Operating Systems

Processes
Based on Ch. 3 of
OS Concepts by SGG

Definition

- A process is a program in execution.
- There can be more than one processes running on the same system (if there are enough cores/memory/load capacity)
  - Even if there is a single core on the system, they share it by taking turns.
- There can be more than one processes that execute the same program.
- (Sometimes processes are called jobs or tasks)

Process Relationships

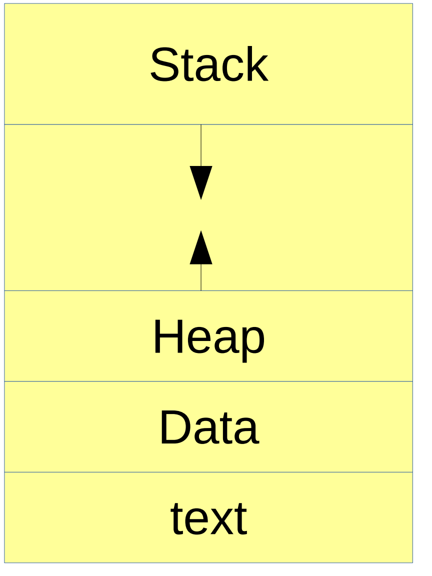
- Processes can spawn other processes
  - The one will be called parent and the other child
- The grand-grand-...-grandmother of all is called init (in Linux and Unix, or systemd on some Linux versions)
  - Execute ps -1
- Before a process dies it kills its children
  - Unless the children take precautions (see sigaction(2))
  - Children that kill their parents become deamons and are adopted by init.
- When a child process dies tells its parent and becomes zombie. Only after the parent acknowledges the death the child is reaped (dead for good) (see wait(2))

Process in Memory

- Each process has its own chunks of memory
- Each chunk can be mapped to the virtual memory or to a file, shared with other processes, etc
- There are many magic tricks (Ch. 9&10) that allow this and many other amazing feats.
- Every process has usually the following:
  - Text: the executable program
  - Data: the static data for the program
  - Heap: dynamic data
  - Stack: the program stack

Process Memory Layout

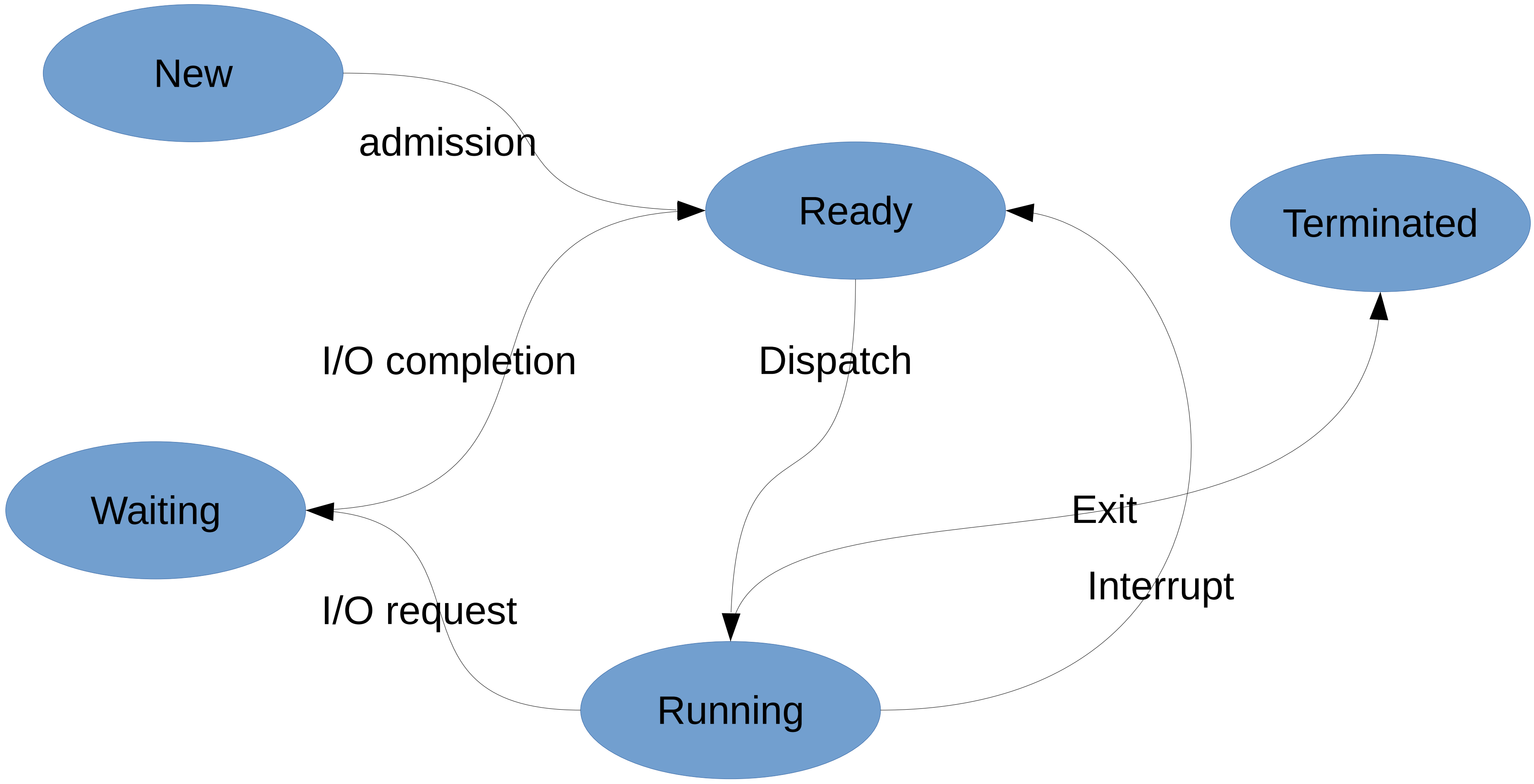
- Simplified model of memory
- We do not consider
  - Multiple threads
  - Kernel memory
  - Memory mappings
  - Unmapped memory
- Address space may be much (MUCH) bigger than physical memory



Process State

- A process can be one of the following states
  - New.
  - Running.
  - Waiting.
  - Ready.
  - Terminated (zombie).
- One process can be in running state per core
  - Or two in a two way hyperthreaded cores.
- To see the list of processes do: ps aux
  - Or cd /proc
  - Filesystem /proc is a virtual (fake) filesystem that instead of files has kernel datastructures.

Process state



Process Control Block

- PCB contain all the info the kernel needs to manage the process
- It should contain:
  - Process ID
  - Program counter
  - CPU registers
  - Scheduling info
  - Memory management info
  - Accounting
  - I/O info
- On a linux box, most of this info is in /proc (the fake filesystem from the previous slide)

Process Scheduling

- There is a component of the OS that does process scheduling
- Maintains a set of queues and lists
  - Job list
  - Ready queue
  - One device queue per device (some systems)
- The scheduler tries to optimize the performance or safety
  - Processes may be kicked out of the CPU early
  - Scheduling may be prioritized.

Context Switching and Swapping

- Context switching is when a process goes in or out of the READY queue
  - Save/restore registers
  - Save/restore memory information
  - Update book-keeping information
- Swapping is when a process is moved to the disk to decongest the system

Process Creation

- Any process can create another process
  - Sometimes is called spawning
  - In Linux/Unix it is called forking (similar but not the same)
- A process needs resources to run. Among them:
  - A terminal/window/whatever
  - Info from the parent
  - Memory space
  - Executable program
- Spawning a process in one step is tricky.

Options for Child

- In terms of execution
  - Child executes concurrently with parent
  - Parent waits until child is dead
- In terms of memory
  - Child gets a duplicate of the parent memory
  - Child shares parent memory
  - Child gets new memory

Fork

- In Unix/Linux fork(2) is the main way to create process
  - See fork -s 2 fork
- Fork creates a new process with:
  - A copy of parent's memory (can be faked)
  - All open files (unless we asked not to)
- The only difference is that fork returns once in the parent with the PID of the child and once in the child with PID=0 (the child can easily find the PID of its parent)
- After forking the child usually replaces its program with another one (by executing execlp or something similar)

Clone

- Linux can also clone
- With clone we can select what parts are shared and what parts are replicated
- Main use is for threads (next chapter)
- It is a very simple and very customizable mechanism

Fork

```
pid_t pid;

pid = fork();
if (pid<0)
{
  fprintf(stderr, “Error...”);
  exit(1);
}
if (pid==0)
{
  execlp(“/bin/ls”,”ls”,NULL);
  fprintf(stderr, “I should not be here...”);
  exit(-1);
}
wait(NULL);
printf(“Parent and child done\n”);
```

CreateProcess

- Windows uses a spawn-type mechanism
  - Needs ten parameters
  - All options are readily available
  - Protects programmer from shooting his own foot.
  - Not as flexible as fork/exec

Process Termination

- Before a process dies it kills its children
  - Unless the children have chosen to (and are able) ignore the kill signal.
  - This allows all the processes to terminate after the user logs out.
- If the children survive the kill signal of a dying parent they are reparented to init.
- If the children die and their parents do not wait(2) they become zombies
  - This way their parent can get some last info from them (PID and status)

Interprocess Communication

- We need to have mechanisms for processes to communicate
  - Information Sharing
  - Computation speedup
  - Modularity
  - Convenience
- Two main mechanisms
  - Shared memory
  - Message passing

Shared Memory

- The two (or more) processes have to have a region of memory they share (one of the magic tricks of modern memory management)
- Then one process can write and the other read
- Sounds much easier than it is!

Producer-Consumer Problem

- The process that writes is the producer and the process that reads is the consumer
- Normally when we read something it is gone (consumed)
- We call the region that is set aside for this “buffer”
- Questions:
  - How does the consumer know there is something to consume
  - How does the producer know there is space left
  - How do they avoid creating a mess

Producer-Consumer

```
while (true)
  {
    /* produce something */
    while (((in+1)%BUFFER_SIZE)==out)
     ; /*do nothing*/

    buffer[in] = something;
    in = (in+1)%BUFFER_SIZE;
  }
```

```
while (true)
  {
    while (in==out)
     ; /*do nothing*/

    food = buffer[out];
    out = (out+1)%BUFFER_SIZE;
    /* consume food */
  }
```

Message-Passing

- Message passing can work even if the system is distributed (and thus there is no shared memory)
- We need (mainly) two operations: send and receive
- The operations can be
  - Direct or indirect
  - Synchronous or asynchronous
  - Zero, finite or infinite buffer size

Direct Messaging

- Process P sends process Q a message
  - `send(Q, message2Q)`
- Process Q receives the message
  - `recieve(P, messagefromP);`
- What could be simpler?
  - There is always **one** direct link between every process pair
  - Every link involves exactly **two** processes
  - They need to know each other's PID.

Indirect Messaging

- We have mailboxes
- In this scheme:
  - A link is established between two processes if they use the same mailbox
  - More than two processes can share a link
  - More than one link can be between two processes
- What happens if three processes share a mailbox:
  - Prohibit it
  - One of them gets the message arbitrarily
  - Allow only one of them to execute receive()

Synchronization

- What happens when we send and the receiver is still busy with something else
  - Blocking send
  - Non blocking send
- Same for receiving
  - Blocking receive
  - Non blocking receive
- If both send and receive are blocking the two processes can have a rendezvous

Buffering

- We can have three kinds:
  - Zero capacity (sender blocks or fails)
  - Bounded capacity
  - Infinite capacity

POSIX InterProcess Communication

- POSIX IPC uses memmory mapped files to share information.
- It may not involve actual files on a disk, but the interface is identical to the file interface
- Typically
  - `fd = shm_open("/minas",O_CREAT|O_RDRW, 0666);`
  - Which is followed usually by
  - `ftruncate(fd, sh_size);`
  - `shm_ptr = mmap(NULL,sh_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);`
  - After this pointer `shm_ptr` points to a block of size `sh_size` which can be shared.

Sockets

- Sockets are end points of communication.
- A socket looks like (almost)
  - news.google.ca:80
- Some services are well known
  - FTP: 21, SSH:22, telnet:23, HTTP:80
- All ports below 1024 are reserved for well known
- Are of two kinds: TCP and UDP

RPC

- Remote Procedure Calls
- When we call a procedure we
  - Specify some code to be executed (function or procedure)
  - Give this code some data (parameters)
  - Wait until the code finishes
  - Get the results
- When we send a message we do more or less the same things.
- Message passing can be used for remote procedure calls

RPC

- To call a remote procedure on some server 259.259.300.300 (yeah, sure) we have to
  - Find the port of the procedure (like a TCP port)
  - Package (“marshal”) the data in a specific format (XDR is a classic)
  - Send the message
  - Block
  - When response arrives, unpack the data.

Pipes

- Early IPC mechanism
- Can be thought of as sockets, sometimes local ones
- Can be uni- or bi-directional, depending on the system and type.
- Can be created or inherited
- There are two types
  - Ordinary or anonymous
  - Named or FIFO

Unix ordinary Pipes

- Can be only inherited
- Are unidirectional and local
- Are extensively used on Unix/Linux
  - Part of the philosophy and culture

Unix Example

```
char wmsg[BUFSZ] = “Hello”;
char rmsg[BUFSZ];
int fd[2];
pid_t pid;

if (pipe(fd)<0) {error...}
pid = fork();
if (pid<0) {error...}
if (pid>0)
{
  close(fd[0]);
  write(fd[1], wmsg, strlen(wmsg)+1);
  close(fd[1]);
}
else
{
  close(fd[1]);
  read(fd[0], rmsg, BUFSZ);
  printf(“Child: %s\n”,rmsg);
  close(fd[0]);
}
```

Bugs:
Did we check write?
Did we check read?

Named Pipes on Unix

- Also called FIFOs
- Look like regular files
- Permissions like files
- No need to share a parent to share a pipe
- Must be on the same machine
- Two FIFOs are need for bidirectional communication
- Can be created with `mkfifo()`
- Std functions like `open`, `close`, `read`, `write` can be used on them.

Named pipes on Windows

- Allow bidirectional communication through a single pipe
- Allow byte oriented or message oriented communication
- The end points can be on a different machine
- Can be created with `CreateNamedPipe()`
- Std functions like `ReadFile()` and `WriteFile()` can be used on them

Signals in Unix/Linux

- Allow a process to signal another
  - There are restrictions to avoid sending malicious or annoying signals
- They are meant to send asynchronous signals
  - But they can be used for synchronous signals as well
- They are built in the system. Much of the functionality of the OS is facilitated by signals
  - SIGINT (interrupt), SIGKILL, SIGALRM, SIGPIPE, SIGCHLD, SIGHUP, SIGSEGV, SIGFPE, etc.
- Main documentation is in signal(7)

Signal Dispositions

- The term means what to do when receiving a signal
- The possible dispositions are:
  - Terminate (possibly with a core dump)
  - Ignore
  - Block (the signal is delivered when unblocked)
  - Stop
  - Continue (if stopped)
  - Catch (with a programmer supplied function)
- All signals come with a default disposition, which in most cases can be modified
  - With the exception of SIGKILL (aka signal 9) and SIGSTOP

Signal Dispositions

- Every process has its own set of dispositions (one for each signal)
- It can be changed with sigaction(2)
  - Also with signal(2) but this should be avoided since it has portability issues
- If we want to block a signal we use sigprocmask(2) or sigvec(3) to change the set (mask) of blocked signals
- If we want to send signals synchronously (suspend until a signal is received) we use sigsuspend.
  - If we want to receive SIGUSR1 synchronously we block it with sigvec(3), then we sigsuspend(2) with a mask that does not block SIGUSR1.