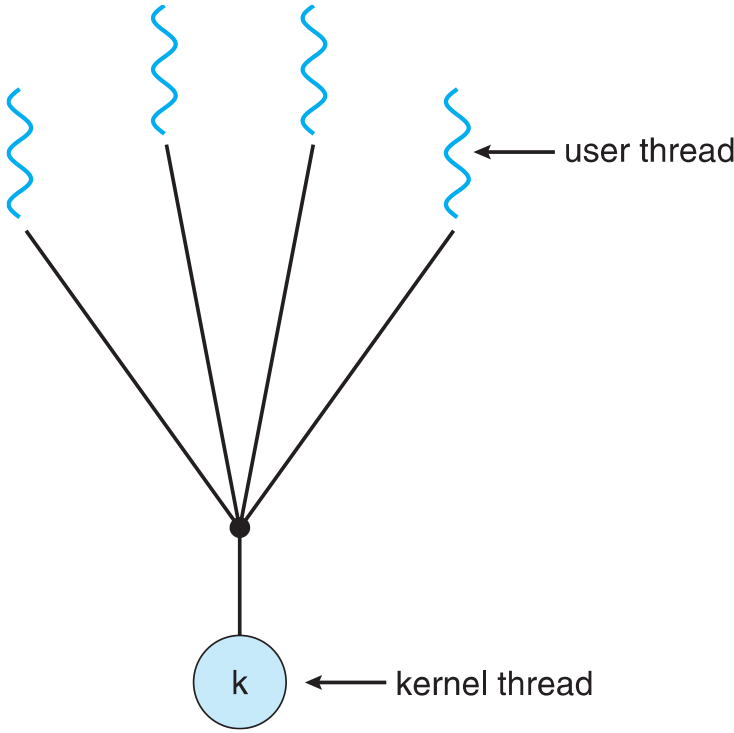
- Relationship between user-level and kernel-level threads?
  - Many-to-One
  - One-to-One
  - Many-to-Many
  - Two-level

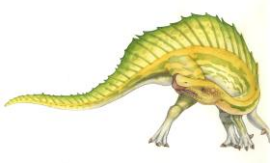


Many-to-One

- Many user-level threads mapped to single kernel thread
- ☺ Efficient thread management in user space, no need to call syscall for creation
- ☹ Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- ☹ One thread blocking causes all to block

- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

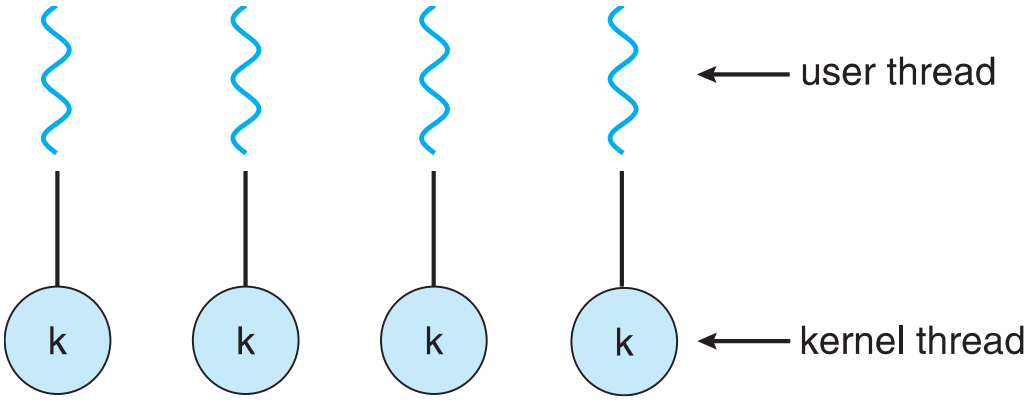


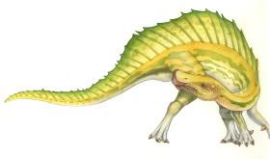


One-to-One

- Each user-level thread maps to kernel thread
- ☺ More concurrency than many-to-one
- ☹ Creating a user-level thread creates a kernel thread (overhead)
  - Number of threads per process sometimes restricted due to overhead

