Process

A **process** is a unit of work; it is a job. The "definition" of a process is: a process is a program in execution (an active entity).

As a process executes, it changes state. Each process may be in one of the following states:

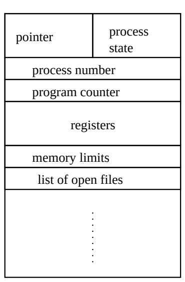
New: the process is being created.

Running: instructions are being executed.

Waiting: the process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Ready: the process is waiting to be assigned to a processor.

Terminated: the process has finished execution.



Process Control Block

Process State: the state may be new, ready, running, waiting, halted, etc.

Program counter: the address of the next instruction to be executed by this process.

CPU registers: depend on the computer architecture, but they include accumulator, index registers, stack pointers. Along with the program counter, this info must be saved when an interrupt occurs.

CPU-scheduling information: includes a process priority, pointers to scheduling queues, and other scheduling parameters.

Memory-management information.

Accounting Info: amount of CPU and real-time used, time limits, account numbers, job or process numbers, etc.

I/O status information: list of I/O devices allocated to this process, a list of open files, and so on.

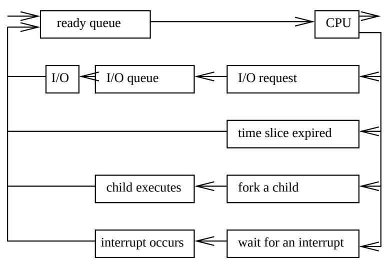
Process Scheduling

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.
- The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

As processes enter the system they are put into a **job queue**. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first (and last) PCB in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.



Queuing-diagram

A process migrates between the various scheduling queues throughout its lifetime. The OS must select processes from these queues; this is done by the appropriate **scheduler**.

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a **context switch**. The context of a process is represented in the PCB of that process.

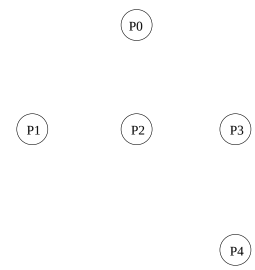
Context-switch times are dependent on hardware support. For instance, the Sun UltraSPARC provides multiple sets of registers, so a context-switch simply includes changing the pointer to the current register set. (When active processes exceed register sets, then the system must resort to copying register data to and from memory.)

Context switching has become such a performance bottleneck that programmers are using threads to avoid them whenever possible.

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus the OS must provide a mechanism for process creation and termination.

A process (**parent**) may create several new processes (**children**).

For example, in Linux:



P0 creates P1,P2, and P3, and then P3 in turn creates P4.

The descriptor of each process P includes the following fields:

`p_opptr` points to the process that created P or to Process 1 (init) if the parent process no longer exists. For example, if a shell user starts a background process and exits the shell, the background process becomes child of init.

`p_cptr` points to the youngest child of P.

`p_ysptr` points to the youngest sibling, that is, to the process that has been created immediately after P by P's parent.

`p_osptr` points to the older sibling.

When a process creates a new process, two possibilities exist in terms of execution:

1. the parent continues to execute concurrently with its children

2. the parent waits until some or all of its children have terminated

There are two possibilities in terms of the address space of the new process:

1. the child process is a duplicate of the parent process

2. the child process has a program loaded into it

In Linux (UNIX) each process is identified by its **process identifier**, which is a unique integer. For example (`ps -A`):

PID	CMD
1	init
2	keventd
3	kapm-idled
4	ksoftirqd_CPU0
5	kswapd
6	kreclaimd
7	bdflush
8	kupdated
9	mdrecoveryd
13	kjournald
88	khubd
181	kjournald
712	adsl-connect
725	pppd
726	pppoe
850	syslogd
855	klogd
875	portmap
903	rpc.statd
1015	apmd

Init: is the parent of all processes. Its primary role is to create processes from a script stored in the file /etc/inittab. This file usually has entries which cause init to spawn gettys on each line that users can log in.

(`top`)

PID	PRI	SIZE	SHARE	STAT	CPU	MEM	COMMAND
1328	17	15916	2268	S	2.3	8.9	X
2294	11	1044	836	R	0.1	0.8	top
1	8	120	68	S	0.0	0.0	init
2	9	0	0	SW	0.0	0.0	keventd
3	9	0	0	SW	0.0	0.0	kapm-idled
5	9	0	0	SW	0.0	0.0	kswapd
6	9	0	0	SW	0.0	0.0	kreclaimd
7	9	0	0	SW	0.0	0.0	bdflush
8	9	0	0	SW	0.0	0.0	kupdated
9	-1	0	0	SW	0.0	0.0	mdrecoveryd
13	9	0	0	SW	0.0	0.0	kjournald
88	9	0	0	SW	0.0	0.0	khubd
181	9	0	0	SW	0.0	0.0	kjournald
712	9	616	456	S	0.0	0.5	adsl-connect
725	8	528	392	S	0.0	0.4	pppd
726	9	100	56	S	0.0	0.0	pppoe
850	9	212	156	S	0.0	0.1	syslogd
855	9	988	336	S	0.0	0.6	klogd
875	9	84	4	S	0.0	0.0	portmap
903	9	384	268	S	0.0	0.2	rpc.statd
1015	9	212	188	S	0.0	0.1	apmd
1080	9	172	4	S	0.0	0.0	cardmgr
1113	9	696	512	S	0.0	0.4	xinetd
1139	8	624	456	S	0.0	0.4	lpd
1167	9	1436	880	S	0.0	0.9	sendmail
1186	9	100	44	S	0.0	0.0	gpm
1204	9	224	156	S	0.0	0.1	crond
1254	9	3988	1012	S	0.0	2.7	xfs
1290	9	100	44	S	0.0	0.0	atd
1313	9	64	4	S	0.0	0.0	mingetty
1314	9	64	4	S	0.0	0.0	mingetty
1315	9	64	4	S	0.0	0.0	mingetty
1316	9	64	4	S	0.0	0.0	mingetty
1317	9	64	4	S	0.0	0.0	mingetty
1318	9	64	4	S	0.0	0.0	mingetty
1319	9	1380	1232	S	0.0	1.0	gdm
1327	9	1912	1564	S	0.0	1.4	gdm

PID The process ID of each task.

PRI The priority of the task.

SIZE The size of the task's code plus data plus stack space, in kilobytes.

SHARE The amount of shared memory used by the task.

STAT The state of the task is shown here. The state is either S for sleeping, D for uninterruptible sleep, R for running, Z for zombies, or T for stopped or traced.

CPU The task's share of the CPU time since the last screen update, expressed as a percentage of total CPU time per processor.

MEM The task's share of the physical memory.

COMMAND The task's command name.

Forking

Before you long in, your terminal is monitored by a

getty

process. After you log in, the getty process dies (a new one is started by the kernel when you log out) and your terminal is managed by your shell, which is a different process. The shell creates a new process each time you enter a command. The creation of a new process is called

forking

because one process splits into two.

If you are using the X Window System, each process starts up one or more windows. Thus, the window in which you are typing commands is owned by an xterm process. That process forks a shell to run within the window. And that shell forks yet more processes as you enter commands.

Thus, new processes are created by the `fork` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

Both processes (parent and child) continue execution at the instruction after the fork system call, with one difference: the return code for the fork system call is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

For example:

```
#include <stdio.h>

void main()
{
        int pid;
        pid = fork();
        if (pid < 0){
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0){
                execlp("/bin/ls","ls",NULL);
        }
        else{
                wait(NULL);
                printf("Child Complete\n");
                exit(0);
        }
}
```


The first argument of `execlp` is the filename of the command. The second argument is the command name. The third is the list of arguments for the command.

The `execlp` call overlays the existing program with the new one, runs that, then exits. The original program gets control back only when there is an error, for example if the file can't be found or is not executable.

The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execlp, which does the ls and then it dies. In the parent, fork returns non-zero so it skips the execlp to the last condition. (If there is an error, fork returns -1.)

The parent waits for the child to terminate before continuing itself; this is done with the system call wait. The wait system call return the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates, all its children have assigned as their new parent the init process.

A process terminates when it finishes executing its final statement and asks the OS to delete it by using the `exit` system call. At that point, the process may return data (output) to its parent process (via the `wait` system call). All the resources of the process (including physical and virtual memory, open files, and I/O buffers) are deallocated by the OS.

Termination can also occur if a parent process causes an `abort` sustem call. A parent may terminate the execution of one of its children for a veriety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the OS does not allow the child to continue without the parent.

Cooperating Processes

The concurrent processes executing in the OS may be independent or cooperating. A process is **independent** if it cannot affect or be affected by the other processes executing in the system (a process that does not share data with other processes is independent).

A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is cooperating.

The producer-consumer problem is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by the **consumer** process. For example:

A print program produces characters which are consumed by the print driver.

When you invoke GNU CC, it normally does:
1. preprocessing (produces macro substiution, file inclusion, etc.)
2. compilation (produces assembly code)
3. assembly (produces object code)
4. linking

To allow produces and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item, while the consumer is is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item which has not yet been produced.

The buffer may be either provided by the OS through the use of an **interprocess-communication (IPC)** facility, or be explicitly coded by the application programmer with the use of shared memory.

Shared-memory solution to the bounded-buffer problem

```
#define BUFFER_SIZE 10

typedef struct { ... } item;

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

The producer process has a local variable `nextProduced` in which the new item to be produced is stored:

```
while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* buffer is full */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

The consumer process has a local variable `nextConsumed` in which the item to be consumed is stored:

```
while (1) {
        while (in == out)
                ; /* buffer is empty */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
```


PIPES
(an example of an IPC facility)

Pipes are an interprocess communication mechanism that is provided in all flavors of UNIX. A **pipe** is a one-way flow of data between processes: all data written by a process to the pipe is routed by the kernel to another process, which can then read it.

In UNIX command shells, pipes can be created by the means of the | operator. For instance, the following statement instructs the shell to create two processes connected by a pipe:

```
$ ls | more
```

The standard output of the first process, which executes the ls program, is redirected to the pipe; the second process, which executes the more program, reads its input from the pipe.

Note that the same result can also be obtained by issuing two commands such as the following:

```
$ ls > tmp
$ more < tmp
```

The first command redirects the output of ls into a regular file; then the second command forces more to read its input from the same file.

Using pipes instead of temporary files is usually more convenient since:

- The shell statement is much shorter and simpler.
- There is no need to create temporary files, which must be deleted later.

Pipes may be considered open files that have not corresponding image in the mounted filesystem. A new pipe can be created by means of the

`pipe()`

system call, which returns a pair of file descriptors. The process can read form the pipe by using the

`read()`

system call with the first file descriptor; likewise, it can write into the pipe by using the

`write()`

system call with the second descriptor.

POSIX defines only half-duplex pipes, so even though the `pipe()` system call returns two file descriptors, each process must close one before using the other. If a two-way flow of data is required, the processes must use two different pipes by invoking `pipe()` twice.

SVR4 allows full-duplex.

Resume the example: `ls | more`

When the command shell interprets the `ls | more` statement, it essentially performs the following actions:

1. Invokes the `pipe()` system call; assume that `pipe()` returns the file descriptor 3 (the pipe's *read channel*) and 4 (the pipe's *write channel*).
2. Invokes the `fork()` system call twice.
3. Invokes the `close()` system call twice to release file descriptors 3 and 4.

The first child process, which must execute the ls program, performs the following operations:

1. Invokes `dup2(4,1)` to copy file descriptor 4 to file descriptor 1. From now on, file descriptor 1 refers to the pipe's write channel.
2. Invokes the `close()` system call twice to release file descriptors 3 and 4.
3. Invokes the `execlp()` system call to execute the `/bin/ls` program. By default, such a program writes its output to the file having descriptor 1 (the standard output), that is, it writes into the pipe.

The second child process must execute the more program; therefore, it performs the following operations:

1. Invokes `dup2(3,0)` to copy file descriptor 3 to file descriptor 0. From now on, file descriptor 0 refers to the pipe's read channel.
2. Invokes the `close()` system call twice to release file descriptors 3 and 4.
3. Invokes the `execlp()` system call to execute `/bin/more`. By default, that program reads its input from the file having file descriptor 0 (the standard input); that is, it reads from the pipe.

Application programmers have a variety of needs that call for different communication mechanisms. Here are the basic IPCs for Linux:

- Pipes and FIFOs: Best suited to implement producer-consumer interactions among processes. Some processes fill the pipe with data while other extract data from the pipe.
- Semaphores: the User Mode version of the kernel semaphores.
- Messages: Allow processes to exchange messages (short blocks of data) in an asynchronous way. They can be thought of as signals carrying additional information.
- Shared memory regions: Best suited to implement interaction schemes in which processes must share large amounts of data in an efficient way.
- Sockets: They were introduced initially to allow data communication between application programs and the network interface. They can also be used as a communication tool for processes located on the same host computer; the X Window System graphic interface, for instance, uses a socket to allow client programs to exchange data with the X server.

Threads

A **thread**, sometimes called a **lightweight process (LWP)** is a basic unit of CPU utilization. It comprises:

–thread ID
–program counter
–register set
–stack

It shares with other threads belonging to the same process its code section, data section, and other OS resources, such as open files and signals.

Many software packages are **multi-threaded**. They are implemented as a single process with several threads of control.

Example: a web browser might have one thread displaying images or text, while another thread retrieves data from the network. A server may be running several threads to service different requests from clients.

The benefits of multithreaded programming can be broken into four major categories:

–Responsiveness: multi-threading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.

–Resource sharing: threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.

–Economy: in general it is much more time consuming to create and manage processes than threads. It is not as costly to context switch threads.

–Utilization for multiprocessor architectures: each thread may be running in parallel on a different processor. A single threaded process can only run on one CPU, no matter how many are available.

User Threads: are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel.

Kernel Threads: are supported directly by the OS: the kernel performs thread creation, scheduling, and management in kernel space. Since the kernel is doing the scheduling, if a thread performs a blocking call, the kernel can schedule another thread in the application for execution.

Linux provides a

`fork`

system call with the traditional functionality of duplicating a process. Linux also provides the

`clone`

system call that is analogous to creating a thread. `clone` behaves much like `fork`, except that instead of creating a copy of the calling process, it creates a separate process that shares the address space of the calling process. It is through this sharing of the address space of the parent process that a cloned task behaves much like a separate thread.

Linux does not distinguish between processes and threads. In fact, Linux often uses the term **task** when referring to a flow of control within a program.

Aside from the cloned process, the Linux kernel does not support multi-threading. However, various Pthreads implementations are available for user-level multi-threading.

Pthreads

Pthreads refers to the POSIX standard definign an API for thread creation and synchronization. OS designers may implement the specification in any way they wish. Generally, libraries implementing the Pthread specifications are restricted to UNIX OSs.

In Linux, the libaray specifications for Pthreads are located in `/usr/include/pthread.h`; Linuxthreads is a simple `clone()`-based implementation of Posix threads.

We give as example a C program that creates a separate thread that determines the summation of a non-negative integer. In a Pthread program, separate threads begin execution in a specified function.


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

3  int sum;
4  void *runner(void *parm);

5  main(int argc, char *argv[])
6  {
7         pthread_t tid;
8         pthread_attr_t attr;
9
10        pthread_attr_init(&attr);
11        pthread_create(&tid,&attr,runner,argv[1]);
12        pthread_join(tid,NULL);
13        printf("sum = %d\n",sum);
14 }

15 void *runner(void *parm)
16 {
17        int i;
18        sum = 0;
19        for (i = 1; i <= atoi(parm); i++)
20                sum += i;
21        pthread_exit(0);
22 }

$ gcc example.c -o example -lpthread
$ ./example 20
sum = 210
```


1 Include the Pthreads library; all Pthread programs must include `pthread.h`

5 When the program begins, a single thread of control begins in `main`. After some initialization, `main` creates a second thread that begins control in the `runner` function.

7 Declare the identifier for the thread we will create.

8 Each thread has a set of attributes including stack size and scheduling information.

10 Set the attributes; because we did not explicitly set any attributes, we use the default attributes provided.

11 A separate thread is created: we pass the thread identifier, the attributes for the thread, the function name where the thread will start execution (`runner` function), and the integer parameter provided on the command line (`argv[1]`).

12 At this point, the program has two threads: the initial thread in `main` and the thread performing the summation in the runner function. After creating the second thread, the main thread will wait for the runner thread to complete by calling the pthread_join function. The runner thread will complete when it calls the function pthread_exit.

The pthreads library is a set of C calls that provides a mechanism for writing multithreaded code. There are three major classes of calls in this library:

1. calls to create and destroy threads
2. calls to synchronize threads and lock program resources
3. calls to manage thread scheduling

Thread scheduling can considerably complicate your algorithm, and it can also get you in trouble when migrating the code from uniprocessor to multiprocessor machines.

Each thread sees the same global data and files; however, each thread has its own stack and registers.

```
+-------------------------------------------------------+
|                                                       |
|                        Process                        |
|                                                       |
|+-------+  +------------+ +------------+ +------------+|
|| Files |  | Thread     | | Thread     | | Thread     ||
|+-------+  |+----------+| |+----------+| |+----------+||
|           || Registers|| || Registers|| || Registers|||
|           |+----------+| |+----------+| |+----------+||
|           |            | |            | |            ||
|.......................................................|
|           |            | |            | |            ||
|  Memory   |            | |            | |            ||
|           | +--------+ | | +--------+ | | +--------+ ||
|+--------+ | | Stack  | | | | Stack  | | | | Stack  | ||
||  Heap  | | |        | | | |        | | | |        | ||
|+--------+ | |        | | | |        | | | |        | ||
|           | |        | | | |        | | | |        | ||
|+--------+ | |        | | | |        | | | |        | ||
|| Static | | |        | | | |        | | | |        | ||
|+--------+ | |        | | | |        | | | |        | ||
|           | |        | | | |        | | | |        | ||
|+--------+ | |        | | | |        | | | |        | ||
||  Code  | | +--------+ | | +--------+ | | +--------+ ||
|+--------+ +------------+ +------------+ +------------+|
|                                                       |
|                                                       |
+-------------------------------------------------------+
```

From the DECthreads manual, from the POSIX Threads tutorial by Mark Hays.

Another example of thread creation and destruction: the “Hello, world” program. The flow of the program is as follows:

```
Time     Master Thread
  |            |
  |            |        create workers with
  |            |        pthread_create()
  |            |
  |          // \\      workers start up
 \|/        / | | \
            | | | |
            | | | |     workers do their jobs
            | | | |
            \ \ / /
             \\ //      workers terminate
               |
               |        join workers with
               |        pthread_join()
               |
         Master Thread
```

Starting from an initial thread in `main()`, we create several worker threads. Each of these will print its message while the original thread waits for them to complete. As each worker thread exits, the master thread will join it and destroy the system resources it uses. Finally, the original thread will call `exit()` and the whole process will terminate.


```
5  #define NTHREADS 4
6  void *hola(void * arg) /* this is the thread code */
7  {
8    int myid=*(int *) arg;
9
10   printf("Hello, world, I'm %d\n",myid);
11   return arg;
12 }
13 /* this is the main thread's code */
14 int main(int argc,char *argv[])
15 {
16   int worker;
17   pthread_t threads[NTHREADS];/* holds thread info  */
18   int ids[NTHREADS];          /* holds thread args  */
19   int *status;                /* holds return code  */
20   /* create the threads */
21   for (worker=0; worker<NTHREADS; worker++) {
22     ids[worker]=worker;
23       pthread_create(&threads[worker],/*thread struct */
24                    NULL, /* default thread attributes */
25                    hola,         /* start routine */
26                    &ids[worker]))/* arg to routine */
27   }
28   /* reap the threads as they exit */
29   for (worker=0; worker<NTHREADS; worker++) {
30     /* wait for thread to terminate */
31     pthread_join(threads[worker],(void *) &status))
32   }
33   return(0);
34 }
```


The program starts in `main()` on line 14.

It will create NTHREADS threads (NTHREADS is defined on line 5) on line 23, each of which starts executing the function `hola()` on line 6.

The main thread waits for each of the worker threads to terminate by calling `pthread_join()` for each thread on line 31.

Meanwhile, each of the worker threads prints a message and its thread ID and then exits. Once the main thread has reaped all of the workers, the main program exits.

Threads are created with the pthread_create() call. The first argument is a pointer to a pthread_t, which is an opaque data structure (i.e., it is a black box) that describes the thread.

Each thread must have a unique pthread_t to describe it. In this case, we store the descriptor for each thread in the array threads[] defined on line 17.

The second argument to pthread_create() is of type pthread_attr_t and specifies the attributes of the thread being created (scheduling priority, etc). Normally you can use the value NULL for this argument.

The third argument is the name of the function in which the thread should start — this function is expected to return a void *.

The last argument is passed untouched to the start routine. In this case, we just pass the thread's number (loop count) to the start routine. In more complicated situations, you can pass a pointer to a block of thread-specific data.

pthread_create() returns 0 on success and an error number on failure.

The main thread next waits for each of the workers to finish in lines 29–32. It calls pthread_join() to wait for the thread to terminate. The thread we're waiting on is specified in the first argument and its exit code is placed into the second argument (declared on line 19 as type int *).

The code that each worker thread executes occupies lines 6–12. First, we cast the argument to the start routine to an int (since we passed a pointer to an int to pthread_create()). The thread then prints its message and returns its thread ID. At this point, the thread terminates and is reaped when the master thread calls pthread_join(). You can also have a thread terminate itself with the pthread_exit() call.

The call pthread_join() causes the calling thread to go to sleep until the specified thread terminates. In the examples given in this document, the master thread cannot proceed until all the workers complete their tasks. Once a thread has been joined, its system resources are reclaimed. If you need to terminate a running thread, use pthread_cancel().

Once you join a thread, you cannot reassign it a new task. The reason is that once a thread has been joined, it has (by definition) terminated. If you need more threads to perform another task, you have to call pthread_create() again.

Aside from returning a thread's exit status, the major function of pthread_join() is to reclaim the system resources used by the terminated thread. If you forget to call pthread_join() in a program that creates many threads, you will run into problems sooner or later (like what happens when you call malloc() many times and never call free()).