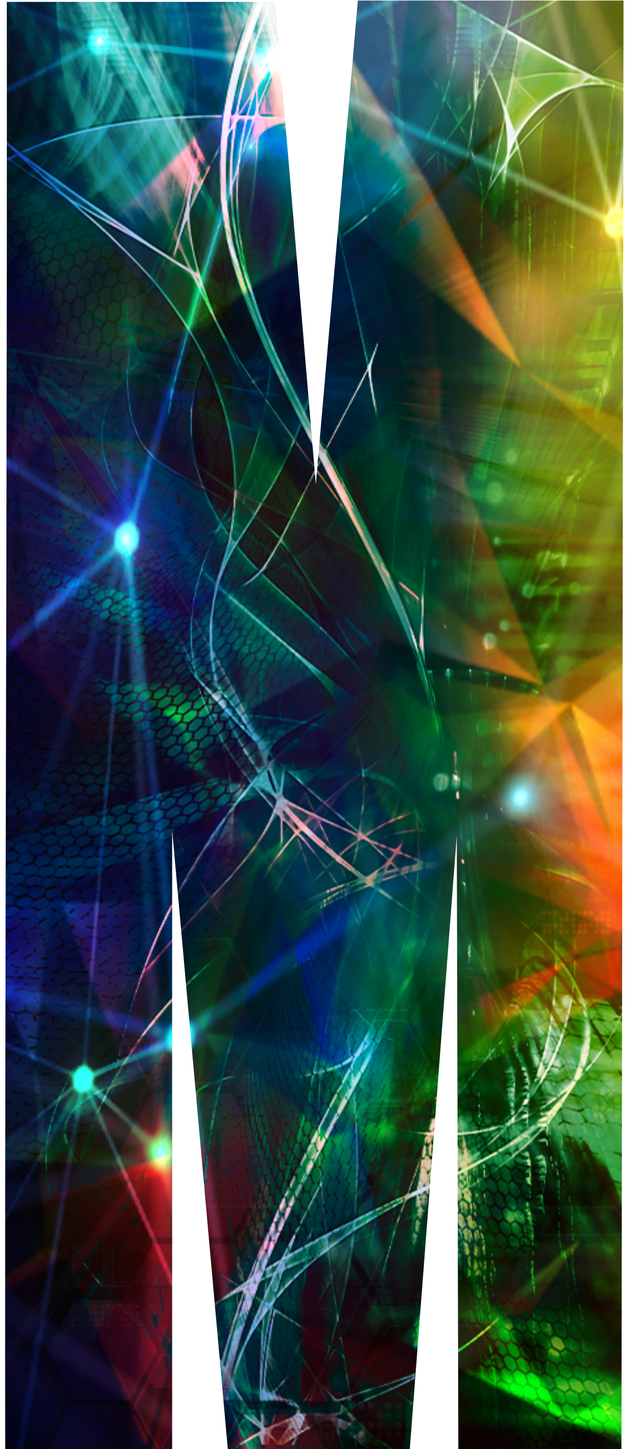
MONASH University

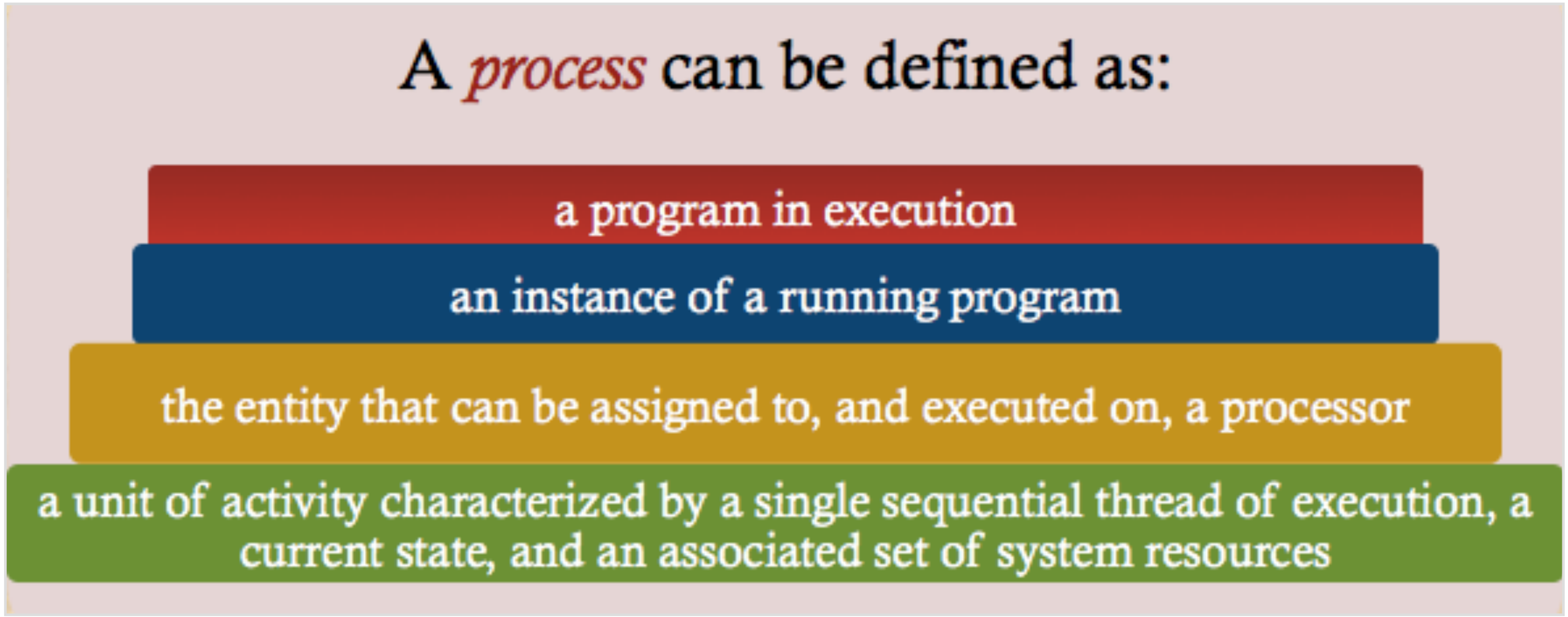
FIT2100 Operating Systems

Semester 2, 2020

Lecture 5: Processes



The Process Concept



Process Elements

Process Location

- A process must include a program or set of programs to be executed
- A process will consist of at least sufficient memory — to hold the programs and data of that process
- Execution of a program typically involves a stack — to keep track of procedure calls and parameter passing between procedures

Process Attributes

- Each process has associated with it a number of attributes — used by the OS for process control
- Collection of program code, data, stack, and attributes is referred as — ***process image***
- Process image location depends on the memory management scheme being used

Processes in Unix

- A process in Unix is an active entity; a program in execution.
- *Task* is often used to refer to process
- Unix is multitasking (preemptive)
- Processes can change their *state* during their lifecycle.
- Every process in Unix (except *init*) has a parent process.

Sample ps –ely output

```
S   925 20772 20706  0  75   0  1612  2100 -      ?        00:00:00 sshd
S   925 20774 20772  0  78   0  1908  1140 -      pts/2    00:00:00 bash
R  1003 20991 20547  0  77   0   816   587 -      pts/1    00:00:00 ps
S  1003 20992 20547  0  78   0   908   779 -      pts/1    00:00:00 more
S  1003 20993 31430  0  75   0   624   700 -      ?        00:00:00 sleep
S   928 26262     1  0  76   0   424   705 -      ?        00:00:00 script
S   928 26263 26262  0  76   0   556   428 -      pts/8    00:00:00 sh
T   928 26277 26263  0  76   0  1820  1071 finish pts/8    00:00:00 vi
T   928 26737 26263  0  76   0  1900  1114 finish pts/8    00:00:00 vi
```

Effective UID

- Effective UID determines the access processes have to files:

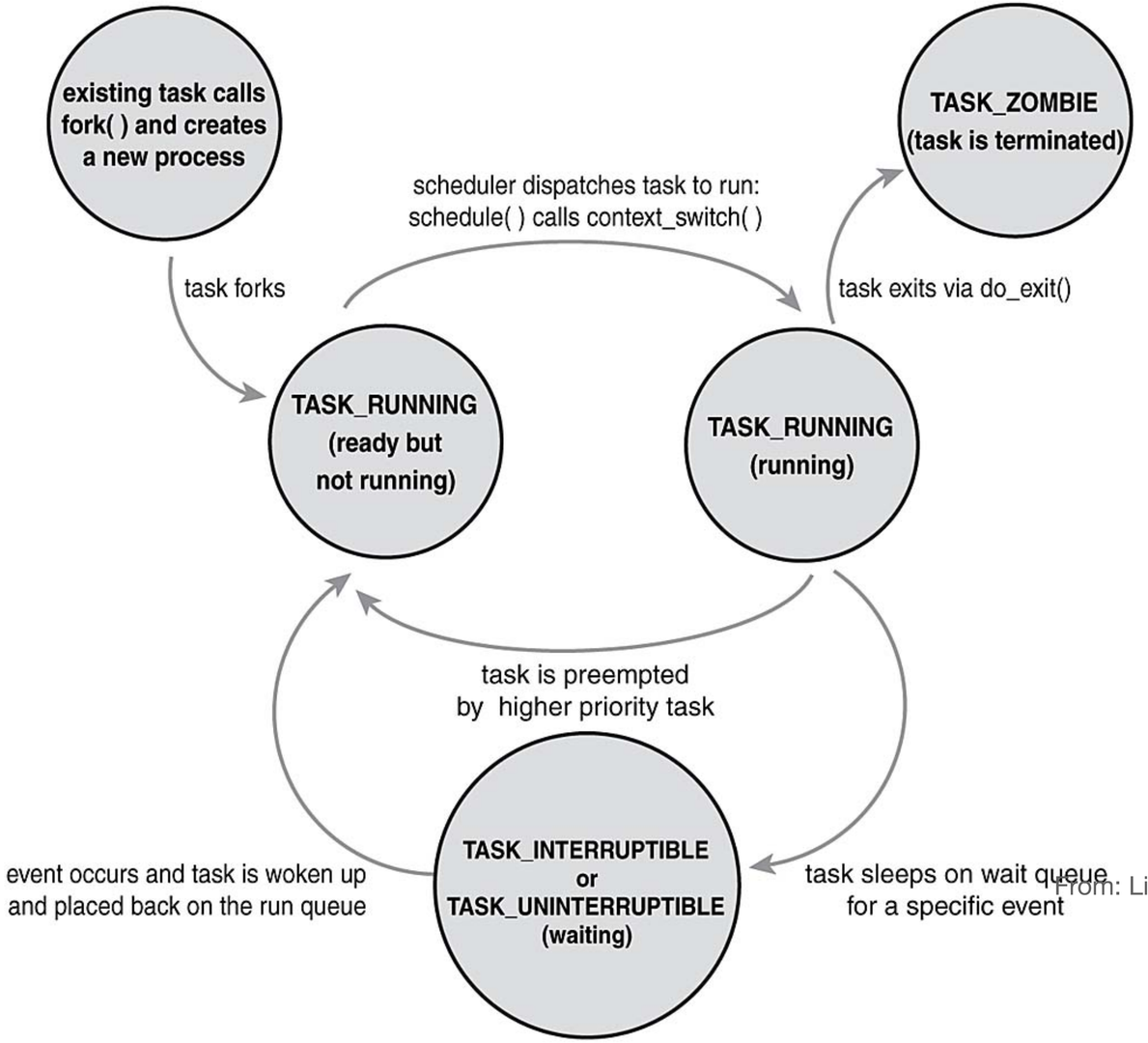
```
cwilson@ubuntu:/usr/src/linux-headers-5.4.0-42-generic/include/linux$ passwd cwilson
Changing password for cwilson.
Current password:
```

```
cwilson@ubuntu:/usr/src/linux-headers-5.4.0-42/include/linux$ ls -l /etc/passwd
-rw-r--r-- 1 root root 3112 Aug 25 19:58 /etc/passwd
cwilson@ubuntu:/usr/src/linux-headers-5.4.0-42/include/linux$ ps -ely | grep passwd
S     0   11978    5698  0  80   0  3396  4787 -      pts/0    00:00:00 passwd
cwilson@ubuntu:/usr/src/linux-headers-5.4.0-42/include/linux$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 May 27 23:37 /usr/bin/passwd
cwilson@ubuntu:/usr/src/linux-headers-5.4.0-42/include/linux$
```

Process state (Linux)

- Processes are in one of 5 states:
  - TASK_RUNNING
    - Process is eligible to run on the CPU(s).
  - TASK_INTERRUPTIBLE
    - Process is sleeping (blocked) waiting for some event or condition. When the event occurs, the kernel sets the state back to TASK_RUNNING (i.e. *wakes it up*). Task will also change state to TASK_RUNNING if it receives a signal.
  - TASK_UNINTERRUPTIBLE
    - Process is sleeping but does not respond to signals (e.g. it should not be interrupted during some important IO operation). Does not respond even to SIGKILL.
  - TASK_ZOMBIE
    - Child process has terminated but the parent has not yet issued a wait() system call. Remember – if the parent process exits, the child process is inherited by *init*.
  - TASK_STOPPED
    - Process execution has finished; the process is not eligible to run.

Process state (Linux)



From: Linux Kernel Development

By Robert Love

Process IDs

- Every process has a unique PID.
- This PID is used in various commands, most notably ps and kill.
- The way that Unix creates new processes is by creating copies of existing processes. We will see how this process creation works shortly by looking at the *fork* and *exec* system calls.

More on PID/PPID – Process hierarchy

- Since new processes are created by existing processes, upon creation of a new process, the existing process is the parent, the new process is the child.
- Therefore, each process also has a *parent process ID* (PPID)
- Process with pid 0 is the scheduler.
- Process with pid 1 is *init*, responsible for booting the system, and is the ancestor of all the other processes in the system.

Scheduling in Unix

- Unix process scheduling is designed to favour interactive processes.
- Every process is associated with a *priority* – smaller numbers indicate higher priorities. We will use PRI to refer to the priority value as shown in ps/top
- Processes with negative PRI are important system processes (owned by root user)
- User processes have positive PRI values.

Scheduling in Unix

(More on scheduling next week!)

- PRI value increases as the process accumulates more CPU time (to minimize CPU hogging behaviour)
- Process aging employed to prevent starvation.

Nice

- Users can alter the default PRI value of their processes by starting the process with the *nice* command:

  nice –n *command commandargs*

- Alter the PRI value of running processes using *renice*

Processes

- In multitasking operating systems, processes are typically associated with a *process control block* (PCB) which contains information about the process.
- Typical PCB would contain amongst other attributes:
  - Process ID (PID)
  - Parent process ID (PPID)
  - Process state
  - CPU registers (e.g. accumlator, stack pointer etc)
  - Program counter
  - Process priority
  - UID, EUID, GID, EGID
  - Pointers to open files
  - etc\
- The process table (a logical structure) keeps track of all the PCBs.

Processes in Linux

- In Linux the PCB is of type struct task_struct
- The process table is a linked list of these task_structs…

Process and File Description Table

- All Unix processes (except PID 0) are created using the fork() system call.
- When a process is created by *fork()*, it contains duplicate copies of the text, data and stack segments of its parent.
- Processes have a File Descriptor Table that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file.

Context Switch

- When the kernel decides that it should execute another process (e.g: The kernel wants an IO bound process which has issued an I/O operation to give up the CPU) , it does a context switch, so that the system can execute another process.
- When doing a context switch, the kernel saves enough information so that it can later switch back to the first process and resume its execution. That is 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.
- Context-switch time is overhead; the system does no useful work while switching.
- Overhead time mainly dependent on hardware support.

Process in user and kernel modes

- A UNIX process can run in two modes:
  - **User mode**
    - Can access its own instructions and data, but not kernel instruction and data
  - **Kernel mode**
    - Can access kernel and user instructions and data
- When a process executes a system call, the execution mode of the process changes from user mode to kernel mode.
- When moving from user to kernel mode, the kernel saves enough information so that it can later return to user mode and continue execution from where it left off.
- *Process running in kernel mode cannot be preempted by other processes.*

Termination status

- Three ways for a process to terminate normally:
  - return from main()
  - exit() – closes *stdio* streams, then calls _exit()…
  - _exit() – closes open files, sends a signal to the parent process, set's ppids of child processes to 1, frees up IPC resources etc. Generally we don't call _exit() directly.
- Can register exit handler functions with atexit() which are then called when exit() is called.

Termination status

- Two ways for a process to terminate abnormally:
  - Call abort()
  - Receive a signal from itself, the kernel or another process.
- When a process terminates, the kernel provides a termination status to the parent. On normal termination, this is an *exit status* used to indicate success, failure (defined by the process). On abnormal termination, the termination status tells the parent how the process terminated (i.e. what signal it received) and whether a core dump was produced.
- It is up to the parent process to request the termination status (if it wants to) via the wait() system call.

Zombies and Orphans

- Remember it is up to the parent to request the termination status of the child.
- When the child terminates, Unix frees up all the resources used by the process but keeps its entry in the *process table*. This entry can be used to deliver the termination status to the parent when (and if) the parent requests it via *wait*().
- When the termination status has been delivered to the parent, the process table entry is deleted (marked for reuse).
- During the time between the termination of the child and the parent receiving the termination status, the child is a *zombie*.

Zombies and Orphans

- If the parent process terminates *before* the child process, the child process becomes an *orphan*. In this case, the process is *adopted* by *init* (i.e. the ppid for the process becomes 1). This includes zombie processes whose parent has terminated.
- Note: *init* will always call wait() each time one of its child processes to terminates. It thus receives the child termination status so the process does not become a *zombie* forever!

Creating a new process – fork()

- The fork function creates a (more or less) exact ***copy*** of the calling process. The child process inherits (amongst a lot of other things – these are just things relevant to us so far) from the parent:
  - Real and effective uids and gids
  - Variable values (except the return value from fork)
  - All of the parent's open file descriptors and file offsets
  - Parent's *umask*
  - Parent's current working directory, controlling terminal, resource limits

fork()

- Child process differs from parent in the following ways:
  - Child has unique pid
  - Child will have a different ppid
  - Although the file descriptors are copied (that is the child can close any of these without affecting the parent), the file offset for each descriptor is *shared* – if the child moves forward in the file, so does the parent..if the child writes to the file at the same time, the output could be intermingled.
- There is no guarantee of order of execution (e.g. parent before child) – both processes execute concurrently. Be careful then with deadlock situations?

fork()

- fork returns twice, once in the parent (returning the pid of the newly created child) and once in the child (returning 0). This allows each copy of the process to distinguish itself.
- fork returns -1 on failure.

fork() execution

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{
int a=5;
pid_t pid;
printf("%d\n",a);
pid=fork();
if (pid==0) printf("%d\n",++a);
else if (pid>0) printf("%d\n",--a);
else fprintf(stderr,"fork failed\n");
}
```

fork() example

```
/* spawn.c - demonstrate fork()*/
/* Haviland p. 91 */
#include <sys/types.h>
#include <stdio.h>              /* for printf() */
#include <unistd.h>             /* for fork() */
#include <stdlib.h>             /* for perror() */


int main(void)
{
   pid_t pid;
   printf("Just one process so far\n");
   printf("Calling fork...\n");
   pid = fork();
   if (pid == 0)
      printf("I'm the child\n");
   else if (pid > 0)
      printf("I'm the parent, child has pid %d\n", pid);
   else
      perror("Fork error:");
}
```

The exec() family

- fork() creates a copy of the calling process.
- This is of limited use in isolation.
- To create a new process which is different to the calling process, we use the exec() functions – actually a family of functions.
- All the exec() functions work by overlaying the image of the calling process with the image of the new process.
- The new process is constructed from an executable program or shell script.
- If exec succeeds, it will never return – because there is nothing to return to – the original calling process has been replaced by the new process

exec() example

```
/* runls.c - uses execl to run ls */
#include <unistd.h>
#include <stdio.h>
int main(void)
{  printf(“Executing ls\n”);
    /* Note last arg is null pointer in execl to indicate
    the end of the argument list */
   execl(“/usr/bin/ls”, “ls”, “-l”, (char*)0);
   perror(“exec failed to run ls”);
   exit(1);
}
```

fork(), exec() and wait()

- Using fork and exec together allows a program to create another with obliterating itself.
- In the example, runls3.c, the original process (called the parent) just waits for the new process (the child) to finish what it is doing.
- This is in fact what happens when a command is called from the Unix shell (unless it is called as a background process with an &)

fork(), exec() and wait()

```
#include <sys/types.h>
#include <stdio.h>      /* for printf() */
#include <unistd.h>     /* for fork() */
#include <stdlib.h>     /* for perror() */
#include <sys/wait.h>   /* for wait() */
int main(void)
{
   pid_t pid;
        switch(fork()) {
     case -1 :
        perror("fork failed"); exit(1);
     case  0 :
          execl("/bin/ls", "ls", "-l", (char*)0);
        perror("Failed to exec ls"); exit(1);
     default : /* parent waits for ls to complete */
        wait((int*)0);
        printf("ls completed\n");
        exit(0);
   }
}
```

Signals

- Signals are messages which can be sent to processes.
- The signal itself does not carry any information apart from its own type (an integer representing their normal purpose).
- On the command line, signals can be sent using the kill command.
- signal.h defines the correspondence between the integers representing the signals and their mnemonics.

Processes and signals

- The *disposition* of a signal is the action the kernel will take on behalf of a process upon generation of a particular signal. A process can indicate the disposition of a signal.
- There are 4 possible dispositions:
  - Ignore the signal – the kernel will discard the signal and never deliver it to the process.
  - Block the signal – the kernel will hold on to the signal and only deliver it to the process when the process unblocks the signal.
  - Trap the signal – the kernel will suspend the process and call a *signal handler* function. Process will be resumed when signal handler returns.
  - Implement the default disposition – the default disposition of most signals is to terminate the process – may be normal (e.g. SIGTERM, SIGINT,…) or abnormal (SIGABRT, SIGSEGV, SIGILL,…). If signal caused normal termination, can use wait to find out termination status and what signal caused termination (last week). Most abnormal terminations result in core dump.

Original POSIX signals
Various systems will support more signals

Name	Number	Def Disp	Description
SIGHUP	1	Term	Hangup detected on controlling term. or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Aside: Signal handling in older Unix

- Signal handling in older versions of Unix was *unreliable*.
- 4.2BSD and SVR3 included *reliable* signals. This included the ability to block (hold) signals.
- Modern versions of Unix include reliable signals.

Daemons

- A daemon is a background process that performs a single, fairly simple task.
- A daemon may be started at boot time and run forever, waiting for someone to make a request for its service.
- Example of daemons are:
  - lpd - printer demon to print files
  - crond - execute jobs after a specified time
  - inetd - a network daemon - services include ftp, rsh, telnet, etc.

MONASH University

FIT2100 Operating Systems

Next week - process scheduling…