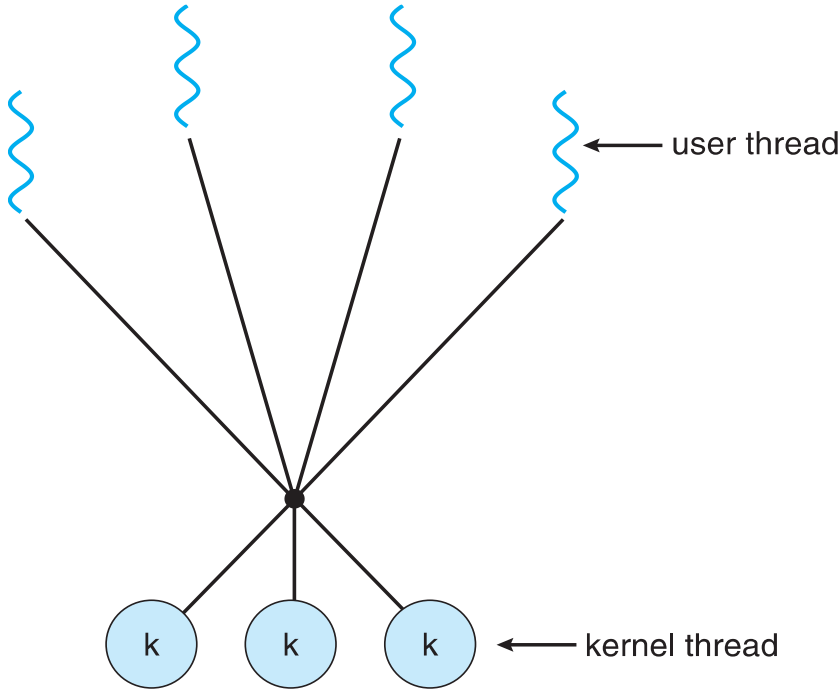
- Examples
  - Windows
  - **Linux**



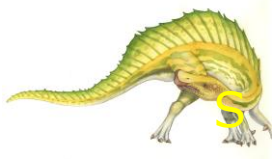


Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- ☺ Allows developer to create as many threads as she wishes, and threads can run concurrently
- ☺ Allows the operating system to create a sufficient number of kernel threads
  - ☺ A thread blocking does not block other threads

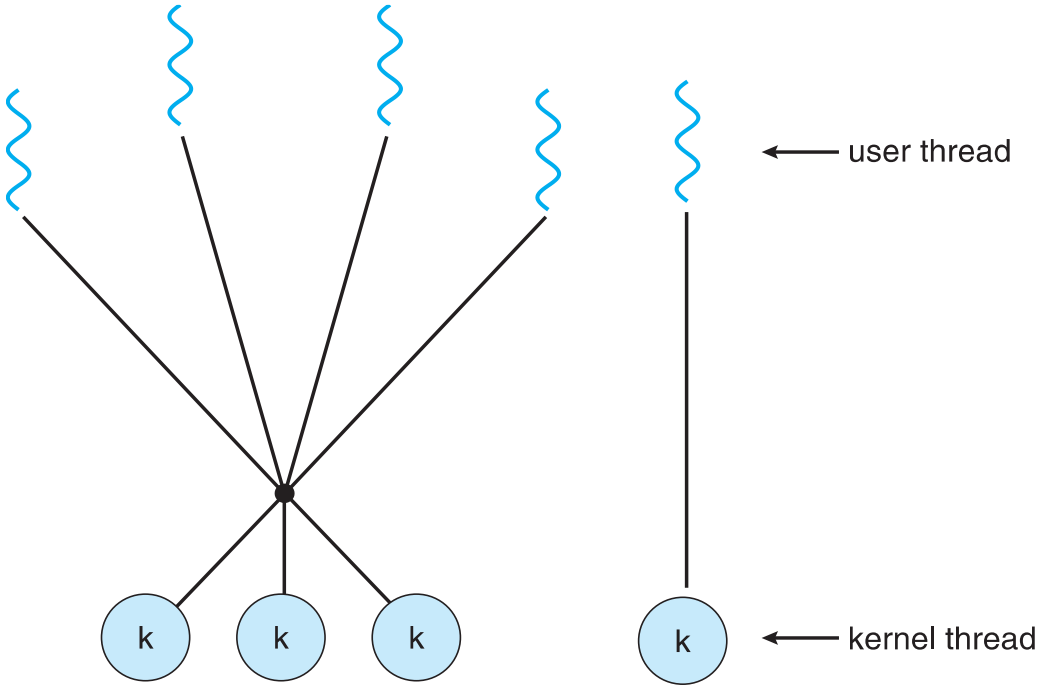


- Examples:
  - Windows NT/2000 with the *ThreadFiber* package
  - Otherwise not very common



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
    - No kernel support
    - No system calls
  - Kernel-level library supported by the OS
    - Code and data structures exist in kernel space
    - API function → system call

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

entry point

thread ID

note: no return value

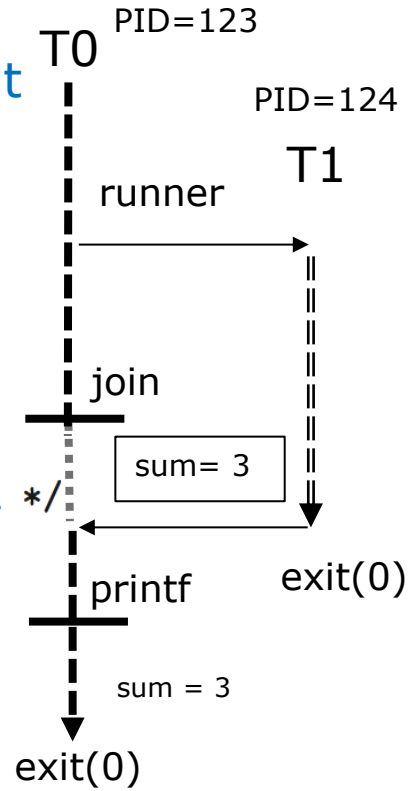
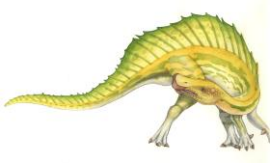


Figure 4.9 Multithreaded C program using the Pthreads API.

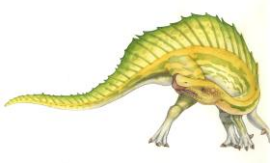


Linux Threads

- **NPTL** (*Native POSIX Thread Library*; formerly: LinuxThreads)
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

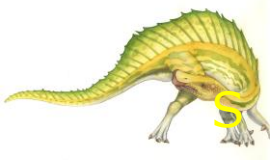
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)



Threading Issues

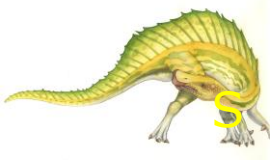
- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred



Semantics of fork() and exec()

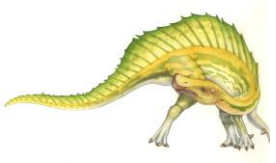
- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork

- **`Exec()`** usually works as normal – replace the running process including all threads



Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process
- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

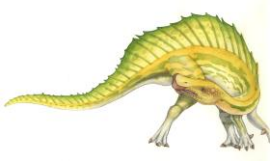


Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread** specified through its **tid**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Dedeferred cancellation** allows the target thread to periodically check if it should be cancelled

- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```



Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals